Assessing the Quality of Scientific Software

Diane Kelly
Royal Military College of Canada
Kingston, ON Canada
Kelly-d@rmc.ca

Rebecca Sanders
Queen's University
Kingston, ON Canada
sanders@cs.queensu.ca

Abstract

This position paper comments on the current quality assessment practices amongst scientists writing computational software and suggests directions in research that would both increase our understanding of the difficulties involved in producing high quality computational software and provide useful advice and methodologies for the scientists.

Our views are based on a series of interviews carried out at RMC and Queen's, our personal experiences in writing computational software for industry, other research at RMC and Queen's, and software related literature written by computational scientists.

Keywords

Scientific software, software verification and validation, software testing, software inspection, software development process

1. Introduction

Testing remains the most commonly used quality assessment activity for software of any kind. This is despite wide recognition that inspection is the most effective activity (eg.[5]) for quality assessment and the fact that formal methods are mathematically rigorous (eg. [2]). For computational (or scientific) software, a type of validation testing is used almost exclusively by practicing scientists. This established practice has been recognized as a marginal assessment at best (eg. [6]). The question remains, what will work better? By "work", we mean an activity that has the following features: an activity that performs as expected, that is amenable to the intended users, that meshes with established practices, that visibly contributes to the real work the users want to do, and the returns on invested time and effort in the activity are sufficient to encourage its use.

To possibly answer these questions, we need first to understand the environment where we intend to introduce these new activities. Then we need to develop and validate our activities in that environment.

One of us worked for over twenty years as a scientific software developer. Both have made personal observations in various areas of scientific software development. Together, we carried out a series of interviews of scientists who write or use scientific software. The conclusion is that there is substantial potential for research and collaboration between scientists writing software and the software engineering community.

First International Workshop on Software Engineering for Computational Science and Engineering, Leipzig Germany, May 2008.

2. A Brief Characterization of Scientists Developing Software

In our series of interviews, we talked to academic scientists from ten different disciplines about their software development practices. For all the scientists we interviewed, the development or use of software was a means to another end, that of pursuing research in their own scientific field. The software background of the scientists in our study, whether formal or not, varied from none to significant. Similarly, the comfort level of the scientists dealing with software varied significantly. The size of the software these scientists dealt with varied from less than 1000 lines of code to over 100 000.

Despite this variation in the characteristics of our interviewees, testing the computational portion of their software involved limited testing against values gathered from the world related to their field of science or engineering. These values could be analytical, experimental or measurements of natural events. They test to assess their models, not the software. Only if the software comprised a significant user interface did some scientists focus on other testing goals such as usability.

For the computational portion of their software, the quality attribute of almost exclusive concern was computational correctness. In other words, did the calculations expressed in the software code give answers that agreed sufficiently with expected answers? In industry, as in our interviews with academic scientists, correctness was the prime quality of interest. If the software does not return correct answers, then it is useless. Being on-time, under budget, maintainable, or highly usable all take a backseat to correctness.

3. What do Verification and Validation Mean?

One of the many problems facing scientists writing their own software and attempting to find effective advice for improving its quality is confusion in terminology. Particular examples are the terms verification and validation.

The software engineering community has not itself agreed on definitions for these terms. One set of definitions are stated succinctly as the questions, "Are we building the product right?" and "Are we building the right product?" The more formalized definitions are found in ISO standards and their derivatives. These definitions are linked to measuring whether software production activities have produced the desired product. From what we have observed, these definitions and their associated processes are orthogonal to how scientists work.

Amongst the scientists, consistency in the definitions for verification and validation fares no better. Roache [12] transforms

the succinct software definitions into "Verification – solving the equations right" and "Validation – solving the right equations". A Canadian standard, CSA N286.7 [3], regulating the production of analysis (computational) software for the nuclear industry, uses a variant of the ISO definition for verification, implying a waterfall type development process:

"Verification - the process of determining whether or not the products of a given phase of the computer program development cycle fulfill the requirements established during the previous phase."

Along with this, N286.7 uses a domain model-focused definition for validation:

"Validation – comparison of the results of the computer program with measurements or experimental data or known analytical or numerical solutions, so that the accuracy or uncertainty of a particular application can be determined."

Yet another set of views on verification and validation comes from Stevenson [14]:

"Validation answers the question 'How well does the model reflect objective observations?' " and "The verification problem is one of formal systems and therefore applies only to the theoretical system."

Such definitions indicate that the layers of complexity in computational software may need to be assessed individually as well as a whole. We suggest that the word *assessment* be used as a general term instead of verification, validation, and that the differentiation of types of assessment be done by specifying the quality goal of the assessment.

4. Software Inspection

Software assessment approaches defined by the software engineering community can be roughly classified as testing (dynamic assessment), inspection (static assessment), and formal methods (mathematically based assessment).

One of the authors worked in industry for over twenty years in various capacities related to computational software development. One experience from industry that was not repeated in our academic interviews was that of software inspections.

