Effective Research Software Verification

David Benn (CSIRO), Waite Campus

Research Software Engineers (RSEs) often work alone or in small teams, potentially on multiple concurrent projects so may be time poor. Both verification (building it correctly) and validation (building the right thing) are important. Limiting the focus to verification, what methods make sense for the variety of application types in a research context?

Introduction

As with software development in general, writing code for research applications is often only half the story and the effort; less in some cases. Along with discovering what is to be built and documentation, there is the need for some sort of *verification.*

Testing involves *verification* of the *dynamic behaviour* of code in which inputs are supplied to running software and outputs checked against some expected result, via a collection of test cases.

Static verification or analysis involves inspection of code by a human (e.g. via git pull requests) or a program to check that it adheres to coding conventions, does not exhibit certain anti-patterns [1], measures some metric of interest, or attempts to prove that some important properties always hold true.

Additionally, issues such as numerical tolerance and reproducibility loom large in research software and scientific computing.

The emphasis here is on answering the question: what approaches are most effective for a given research software application type?

The Usual Suspects

On the *dynamic verification front*, unit tests, integration tests, system tests, and acceptance tests are generally accepted divisions. Unit tests check that a function, class or module yields the expected outputs given known inputs. *Integration tests* assume that units have been verified and check the expected behaviour when units interact. *System testing* treats the whole system of interacting, integrated components as a black box to be verified against a specification or set of requirements of some kind. *Acceptance testing* involves users in determining whether software is fit for purpose in their context and with reference to a specification. Other more specialised system testing categories are performance testing, accessibility testing, security testing.

On the *static verification* front, tools exist for numerous languages that carry out static analysis of source code to find problems in code without having to run it. Integrated development environments (e.g. Eclipse, Visual Code) often also include static analysis features, flagging problems as code is entered.

As a code base grows, static analysis is arguably more important for dynamically typed languages such as Python or R whose type systems don't rule out certain classes of error before run-time such as whether an integer is being passed to a function when a string is required.

```
69      logger.debug("test_rpn_add_arbitrary: ", arbitrary_length_summation)
⊗ rpn_hyp.py 2 of 5 problems
Too many arguments for logging format string pylint(logging-too-many-args)
70      assert rpn(arbitrary_length_summation) == pytest.approx(sum(ns), rel=1e-2)
```

Properties!

More advanced forms of static verification come from the field of *formal methods* in which important *properties* of code are formally *proved*, e.g. that deadlock is not possible, that some sequence of events should always lead to some particular result. Specific properties are expressed in a formal specification language. [2]

Properties can be expressed and formally checked without running code. If a proof fails, a *counter-example* is generated. Formal verification is expensive however, especially as code base size grows, potentially leading to a state-space explosion.

Properties can also be used to generate test cases, often coming up with values the programmer may not think to try. Again, a *c*ounter-example, or falsifying example, is generated when a test case fails to yield the expected result. The difference is that such property-based testing tools run the code and do not prove that a property holds in all cases by attempting to exhaustively explore the state space of possibilities. But more test cases are created and run than would be likely with hand-written unit tests. Additionally, if formal verification is feasible for a given code base, these properties can also be used for that purpose. [2]

Consider a function, rpn(str), that takes a string containing a reverse polish notation expression, such as "12 4 * 5 +" corresponding to the expression 12*4+5, and returns the numeric result, in this case: 53. A pytest [3] function that would cover this case is:

```python
def test_rpn_mul_then_add():
    assert rpn("12 4 * 5 +") == 53.0
```

Many other tests are possible here of course, e.g. involving negative, large or small values. Rather than writing many such unit tests, property-based testing libraries such as hypothesis [4] generate many test cases given a property that should always be true, such as: a string containing two real numbers a and b, followed by the "*" operator, followed by another real number c, followed by a "+" operator yields the result a*b+c. The following hypothesis-based unit test function does exactly this:

```python
@given(a=strategies.floats(allow_nan=False, allow_infinity=False),
       b=strategies.floats(allow_nan=False, allow_infinity=False),
       c=strategies.floats(allow_nan=False, allow_infinity=False))
def test_rpn_mul_then_add(a, b, c):
    assert rpn("{} {} * {} +".format(a, b, c)) == a*b+c
```

So, What Kind of Testing?

If the question to be answered is: does code yield the same result when parallelised as the original single-threaded code? When requested to make C++ or Fortran code run faster via OpenMP, MPI, CUDA, Spark or other parallelisation approach, there may only be a text file as expected output. It may be sufficient to ask: once the code is parallelised, does running it produce the same output as this *golden reference* file, while completing sooner? Since the order or execution may be different, so may the output, hence the need to capture or check only the essential output required to answer this question.

When code containing units without tests is inherited, before modifying or extending the code, writing unit tests to understand behaviour and guard against future change allows modification (including possible bug fixes or performance enhancements) with some confidence. New unit tests should be added for extensions to functionality to guard against future changes re-introducing the same bug.

If writing unit tests when developing code from the ground up, should tests be written before code, so-called Test-Driven Development (TDD)? One benefit of this is that the developer is forced to think about required inputs and expected outputs in terms of an interface, even before a function is written.

Resources

Determining the appropriate approach to verification is important to the fitness, reliability and ongoing maintenance of research software. An ongoing effort to organise a set of resources (see QR code), training materials, and shared experience is intended to benefit a community of software development practitioners. A repository for case studies and patterns derived from experience is being created for development activities such as porting and parallelising in conjunction with methods such as reference testing, TDD, and property-based testing. This poster is also intended to stimulate discussion.

FOR FURTHER INFORMATION

David Benn
IMT
David.Benn@csiro.au
https://www.csiro.au/en/Research/Technology/Scientific-computing

REFERENCES

1. SOFTWARE TESTING ANTI-PATTERNS. AVAILABLE FROM: HTTP://BLOG.CODEPIPES.COM/TESTING/SOFTWARE-TESTING-ANTIPATTERNS.HTML, ACCESSED ON MARCH 11 2020
2. COGENT. AVAILABLE FROM: HTTPS://TS.DATA61.CSIRO.AU/PROJECTS/TS/COGENT.PML, ACCESSED ON MARCH 11 2020
3. PYTEST. AVAILABLE FROM: HTTPS://DOCS.PYTEST.ORG/EN/LATEST/, ACCESSED ON MARCH 11 2020
4. HYPOTHESIS. AVAILABLE FROM: HTTPS://HYPOTHESIS.READTHEDOCS.IO/EN/LATEST/, ACCESSED ON MARCH 11 2020