Studying Silent Faults in Scientific Software using Program Mutation

Daniel Hook
Queen's University
`hook@cs.queensu.ca`

Diane Kelly
The Royal Military College of Canada
`kelly-d@rmc.ca`

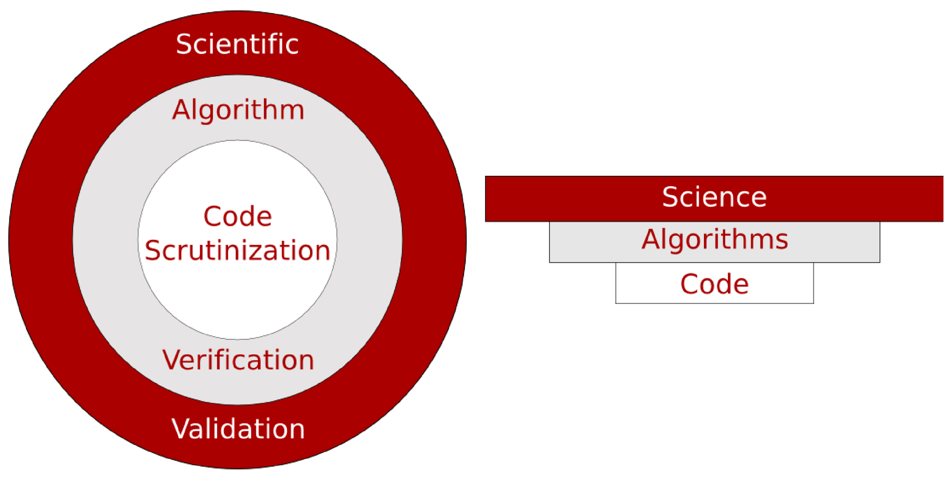
Abstract

Highly accurate scientific software requires valid scientific models, accurate numerical methods, and highly correct code. Software engineers specialize in testing code, but the lack of test oracles and the existence of "silent faults" makes it very difficult to test the correctness of scientific code. We suggest that code mutation can be used to study code faults in scientific software in the hope that software engineers can use the derived knowledge to make valuable contributions to the quality of scientific software and the associated research. This poster highlights challenges of scientific software testing before briefly describing the mutation testing process and providing sample results from mutation sensitivity tests.

1. Introduction

SCIENTIFIC software must be sufficiently accurate; all other quality attributes of scientific programs are of secondary importance. If the error (i.e., the distance from the "true" answer) in the output of a scientific program is not bound within a specified tolerance then the program cannot be trusted. However, due to the lack of a testing oracle, it is often very difficult to determine if a scientific program's outputs fall within an acceptable range when the software is under test.

Therefore, instead of evaluating and insuring accuracy by using traditional software testing techniques, scientific software must be carefully scrutinized to determine if confidence in the program is warranted. This scrutinization can be understood to take place from three distinct *views*. First, the *scientific view* is used to validate the theories and assumptions that are used to model real world phenomena in the computational domain. Second, the *numerical analysis view* is used to verify that the algorithms used in the software are suitable for working with the scientific models. Finally, the *code view* is used to scrutinize the source code to check that it is (reasonably) correct. It is by working in the code view that software engineers can help improve the quality of scientific software projects. However, before software engineers can involve themselves in these activities it is important that they understand the novel problems and concerns that arise when testing scientific software.



(a) (b)

Figure 1: *(a) shows the scope of the three quality "views" while (b) shows the dependency that exists among them. To illustrate how this is meant to be understood: code scrutinization tells the developer nothing about the scientific validity of the program (because scientific validity is not within the scope of code scrutinization), but accurate computational science is dependent on highly correct code (because code underlies science in the hierarchy).*

Silent faults are of particular concern when testing scientific software because they are difficult to detect and can impact scientific software quality in significant and unappreciated ways. Silent faults can cause *silent failures*, i.e., they can cause a program to behave in an undesirable way that is not accompanied by an obvious symptom such as an error message or a crash. Silent failures can seriously degrade the accuracy (i.e., increase the error) of a program's output while remaining unnoticed by a tester because the desired output value is unknown. Hatton does not use the term "silent fault" in [1], but his work demonstrates that accuracy degradation due to unnoticed code faults is a severe problem in scientific software.

What is needed is a way to evaluate the ease of use and effectiveness of various testing techniques—both established and novel—when they are applied to the task of detecting silent faults. We feel that mutation analysis is well-suited to this research for a number of reasons:

- It is a well established and studied process that has existed for over 30 years (cf. [2]).
- The oracle problem is avoided because the unmutated program is (temporarily) assumed to be correct.
- It can be used to demonstrate the insufficiency of the test suites that scientists use to test their software.
- As a consequence of the *coupling effect*, the simple syntactic errors introduced during mutation can be used to study detection methods for more complex faults (cf. [2]).
- It a technique that is largely automated and leaves an easy to record "paper trail"; this helps scientists improve the reproducibility of their test practices.

