A Survey of Approaches for Automated Unit Testing

Peter Carr
Ron Kneusel

Outline

• Introduction/Motivation
• Concolic Testing
• Random Testing
• Evolutionary Testing
• Random/Evolutionary Experiment and Results
• Comparison of the Different Approaches
• Future Work
Introduction – Why Automate Unit Testing?

• Motivation: We’d like to write quality code but doing this requires testing it with inputs to see its behavior
• Must have a good path coverage to ensure reliability, but coming up with inputs to exercise all paths is hard
• Bugs tend to occur in boundary cases, but we don’t think of these cases, hence we don’t test them!

Concolic Testing – A (Very) Brief Overview

• Concolic is a modified form of symbolic execution (sym-ex) utilizing concrete inputs and path enumeration
• Example:
  1: if(x>y) {
  2:       x = x + y;
  3:       y = x - y;
  4:       x = x - y;
  5:       if (x - y > 0)
  6:       assert(false);
  }

Concolic Example (1)

- Sym-ex represents x and y as variables x0 and y0, whereas x and y get concrete int values in concolic.
- Path condition (pc) a boolean formula representing the path currently taken, initialized to true.
- At each branch, a logic formula is conjoined to pc representing the possible inputs.
  - At 1), x0 < y0 or x0 >= y0 can hold so this represents a fork in the path: ie. pc = x0 < y0 and pc' = x0 >= y0.
  - 6) cannot be executed because pc = x0 > y0 & (y0 - x0 > 0) is false.

Concolic Example (2)

- For concolic, a concrete input causes a certain branch to be taken, and that is recorded to the pc.
  - If x = 3 and y = 4 then AND x <= y to the pc.
  - At next execution negate the conjunct to generate inputs that force a different path.
  - Constraint solver is required to generate inputs (concolic) or to see if pc has become false (symbolic).
Concolic vs. Symbolic Execution

• Sym-ex is computationally infeasible for many programs
  – Tries to find every execution path, which is exponential in the number of branches
• Concolic tries to reduce the number of paths and also address problem with constraint solver
  – If constraint can’t be solved in concolic, default to random values

Random Testing

• Conceptually simple-
  – To test function, f(a,b), randomly select arguments, a and b, and apply them to f. If there is an error, a bug has been found.
• Depending upon the dimensionality and domain of f, one might wait a very long time before getting a representative set of inputs to f.
• Something this simple is just begging to be expanded upon...
Random Testing

• ART – Adaptive Random Testing
  – Still random, but consciously select new “random” inputs to f() that are “well away” from any previous input – attempt to cover the input space of f() in a more intelligent manner.

• Quasi-Random Testing
  – RT, but use a shuffled quasi-random generator – this is also “space filling”

• Plus many more... see the paper!

Evolutionary Testing

• Use an evolutionary (genetic) algorithm to evolve a set of inputs to f().
  – Part of “dynamic test data generation”
  – Instrument the code to report information about the function as it is executed.
  – Use this information to decide on the “fitness” of the inputs.
  – Evolve a new set of inputs based on previous fitness scores.
The Experiment

• Motivation
  – Most RT/EA testing designed to uncover bugs – to work until a specific target piece of code has been reached
  – In some cases branch coverage is more desirable goal. ie, to prove all code paths have been executed by the test suite.
  – For example, this is of high interest to the FDA when certifying a new medical device.

The Setup

• Tested 100 randomly generated IDL functions
  – A series of nested If-Else statements instrumented to track when a branch has been taken
  – 2 to 5 input parameters
    – Constrained to scalar 32-bit integer inputs
  – Cyclomatic complexity:
    • Mean 16.2 +/- 4.6, min 7, max 29

• Goal: a set of inputs that maximizes branch coverage
Randomized Testing Approach

- **RT**
  - Pick some inputs, evaluate f(), if this adds a new branch to the existing set of covered branches, keep this set of inputs
  - Repeat *ad nauseam*
  - *So simple my six year old gets it!*

- **Quasi-RT**
  - RT but use a quasi-random generator

Quasi-Random Numbers

Quasi-random

Pseudo-
RT results

<table>
<thead>
<tr>
<th></th>
<th>Coverage</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Median</th>
<th>Stddev</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT</td>
<td>0.10</td>
<td>1.0</td>
<td>0.76</td>
<td>0.79</td>
<td>0.19</td>
<td>0.19</td>
<td></td>
</tr>
<tr>
<td>Quasi</td>
<td>0.10</td>
<td>1.0</td>
<td>0.81</td>
<td>0.86</td>
<td>0.20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#inputs</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Median</th>
<th>Stddev</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT</td>
<td>1</td>
<td>8</td>
<td>3.1</td>
<td>2</td>
<td>1.5</td>
<td>&lt;10^-6</td>
</tr>
<tr>
<td>Quasi</td>
<td>1</td>
<td>11</td>
<td>4.5</td>
<td>3</td>
<td>2.3</td>
<td></td>
</tr>
</tbody>
</table>

Surprisingly, no statistically significant difference between plain RT and Quasi-RT, contrary to other papers.

Possible causes:
- “Irrational” auto-generated code
- Excessive number of tests – space filling regardless of generation type (mean tests ~2,000,000) ???

How EA Works

BEGIN

INITIALISE population with random candidate solutions;
EVALUATE each candidate;
REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
  1 SELECT parents;
  2 RECOMBINE pairs of parents;
  3 MUTATE the resulting offspring;
  4 EVALUATE new candidates;
  5 SELECT individuals for the next generation;
OD
END

from Ch 2, A.E. Eiben and J.E. Smith
EA Results

<table>
<thead>
<tr>
<th>Coverage</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Median</th>
<th>Stddev</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EA</td>
<td>0.10</td>
<td>0.87</td>
<td>0.59</td>
<td>0.59</td>
<td>0.15</td>
<td>$&lt;10^{-6}$</td>
</tr>
<tr>
<td>RT</td>
<td>0.10</td>
<td>1.0</td>
<td>0.76</td>
<td>0.79</td>
<td>0.19</td>
<td>$&lt;10^{-6}$</td>
</tr>
<tr>
<td>Quasi</td>
<td>0.10</td>
<td>1.0</td>
<td>0.81</td>
<td>0.86</td>
<td>0.20</td>
<td>$&lt;10^{-6}$</td>
</tr>
</tbody>
</table>

EA-
- 100 organisms (a collection of inputs)
- 100 cells per organism (an input)
- run until no change for 30 generations
- keep the best organism found so far (apart from the population)

Probabilities-
- of mating, 70%
- otherwise mutate or replicate, (split 50/50)

EA Results

- Poor performance compared to RT
- Why?
  – Simple coverage metric as objective function inadequate to force a true search of the input space
  – A targeted branch approach is (probably) necessary
So Which Technique to Use?

- OOP testing requires good inputs and good method call sequences to get objects into desirable states
- Concolic is good at generating inputs to cover code with complex logic and structure
- Evolutionary techniques can be used to find desirable method call sequences
- Combine the two! (recent work in this area)

Future Work

- Explore EVACON framework which combines aspects of concolic and evolutionary testing for OOP
- Select a better objective function
- Target branches
  - Evolve a set of inputs towards specific branches
- Switch from EA to PSO
  - Demonstrated more powerful than EA in many cases
- Recall that IDL is an array-processing language
  - How to select inputs when the space of inputs consists of multi-dimensional arrays?