Runtime Verification of Scientific Software

PARANOID SCIENTIST

Maxwell Shinn[1], Clarence Lehman[2], Ruzica Piskac[3]
[1]Interdept. Neuroscience Program, Yale University
[2]Dept. of Ecology, Evolution, and Behavior, University of Minnesota
[3]Dept. of Computer Science, Yale University

Yale

Introduction

Scientific software has several defining characteristics:

- **Expected result is unknown.** Only trivial cases may be compared to expected results
- **Run for investigation.** Implemented for personal investigation, run a limited number of times, usually by the programmer
- **Moving target.** Specifications unclear upfront and change rapidly (weekly or even daily)
- **Usually limited or no tests.** Culture and code structure[1] prevent extensive testing
- **Bugs can be dangerous.** Bugs can have major consequences for public health and public policy[2]

Our approach

Developed the *Paranoid Scientist* Python library:

- **Entry and exit conditions.** Runtime checks of function conditions
- **Refinement types.** Conditions specified with refinement types[3]
- **Automated testing.** "Free" offline unit/fuzz testing

Advantages compared to:

- **Static typing**: Favors human over machine interpretation and understanding[4]
- **Contracts:** Refinement types conceptually simple[5]
- **Full verification:** Minimal learning curve and development time[6]

Motivating example

Example bug from a data analysis pipeline:

```
def graph_measure(filename):
  # Load data from file
  timeseries = load_from_csv(filename)
  # Generate a correlation matrix
  corr_matrix = corr_coef(timeseries) # diag = 1+10^-10
  # Normalize from [-1,1] → (-∞,∞)
  normalized = arctanh(corr_matrix) #  diag → NaN
  # Convert to an undirected graph
  G = matrix_to_graph(normalized) # NaN → 0
  # Compute statistics on the graph
  return graph_clustering(G) # ???
```

Simple examples

```
from paranoid.types import Number, Positive, \
                    Natural1, Natural0, Range
from paranoid.decorators import accepts, returns, \
                    requires, ensures

@accepts(x=Number, y=Number)
@returns(Positive)
@requires("x != y")
def some_formula(x, y):
    return 1/((x-y)**2)

@accepts(Number)
@returns(Number)
@ensures("x >= x` --> return >= return`")
def cube(x):
    return x**3
```

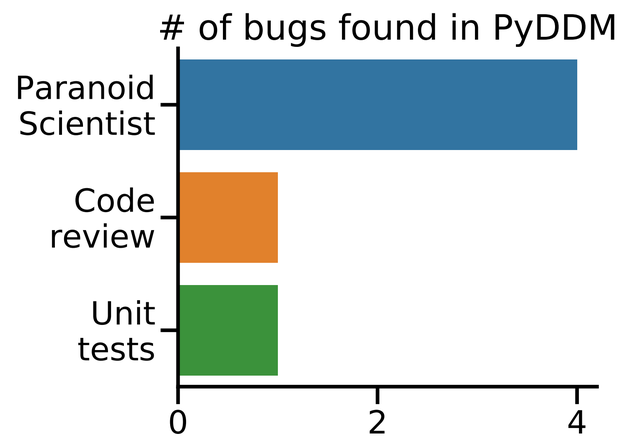
Entry conditions

Exit conditions

(monotonicity)

Case Study

- PyDDM simulates SDEs to understand decision-making
- Our tool found **4 major bugs** difficult to detect otherwise
- It also found **1 bug in user code**
- Unit tests and code review each found 1 bug



Future directions

- Case studies with scientific software
- Increase efficiency (currently ~20-150% performance penalty)
- Improve tooling and usability

More information

Code:
github.com/mwshinn/paranoidscientist
Documentation:
paranoid-scientist.readthedocs.io
Email:
maxwell.shinn@yale.edu

References

[1]Upulee Kanewala and James M. Bieman. Testing scientific software: A systematic literature review. Information and Software Technology, 56(10):1219 – 1232, 2014.
[2]Thomas Herndon, Michael Ash, and Robert Pollin. Does high public debt consistently stifle economic growth? a critique of Reinhart and Rogoff. Cambridge Journal of Economics, 38(2):257–279, 2014.
[3]Tim Freeman and Frank Pfenning. Refinement types for ML. SIGPLAN Not., 26(6):268–277, May 1991.
[4]MyPy project: http://mypy-lang.org/index.html
[5]PyContracts project: https://andreacensi.github.io/contracts/
[6]Nagini project: https://github.com/marcoeilers/nagini