None of our interviewees did any form of formalized software inspections. One scientist described requiring his students to bring source code to monthly meetings so he could look at the code. The activity was not formalized beyond that.

From our industrial experience, inspections are not a regular part of development activities for scientific software. A novel inspection technique (task-directed inspections) and process was introduced by one of us into practice in an engineering company and its effectiveness examined in a series of empirical experiments [8] [9]. The use of the novel inspection activity was considered successful but its limitations were also clear.

Inspections can be used to address different quality factors that play a role in correctness in computational software. Consistency is one of those factors. Software code can be inspected for consistency with documents that describe the theoretical models implemented in the code, the users' manual describing input and output data for the code, and data dictionaries created for the code, amongst other things. The code can also be inspected for self-consistency. Numerical soundness is another factor. Implementation of computations is affected by the move from the continuous realm to the finite realm. Equations rendered into code can be inspected for pitfalls due to round-off errors and other numerical traps. The code can also be examined for integrity of its use of coding constructs. Does the actual execution of the coded statement match the intended semantics? For example, does the initialization as coded produce what was intended, would the condition statement be evaluated in the manner expected?

To be effective, inspections need to be carried out by people who have the knowledge to understand the code they are inspecting. For scientific software, that knowledge includes knowledge of the scientific domain. This limits the availability of effective inspectors. We found this to be true from our industrial experience. To effectively implement our inspection technique, we matched code modules to people who had the necessary background to understand the intent of the code, for example, heat transfer modules to people with heat transfer knowledge, pump simulation modules to people with knowledge of pump models, etc. This meant that we had one inspector per module and a full Fagan-style inspection [4] was out of the question. We provided each inspector with a well-defined, guided reading technique that helped overcome this limitation [8].

Inspections provide a means of finding certain types of defects in source code, and should be added to the scientist's quality assessment toolkit. Even something as simple as desk-checking your own code should be encouraged. In another industrial exercise, two developers acted as inspection buddies (as opposed to testing buddies). One developer designed or wrote code while the other inspected/read the code and design and turned back suggested changes. A thousand lines of highly complex code following cutting edge theory was designed, implemented, tested, and made to work in a week.

5. Formal Methods

We have not ourselves used formal methods to assess computational software but have observed formal methods in use for safety-critical controller type software. Work is being done at McMaster University to extend David Parnas's tabular notation and four-variable models for use with scientific software [11] [13]. Generally, formal methods require some type of detailed requirements specification. The majority of scientific software is developed without the use of detailed requirements specifications. To be affordable in industry, the application of a formal method needs automated tool support. At this point, it is not clear what parts of computational software nor what quality factors would benefit from the very fine-grained examination formal methods provides.

6. Established Testing Techniques in Software Engineering

In research at RMC [7], a software engineering graduate student devised and carried out a set of tests on an example of software written by scientists in nuclear engineering. One of the purposes of the research was to provide software engineering expertise to the science group in order to improve the quality of their software. A variety of testing approaches drawn from established testing literature were applied to the software. Obvious successes included the discovery of a number of hidden defects, the creation of a regression test suite, the organization and cataloguing of test data along with ranges of acceptable results, the discovery of the impact of different compiler options on numerical results, better coverage of the source code with test runs, and the organization of

version control on all artifacts related to the software. These are elements that seem to be typically missing from the toolset of the scientist/developer (eg. [1] [15]). More interestingly, it was apparent that testing techniques designed around such concepts as equivalence classes or code coverage, while not perfect, are even less reliable when applied to computational software. Computational software represents continuous models using finite and discontinuous resources. One set of test data that succeeds does not guarantee the success of test data anywhere in its neighbourhood, even if it follows the same source code path or belongs to the same equivalence class.

7. Other Confounding Factors for Quality Assessment

A number of factors, both technical and human, contribute to the difficulties of assessing scientific software.

7.1 Dynamic interactions

Output from computational software is often extensive and complex. Correctness of that output (or some selected subset of that output) is dependent on not only a long list of factors, but the complex dynamic interaction of those factors. Arnold and Dongarra [1] write, "Veterans of iterative methods agree that finding the right combination of solver, pre-conditioner, scaling and re-ordering is an art form developed only with experimentation within the application area." Add to that differences made by computer word-size, computer architecture, compiler options, computing language libraries, and data. That interaction is not evident until the code is executed. Scientists are aware of the dynamic interactions and testing remains their main quality assessment tool.

7.2 Limited oracle data

Scientists test the computational component of their software to show that their science models are correct, not to find problems with the software. To show that the science models are correct, scientists gather information from the science domain that they are modeling. They use this information as comparisons, or oracles, to check their models as implemented in software. Sometimes the amount of information available for a set of oracles is limited. As one of our interviewees put it, some software deals with situations that "you don't want to see happen".

7.3 The software is invisible

Scientists see the software code as an inseparable entity from their models. They assess the models. They normally do not assess the software as a separate thing that needs attention. The software essentially goes invisible. When discussing the possible inclusion of a software engineer in his group, one of our interviewees declared that the software engineer had to "keep his hands off my model".

