Testing for Trustworthiness in Scientific Software

Daniel Hook
Queen's University
Kingston Ontario

Diane Kelly
Royal Military College of Canada
Kingston Ontario

Abstract

Two factors contribute to the difficulty of testing scientific software. One is the lack of testing oracles – a means of comparing software output to expected and correct results. The second is the large number of tests required when following any standard testing technique described in the software engineering literature. Due to the lack of oracles, scientists use judgment based on experience to assess trustworthiness, rather than correctness, of their software. This is an approach well established for assessing scientific models. However, the problem of assessing software is more complex, exacerbated by the problem of code faults. This highlights the need for effective and efficient testing for code faults in scientific software. Our current research suggests that a small number of well chosen tests may reveal a high percentage of code faults in scientific software and allow scientists to increase their trust.

1. Introduction

In a 1982 paper, Elaine Weyuker [14] pointed out that we routinely assume a software tester can determine the correctness of the output of a test. The basis of the assumption is that the tester has an oracle, a means of comparing the software's output to expected – and correct – results. Weyuker notes that in many cases, an oracle is pragmatically unattainable. With scientific software, this is almost always the case. Weyuker further comments that if the oracle is unattainable, then "from the view of correctness testing … there is nothing to be gained by performing the test." For scientific software, this is a depressing and disturbing conclusion. However, there is instead a goal of trustworthiness rather than correctness that can be addressed by testing.

In this paper, we first outline a typical development environment for scientific software, then briefly discuss testing. This leads to a discussion of correctness and trustworthiness from the view of the scientist who develops and uses scientific software. We then introduce new results from our analysis of samples of scientific software using mutation testing. We conclude with future directions for research.

2. An Environment for Scientific Software

Scientific software resides in a rich environment that has several layers of complexity that affect how to successfully test the software.

The computer language representation, or software, is the culmination of a series of model refinements each of which adds its own errors and/or

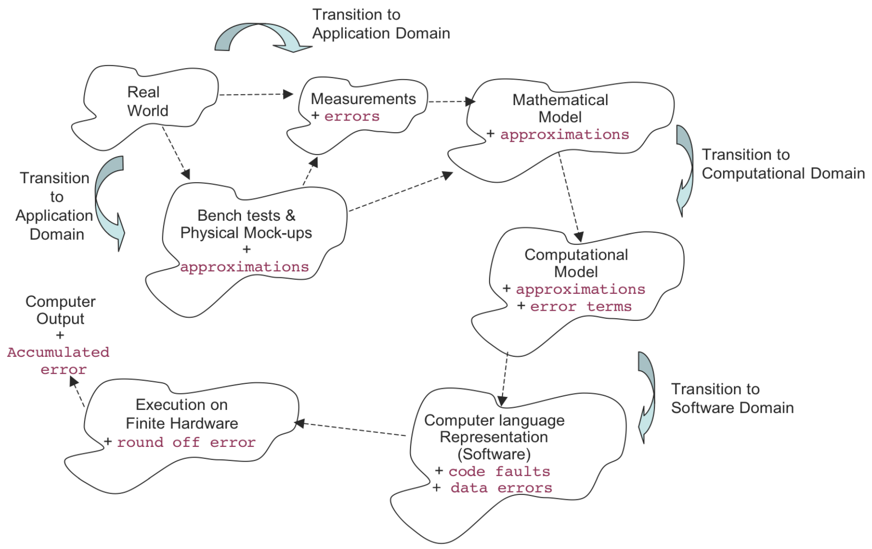
Transition to Application Domain
Real World
Measurements + errors
Mathematical Model + approximations
Transition to Computational Domain
Transition to Application Domain
Bench tests & Physical Mock-ups + approximations
Computational Model + approximations + error terms
Computer Output + Accumulated error
Transition to Software Domain
Execution on Finite Hardware + round off error
Computer language Representation (Software) + code faults + data errors

Figure 1: Contributors to Computer Output Error for Scientific Software

approximations, as shown in Figure 1. The complexity of the refinements is complicated by transitions from one knowledge domain to another. For example, if we were coding an application for aircraft design, we could be using theory from physics, solution techniques from computational fluid dynamics, and algorithms and data structures from computer science. Each of these knowledge domains contributes errors and approximations to the models embedded in the computer code. The final computer output is an accumulation of all such errors and approximations. Assessing correctness of the computer output becomes as complex as analyzing the entire environment shown in Figure 1.