2. Mutation Testing

The following terminology and notation is used to discuss mutation testing:

- **Mutation**: a syntactic change to a program statement.
- **Mutation Operator** (ϕ): a rule that is applied to a program statement to generate mutations. A set of mutation operators is denoted by Φ.
- **Mutation Target** (P_0): a program that is to be mutated.
- **Mutant** (P_m): a program which is syntactically identical to P_0 except that one of its statements contains a mutation. When Φ is applied to P_0 the set of generated mutants $\Phi(P_0)$ is denoted by P_M.
- **Test** ($t_X(P_0, P_m)$): a function that uses some specified valid input X for P_0 to compare $P_m(X)$ with $P_0(X)$ and output a corresponding `pass` or `fail`. A set of tests selected by a tester is denoted by T.
- **Killed**: if $t_X(P_0, P_m)$ = `fail` then P_m is said to have been killed by t_X.
- **Survivors** (S_T): the set of mutants not killed by some T, i.e., $S_T = \{P_m \in P_M$ such that $t_X(P_0, P_m) = \texttt{pass}$ $\forall t_X \in T\}$.
- **Equivalent Mutant**: if $P_m(X) = P_0(X)\ \forall X$ then P_m is said to be equivalent to P_0.

A mutation test consists of the following steps:

1. Φ is applied to P_0 to generate P_M.
2. Every $P_m \in P_M$ is evaluated using every $t_X \in T$ to determine S_T.
3. Equivalent mutants are removed from S_T.
4. If $|S_T| > 0$ then new t_X are added to T in an attempt to kill all $P_m \in S_T$.

Geist *et al.* claim that, "If the software contains a fault, it is likely that there is a mutant that can only be killed by a test case that also reveals the fault."[3] If this statement holds—and evidence indicates that it does when testing non-scientific software—then mutation testing will be effective at finding faults.

However, we are hesitant to assume that established mutation testing techniques will be effective at testing scientific software. Established practices often use strict equality, but, in a scientific context, strict equality is often far too strict. For example, if a scientific program P_0 must be accurate within 10^w and $P_0(X) - P_m(X) = 10^{w-1}$ then should P_m be considered equivalent or not? To make matters even worse, it is rare that the required accuracy of a scientific program can be specified precisely. There are many cases when the only available judge of accuracy is the "common sense" of a scientist, i.e., the output is judged by whether or not it "looks about right."

Therefore, we would suggest that code mutation should not be used as a fault detection tool (at least not initially), but rather as tool that allows developers to assess a program's sensitivity to certain classes of faults.

3. Sensitivity Testing

In order to better understand the nature of faults and failures in scientific software, we suggest a new approach to mutation analysis: instead of using the mutants to assess the adequacy of a test set we have started using the mutants to assess the fault sensitivity of programs. In order to do this we have modified the behaviour of the test functions. Traditional mutation tests use the following mapping:

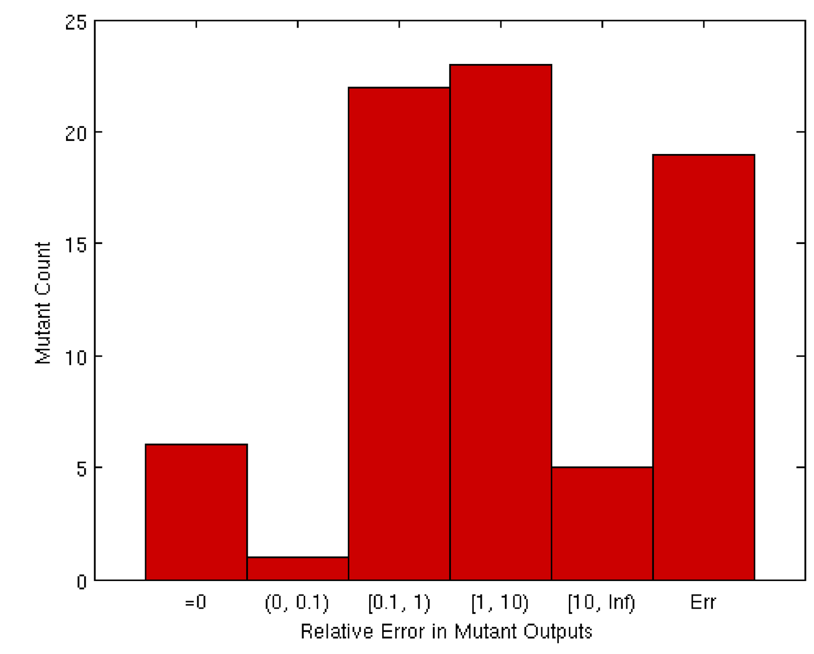
$$t_X(P_0, P_m) \rightarrow \{\texttt{pass}, \texttt{fail}\}$$

