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, correct 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 fault (1) is conducted to determine if a difference 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 code so that it is (as reliably) 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.

1.1. Scientific Software

The oracle problem is avoided because the unmutated program is (temporarily) assumed to be correct.

It can be used as a measure of 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]).

A technique that is largely automated and leaves an easy task to record "paper trail"; this helps scientists improve the reproductibility 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 (\(\Phi\)): a rule that is applied to a program statement to generate mutants. A set of mutation operators is denoted by \(\Phi\).

Mutation Target (\(t\)): a program that is to be mutated.

Mutant (\(P_m\)): a program which is syntactically identical to \(P\) except that one of its statements contains a mutation.

When \(t\) is applied to \(P\) the set of generated mutants \(\Phi(P)\) is denoted by \(P_m\).

Test (\(T(X, P_m)\)): a function that uses some specified valid input \(X\) to compare \(P(X)\) with \(P_m(X)\) and output a corresponding pass or fail. A set of tests selected by a tester is denoted by \(T\).

Killed (\(\Psi(P_m, P)\) = fail): then \(P_m\) is said to have been killed by \(T\).

Survivors (\(S\)): the set of mutants not killed by some \(T\), i.e., \(S = \{P_m \in P_m | \Psi(T(X, P_m)) = \text{pass} \forall X \in T\}\).

Equivalent Mutant: if \(P(X) = P_m(X) \forall X \in T\) then \(P_m\) is said to be equivalent to \(P\).

A mutation test consists of the following steps:

1. Every \(P_m \in P_m\) is evaluated using every \(T\) in \(T\) to determine \(S\).

2. Equivalent mutants are removed from \(S\).

3. If \(|S| > 0\) then new \(T\) are added to \(T\) in an attempt to kill all \(P_m \in S\).

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 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 \(P\) must be accurate within \(10^{-6}\) and \(P(X) = P_m(X) \cdot 10^{-6}\) 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 for scientists 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:

\[ \Psi(P_m, P) = \text{fail} \quad \text{if} \quad P_m(P) \neq P(P) \]

We suggest that \(\Psi\) be used to measure mutation error instead, e.g.:

\[ \Psi(P_m, P) = \frac{|P_m(P) - P(P)|}{P(P)} \]

where \(P\) 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.

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


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

Work funded by NSERC and ARP.