3. Testing Scientific Software

Testing is usually couched in terms of verification and validation. The software engineering definitions of verification and validation, based on process [eg., 6], provide no insight into suitable testing activities for scientific software. To confound things further, verification and validation are not consistently defined across the computational science and engineering communities [eg., 3, 7, 9, 10, 11, 13]. This lack of consistency plus the complexity of the scientific software environment contributes to the omission of a major goal in testing, a goal that should be addressed by what we call code scrutinization. We suggest that for scientific software, there are three separate testing goals that should be addressed, not two.

Validation for scientists primarily means checking the computer output against a reliable source, a benchmark that represents something in the real world. In the literature, validation is described by scientists as the comparison of computer output against various targets such as measurements (of either real world or bench test events), analytical solutions of mathematical models, simplified calculations using the computational models, or output from other computer software. Whether that target is another computer program, measurements taken in the field, or human knowledge, the goal of validation is the same: is the computer output a reasonable proximity to the real world?

Verification is also described as a comparison of the computer output to the output of other computer software or to selected solutions of the computational model. Roache succinctly calls verification "solving the equations right" [11]. This includes checking that expected values are returned and convergence happens within reasonable times. The goal of verification is the assessment of the suitability of the algorithms and the integrity of the implementation of the mathematics.

A third goal of testing, that of searching specifically for code faults, is almost universally missing in the work practices of scientists [eg. 12]. Yet, Hatton and Roberts [5] carried out a detailed study that demonstrated that accuracy degradation due to unnoticed code faults is a severe problem in scientific software. Hatton reiterated this observation in 2007 [4], commenting that problems with code faults in scientific software have not gone away.

We have only come across one technique, developed by Roache and colleagues [7, 10, 11], that specifically addresses faults in scientific code. This technique has been developed for software that solves partial differential equations (pde). Called the Method of Manufactured Solutions, the technique involves manufacturing an exact analytical solution for the computational model. The computer output can be compared to the manufactured solution for accuracy and convergence characteristics. The intent is that any code faults affecting either of these will be detected. The technique is used in niches like computational fluid dynamics, but is limited in its applicability. Its limitations are due to the difference in breadth between analytical solutions and full computational solutions, the need in some cases to alter the code to use the technique, and the fact that pde solvers are only a small fraction of the lines of code that make up the body of computational software.

We suggest a new model for testing, as shown in Figure 2.

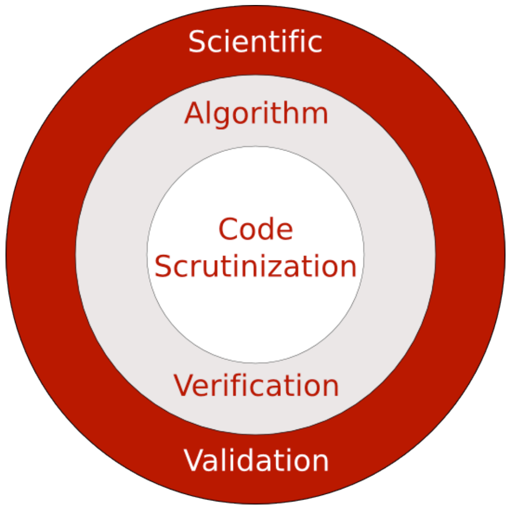

Figure 2: Model of Testing for Scientific Software

Figure 2 shows three circles, or cycles, that represent testing activities whose goals are within the realm of three different specialists.

The outer cycle of testing addresses the need to assess the software against goals in the scientific domain. The ultimate goal of the scientist is to use the software to provide data and insight [13] for problems in his/her domain. This testing addresses the capability of the software as a tool for the scientist. We call this testing activity *scientific validation*.

The next cycle of testing addresses the integrity and suitability of the algorithms and other solution techniques used to provide the scientific solution. This is the domain of the numerical analyst. We introduce the term *algorithm verification* and refine a definition from Knupp and Salari [7]: "*Algorithm verification* is the process by which one assesses the code implementation of the mathematics underlying the science".

The inner-most cycle of testing addresses code faults that arise in the realization of models using a computer language. We call this step in the scientist's testing activity, *code scrutinization*. This step specifically looks for faults such as one-off errors, incorrect array indices, and the like. The goal is to ensure the integrity of the software code. This step is *not* concerned with evaluating the choice of scientific or mathematical models.

Ideally, the order that these testing activities are carried out is from the inner cycle to the outer cycle. A scientist finding problems with a scientific validation test would be hard pressed to decide if the source of the problem is the scientific model, the mathematical model, or a code fault, if neither algorithm verification nor code scrutinization have been carried out.

Code scrutinization is where testing techniques from software engineering could potentially provide benefit. We know of no testing techniques described in software engineering literature that have been specifically developed and/or validated for use with computational software. The goal of our research is to identify such a technique.