7.4 The singular importance of correctness

Stevenson [14] quotes Richard Hamming in his article on quality computational software: "The purpose of computing is insight, not numbers." Computational software is the means of providing the data for that insight. However, that insight is gained from having correct data, or output, from the software. One of our academic interviewees commented that the software had better not lie to him. Much preferred is a complete crash of the system than an insidious error that goes undetected and provides data that corrupts the insight.

7.5 Scientists want to do science

Software activities must not interfere with progress in developing the scientific models. Scientists do not want to be spending time on software issues that do not directly and visibly contribute to their doing science. Scientists are primarily interested in doing science, not software.

8. Where Do We Go From Here?

There are a number of issues that need to be considered. When we use the word "works", we refer to our definition given above.

(i) How much of the problem is education? Are scientists who don't desk-check their code or keep regression test suites unaware of these activities? If education is an issue, what do we, as software engineers, teach scientists? One of us has taught software engineering courses to non-software engineers. It is not at all clear what the curriculum should include.

(ii) Scientists talk of applying the "scientific method" to their code products, particularly when describing validation testing. Their validation testing appears to be weak, yet there are certainly examples of high quality scientific software. How can we leverage the ideas behind the scientific method and apply them consistently to scientific software production?

(iii) Correctness is the highest priority quality attribute for all scientists. At this point, we don't have a full list of factors that contribute to correctness of scientific software, particularly factors in areas that a software engineer could address.

(iv) The disconnect between quality factors and the activities to achieve them is well known [10]. What activities can contribute to factors of importance to correctness? How effective are these activities? Would these activities fit into the established way of working for scientists?

(v) We need to develop inspection/reading techniques and processes amenable for use by scientists with computational software. Their application needs to consider the limited resources, the nature of the software, and the interests of the scientists using them.

(vi) Are there appropriate uses of formal methods with computational software? Where could they be used? What tailoring is necessary to make them work?

(vii) The effectiveness and limitations of existing testing strategies and techniques need to be established when the techniques and strategies are applied to computational software.

(viii) An extensive study of testing approaches currently used by scientists is needed to identify approaches that work and the conditions under which they do work. We need to identify what the scientists are

missing in their testing practices and determine if these "holes" matter.

Combining all the above, we need to identify, validate, and disseminate a toolkit of assessment activities specific for the quality factors important to the scientist and computational software. This is an enormous research activity. Where should we start?

9. ACKNOWLEDGMENTS

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

10. REFERENCES

[1] Dorian C. Arnold, Jack J. Dongarra, "Developing an Architecture to Support the Implementation and Development of Scientific Computing Applications", The Architecture of Scientific Software, Kluwer Academic Publishers, 2000

[2] Daniel M. Berry, "Formal Methods: The very idea, some thoughts about why they work when they work", http://se.uwaterloo.ca/~dberry/FTP_SITE/reprints.journals.conferences/formal.methods.very.idea.extabst.pdf

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

[4] M.E, Fagan, "Design and Code Inspections to Reduce Errors in Program Development", IBM Systems Journal, Vol. 15, No.3, 1976, p.182-211

[5]Robert Glass, "Inspections - Some Surprising Findings", Communications of the ACM, April 1999,Vol.42, No.4, pp. 17-19

[6] Les Hatton, Andy Roberts, "How Accurate is Scientific Software?", IEEE Transactions on Software Engineering, Vol. 20, N0. 10, October, 1994, pp. 785-797

[7] Diane Kelly, Nancy Cote, Terry Shepard, "Software Engineers and Nuclear Engineers: Teaming up to do Testing", proceedings Canadian Nuclear Society Conference, St John New Brunswick, June 2007

[8] Diane Kelly, Terry Shepard, "Task-Directed Inspection", Journal of Systems and Software (JSS), Vol. 73/2, October 2004, pp.361-368

[9] Diane Kelly and Terry Shepard, "Task-Directed Software Inspection Technique: An Experiment and Case Study", Proceedings IBM CASCON 2000, Toronto, November 2000

[10] Barbara Kitchenam, Sheri Lawrence Pfleeger; "The Elusive Target", IEEE Software Jan.1996, pp.12-20

[11] K. Kreyman and D.L. Parnas, "On Documenting the Requirements for Computer Programs Based on Models of Physical Phenomena", *SQRL Report No. 1*, Software Quality Research Laboratory, Dept. of Computing and Software, McMaster University, January 2002, 14 pgs.

[12] Patrick J. Roache, Verification and Validation in Computational Science and Engineering, Hermosa Publishers, New Mexico, USA, 1998

[13] S. Smith, "Systematic Development of Requirements Documentation for General Purpose Scientific Computing Software", *Proceedings of the 14th IEEE International Requirements Engineering Conference,* May 2006, pp. 205-215.

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

[15] G. Wilson, "Where's the Real Bottleneck in Scientific Computing?", *American Scientist, Vol. 94,* January 2006, pp. 5.