We suggest that t_X be used to measure mutation error instead, e.g.:

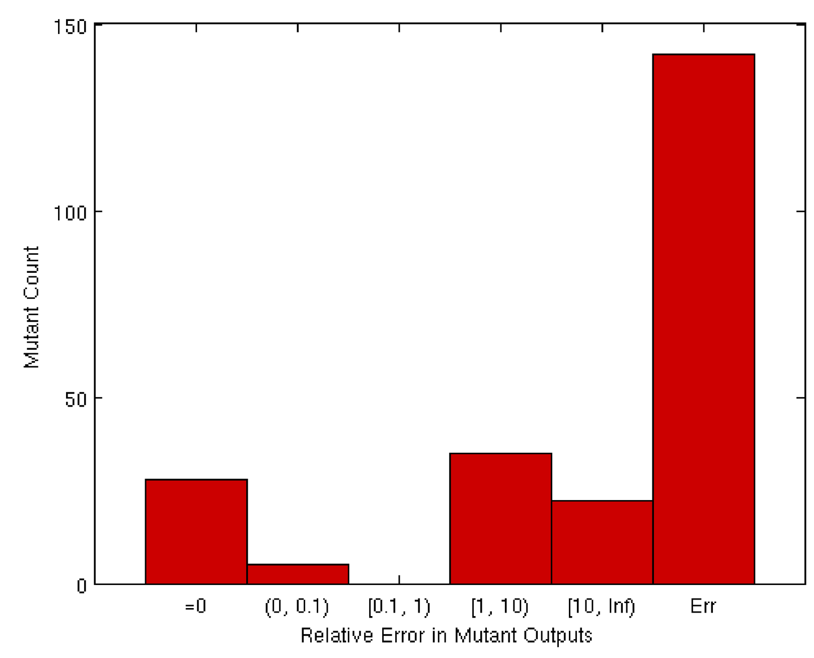
$$t_X(P_0, P_m) = \frac{|P_m(X) - P_0(X)|}{|P_0(X)|}$$

where P_0 and P_m output numerical results (if this is not the case then a different measure can be applied).

Figure 2 shows some sample histograms that can be produced using this error analysis.



(a) **Sensitivity Results for `myclip.m`**



(b) **Sensitivity Results for `GEPivShow.m`**

Figure 2: *(a) gives sensitivity results for a matrix conditioning function; (b) gives sensitivity results for a function that solves a system of equations using Gaussian elimination with pivoting. It is interesting to note the different features of the histograms: using randomly selected inputs, `myclip` exhibits a much high proportion of silent errors than `GEPivShow`.*

By measuring the error in the outputs of mutants it is possible to characterize the sensitivity of the mutation target to code faults. By comparing the sensitivity results with the requirements of the software it may help developers and testers to better target their quality assurance activities, and we believe it will help illustrate the unique problems that are encountered when testing scientific software—problems that are in urgent need of study.

4. Current and Future Work

In order to assess the effectiveness of mutation testing as a scientific software testing technique we partnered with a space scientist who is developing satellite tracking functions using MATLAB. In order to test this MATLAB code, Daniel Hook constructed a mutation tester for MATLAB called MATmute (available at `matmute.sourceforge.net`). Preliminary results indicate that MATmute is helping the space scientist find omissions in his test suites, and our sensitivity analyses have demonstrated that silent errors need to be given more attention.

As we apply the MATmute systems we are finding that our work is opening up many potential avenues of exploration. Scientists and software engineers have drifted apart, we feel its time to start bringing them back together.

References

[1] Les Hatton and Andy Roberts. How accurate is scientific software? *IEEE Transactions on Software Engineering*, 20:10, pp. 786–797, 1994.

[2] A. Jefferson Offutt and Roland H. Untch. Mutation 2000: uniting the orthogonal. *Mutation testing for the new century*, Kluwer Academic Publishers, Norwell, MA, USA, pp. 34–44, 2001.

[3] Robert Geist, A. Jefferson Offutt, and Frederick C. Harris, Jr. Estimation and enhancement of real-time software reliability through mutation analysis. *IEEE Transactions on Computers*, 41:5, pp. 550–558, 1992.

CASCON 2008, IBM Centers for Advanced Studies Conference, October 27–30 2008, Richmond Hill, ON, Canada

Work funded by NSERC and ARP.