4. Correctness versus Trustworthiness

Weyuker states [14] that "[being] able to determine whether or not [test] results obtained are correct ... is the fundamental limitation that testers must face." Correctness is essentially a Boolean decision, either true or false. On the other hand, trust is on a graduated scale, ultimately determined by judgment. The level of trust can be different for every example of scientific software, and possibly different for each individual output data. When a scientist says that a computer output is acceptable within forty percent of benchmark data, he/she is talking about trust. Nothing can be said about the Boolean concept of correctness.

If we consider the three cycles of testing in our model, scientific validation testing is about trust: is the software giving output that we believe? To gain trust, we exercise the software in different ways and compare the output to different benchmarks in the scientific domain. As the output conforms to our expectations, our trust increases.

Similarly for algorithm verification, all we can do is exercise the implementation of our algorithms until our trust is sufficiently high. It is well known that we cannot do exhaustive testing. Scientists have developed a number of approaches to judge trustworthiness of the implementation of their mathematics. For example, checking that quantities subject to the conservation laws are in fact conserved or that the matrix solver returns a message about ill-conditioning when expected.

Only in the testing cycle of code scrutinization can we possibly tackle the Boolean true/false of correctness. For a specific code segment, we may have an oracle that would allow us to determine correctness. The impossibility of exhaustive testing still lingers however. Practically, this means we make judicious choices for our testing, and we fall back on the sufficiency of trust.

5. Definition of Silent Fault

Trust means we use "judgment" to decide the acceptability of output from an example of software. Judgment unfortunately can miss evidence of a serious software fault due to insufficient data and misinterpretation. For scientists this is exacerbated by the accumulation of approximations and scientific error as the code is realized from the different models.

Under many different conditions, a code fault may provide no obvious signal that something is wrong and slip past the scientist's judgment. This is what we term a "silent fault". More fully, we define silent faults as those that cause a change in output but do not cause any of (i) an error message, (ii) a system crash, (iii) wildly improbable results, or, (iv) noticeably longer execution time (in the extreme, an endless loop).

Silent faults are what Hatton and Roberts [5] found when they compared the output of nine seismic data processing packages. Code faults such as one-off errors caused significant loss of accuracy. These faults went unnoticed by the geoscientists because the output of the code fell within the limits of plausible values. Yet, as Hatton points out in [4], this output data is used to site oil wells and the loss of accuracy effectively randomized a multi-million dollar decision making process.

It is these types of faults we are interested in exposing. For this research, we are using a technique called mutation testing.

6. Analysis of Software Behaviour Using Mutation Testing

Mutation testing is a well defined technique [eg. 1, 2, 8] used to assess the adequacy of *a set of tests* for a particular piece of software, P_0.

The technique involves generating a number of new source codes, called mutants, by making one small change to the source code of P_0 for each mutant, P_m. The types of small changes are based on a set of mutation operators that research has demonstrated [eg. 2] to be representative of code faults in general. Each mutant then represents a single code fault that could occur in the original code P_0.

The adequacy of a set of tests for P_0 is established by running the set of tests on all the mutants that have been generated. A mutant is said to be "killed" by any one of the tests in the set if the output from the mutant P_m differs from the output of the original program P_0. This is a simple binary decision. Adequacy of the set of tests is judged by the ability of the tests to kill the mutants.

For scientific software with floating point output, and with the accumulation of the many errors and approximations as shown in Figure 1, a binary decision on equality is not useful. Instead, we compute a sensitivity measure S between the output of the mutated code and the output of the non-mutated code for a particular test tx:

$$S(P_0, P_m, tx) = | P_m (tx) - P_0 (tx)| / |P_0 (tx)|$$

The sensitivity measure S gives a relative difference between the output for P_m and P_0 when executing the test tx. We do not use any tests where $|P_0 (tx)| = 0$.

For our research, we generated a set of mutants for various examples of scientific software. We then ran a series of tests to examine the ability of the tests to expose the mutants. In other words, we wanted to determine which tests caused the output of the mutant P_m to be most different from the output of P_0, and hence most visible. In other words, we are interested in tests with maximum S for each mutant P_m.

At the moment, we are using examples of code drawn from a library of MATLAB functions. The codes are all relatively small, but are intended to be part of a larger application. We feel the codes are similar to small numerical code units typically used in larger computational codes.

For each code example, we generated a set of mutants and two sets of tests. The first set of tests was generated randomly with values chosen from the range of reasonable input values. The second set of tests was specifically designed to exercise the valid limits of the functions.

The results we are observing so far suggest that a relatively small set of tests that "push" on the limits of the coded computation can reveal a large proportion of the code faults represented by the mutants.

Figures 4 and 5 illustrate our observations with an example of code that performs an integration using Simpson's Rule. The x-axis gives the real number range of values for each bar in the graph. The height of the bar gives the number of mutants whose maximum S fall in the associated real number range. The right most bar is marked "err". This is the count of mutants that failed with an error condition for at least one test tx.

We'll call the source code implementing Simpson's Rule, P_s. For P_s, 167 mutants were generated using MATLAB variants of the mutation operators described in [1].

The first set of tests we created consisted of 200 random tests. For each test and for each mutant, we calculated $S(P_s,P_m,tx)$. Since we are interested in tests that reveal a mutant, we recorded the maximum S for each mutant. For the set of random tests, Figure 3 shows the counts of the number of mutants whose max(S) falls in the given ranges.

If $S(P_s,P_m,tx) = 0$, then P_s and P_m are considered equivalent for tx. There were 11 mutants where S was zero for all tests in the set. For the purposes of our discussions, the range of $0 < S < 1$ was chosen as representing silent faults. Faults whose output falls in this range under particular test conditions might be difficult for a scientist to detect. There were 50 mutants whose maximum S was in this range. There were 47 mutants whose maximum S was above one. We assumed this difference could be large enough to make the fault visible. Another 59 mutants caused an error condition, also very visible. If a test causes the sensitivity measure for a mutant to fall either into the range $S>1$ or into an error condition, then we say the test strongly reveals the mutant. In summary, our first set of tests strongly revealed 106 of the 156 non-equivalent mutants, or 68% of the mutants.

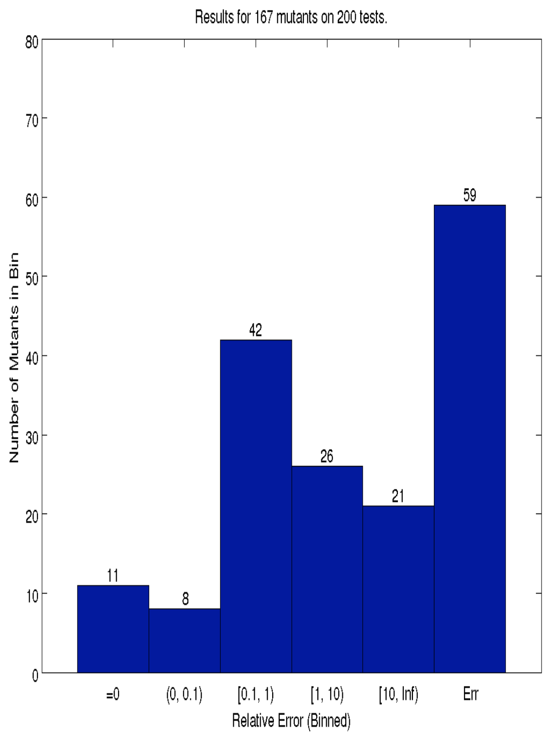

Figure 3: Counts of mutants P_m with given maximum *S* for a set of 200 random tests.

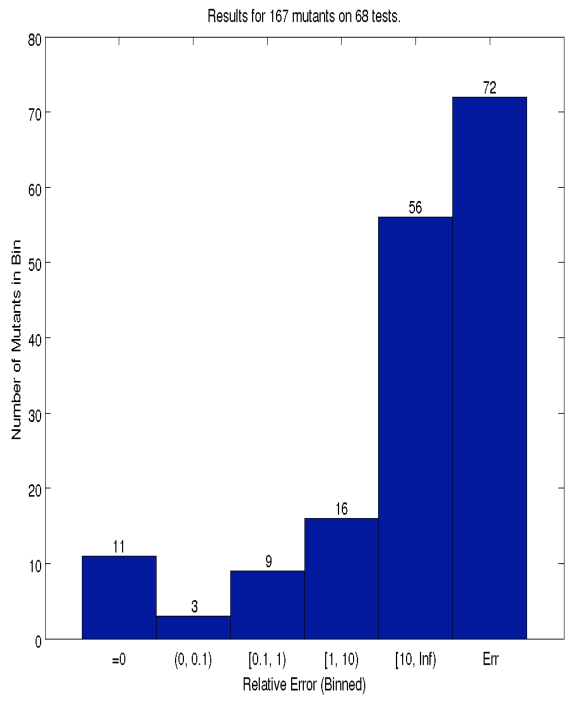

Figure 4: Counts of mutants P_m with given maximum *S* for a set of 68 well-designed tests.

A second set of tests were designed to exercise the function by pushing on its boundaries. We designed tests to exercise the function in ways that would be immediately recognizable to scientists. For example, the number of panels in the integration were chosen as small (1 and 2) and large (1000); sections were chosen to include intercepts, symmetries and non-symmetries. By taking valid combinations of these inputs, we generated 68 tests. Figure 4 shows the counts of mutants with max(*S*) in the given ranges for this set of tests.

We again observed 11 mutants in the range *S*=0 (in other words, equivalent to P_s). However, the number of mutants that either caused an error or whose value for max(S) >1 is 144. In other words, 144 of the 156 non-equivalent mutants, or 92%, were revealed by this smaller set of tests.

On closer examination, one test alone (test #4) strongly revealed 140 of the mutants. The addition of one other test (test #1) resulted in all 144 mutants being revealed. The success of test#4 may have resulted from its un-mutated output being very close to zero; it was easy to perturb the output by code changes. Test #1 used the minimum value of one for the number of panels in the integration and revealed errors that tried to push the number of panels to illegal values.

For this particular example, a set of two well chosen tests revealed 92% of code faults.

Our examination of other MATLAB examples of scientific software has given similar results. In all cases the number of mutants generated was comparable to that for the Simpson's Rule function. Overall, we found for the examples we considered, that 2 to 5 well designed tests strongly revealed a large percentage of the mutants. The percentages ranged from 76 to 100 percent. The well designed tests were based on ideas that would be immediately cognizant to scientists.

7. Conclusions

The difficulty of testing scientific software arises from many factors including the lack of test oracles and the difficulty of judging how much testing is enough. A testing goal of correctness is impractical and should be replaced by a goal of trustworthiness. Scientists are already comfortable with testing their software to increase trust in their science models and their solution techniques. Scientists also need to test the software to address code faults, with a cycle of testing that we call code scrutinization. The practical impossibility of finding all code faults is well understood, which leaves the question of how much is enough? This is of particular concern to scientists where their time needs to be spent doing science, not testing software. To make a particular testing approach attractive to scientists, it must be time-efficient and results-effective. Our research is showing that it may be possible to characterize and define such an approach. At this moment in our research, we are making judicious choices for our testing with the help of 20-20 hindsight. We need to move on to identifying the commonality in these choices and provide scientists with guidance on how to secure trust in their software. Our aim is to identify a time-efficient and results-effective testing approach for scientific software.

8. Acknowledgements

This work is funded by NSERC (Natural Sciences and Engineering Research Council of Canada) and ARP (Canadian Department of National Defense Academic Research Program.)

9. References

[1] James H. Andrews, Lionel C. Briand, Yvan Labiche, Akbar Siami Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria", IEEE Transactions on Software Engineering, Vol. 32, No. 8, August 2006, pp. 608-624

[2] James H. Andrews, Lionel C. Briand, Yvan Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?", Proceedings 27th International Conference on Software Engineering ICSE '05, 2005, pp. 402-411

[3] CSA N286.7-99, Quality Assurance of Analytical, Scientific, and Design Computer Programs for Nuclear Power Plants, Canadian Standards Association, March 1999

[4] Les Hatton, "The chimera of software quality", Computer, 40(8):104, 2007, pp.102-103

[5] Les Hatton, Andy Roberts, "How accurate is scientific software?", IEEE Transactions on Software Engineering, 20:10, 1994, pp. 786-797

[6] IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, 1999

[7] Patrick Knupp, Kambiz Salari, Verification of Computer Codes in Computational Science and Engineering, Chapman & Hall/CRC, USA, 2003

[8] A. Jefferson Offut, Roland H. Untch, Mutation 2000: Uniting the orthogonal. Mutation testing for the new century, Kluwer Academic Publishers, Norwell, MA, USA, 2001, pp. 34-44

[9] Douglass E. Post, Lawrence G. Votta, "Computational Science Demands a New Paradigm", Physics Today, January 2005, pp. 35-41

[10] Patrick J. Roache, "Building PDE Codes to be Verifiable and Validatable", Computing in Science and Engineering, September/October 2004, pp. 30-38

[11] Patrick J. Roache, Verification and Validation in Computational Science and Engineering, Hermosa publishing, USE, 1998

[12] Rebecca Sanders, Diane Kelly, "Scientific Software: Where's the Risk and how do Scientists Deal with it?", IEEE Software, July/August 2008, pp. 21-28

[13] D.E. Stevenson, "A Critical Look at Quality in Large-Scale Simulations", Computing in Science and Engineering, May-June 1999, pp. 53-63

[14] Elaine Weyuker, "On Testing Non-testable Programs", The Computer Journal, Vol. 25, No. 4, 1982, pp. 465-470