Randomized Software Testing

Alex Vondrak

ajvondrak@csupomona.edu

April 25, 2012

Software Quality

Can be measured along two major axes:

- Correctness
- Efficiency

Software Quality

Can be measured along two major axes:

- Correctness
- Efficiency

But what does it mean to *be* correct?

1. Specification
    - *When* does a test pass or fail?
    - Formal specification: difficult to do for the whole system
    - Informal specification: vague and difficult to verify
2. Validation
    - *Did* a test pass or fail?
    - Formal validation: proofs of correctness (still difficult)
    - Informal validation: try a lot of test cases

Test Cases

- Easier to validate with test cases...
- ...But, they're inherently existential ($\exists$)
- Useful properties are usually universal ($\forall$)

Example

Type signatures are universal properties:

```
int sqrt(int x) { ... }
```

$\forall x \in \mathbb{Z}, \sqrt{x} \overset{?}{\in} \mathbb{Z}$

✓ $4 \in \mathbb{Z}, \quad \sqrt{4} = 2 \in \mathbb{Z}$

✓ $9 \in \mathbb{Z}, \quad \sqrt{9} = 3 \in \mathbb{Z}$

✗ $8 \in \mathbb{Z}, \quad \sqrt{8} \approx 2.83 \notin \mathbb{Z}$

Test Cases

Where Do They Come From?

- Unit testing
  - Idea: Specify single cases by hand
  - Pro: Makes sure known edge cases won't break code again
  - Con: Not very general
- Automatically generate "good" test data
  - Idea: Discover what data follows each control-flow path
  - Pro: In the best case, exhaustive
  - Con: Usually too complex to do in general
- Randomly generate data
  - Idea: Try *a lot* of cases, see if any fail
  - Pro: Automatic, yet simple
  - Con: Will data be "random enough"?

QuickCheck

- Homepage: `www.cse.chalmers.se/~rjmh/QuickCheck/`
- A randomized specification-based testing tool
  - By handling validation, specification is much easier
  - Formal specification needn't be complete, since we aren't fully proving its correctness
- Written in Haskell (`haskell.org`)
  - Write properties as actual code—executable specifications
  - Aims to be small, simple, and lightweight
  - Ports to other languages exist (see `en.wikipedia.org/wiki/Quickcheck`)

A Simple Haskell Function

- Suppose we want to test a list-reversing function in Haskell

Definition

The reverse function takes a list of integers and returns a list of integers.
If the list is empty, its reverse is the empty list.
Otherwise, append the 1^{st} item to the end of the remaining items reversed.

Example

```
reverse [1,2,3] == ...
```

A Simple Haskell Function

- Suppose we want to test a list-reversing function in Haskell

Definition

`reverse :: [Int] -> [Int]`

If the list is empty, its reverse is the empty list.
Otherwise, append the 1st item to the end of the remaining items reversed.

Example

`reverse [1,2,3] == ...`

A Simple Haskell Function

- Suppose we want to test a list-reversing function in Haskell

Definition

```
reverse :: [Int] -> [Int]
reverse [] = []
```

Otherwise, append the 1st item to the end of the remaining items reversed.

Example

```
reverse [1,2,3] == ...
```

A Simple Haskell Function

- Suppose we want to test a list-reversing function in Haskell

Definition

```
reverse :: [Int] -> [Int]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Example

```
reverse [1,2,3] == ...
```

A Simple Haskell Function

- Suppose we want to test a list-reversing function in Haskell

Definition

```
reverse :: [Int] -> [Int]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Example

```
reverse [1,2,3] == reverse [2,3] ++ [1]
```

A Simple Haskell Function

- Suppose we want to test a list-reversing function in Haskell

Definition

```
reverse :: [Int] -> [Int]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Example

```
reverse [1,2,3] == (reverse [3] ++ [2]) ++ [1]
```

A Simple Haskell Function

- Suppose we want to test a list-reversing function in Haskell

Definition

```
reverse :: [Int] -> [Int]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Example

```
reverse [1,2,3] == ((reverse [] ++ [3]) ++ [2]) ++ [1]
```

A Simple Haskell Function

- Suppose we want to test a list-reversing function in Haskell

Definition

```
reverse :: [Int] -> [Int]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Example

```
reverse [1,2,3] == (([] ++ [3]) ++ [2]) ++ [1]
```

A Simple Haskell Function

- Suppose we want to test a list-reversing function in Haskell

Definition

```
reverse :: [Int] -> [Int]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Example

```
reverse [1,2,3] == ([3] ++ [2]) ++ [1]
```

A Simple Haskell Function

- Suppose we want to test a list-reversing function in Haskell

Definition

```
reverse :: [Int] -> [Int]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Example

```
reverse [1,2,3] == [3,2] ++ [1]
```

A Simple Haskell Function

- Suppose we want to test a list-reversing function in Haskell

Definition

```
reverse :: [Int] -> [Int]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Example

```
reverse [1,2,3] == [3,2,1]
```

Properties

- In QuickCheck, properties are written as Haskell functions
- In their simplest form, they return booleans
  - "Did the test condition pass?"

Example (prop_reverseSingleton)

Observe that

$$\forall\, x :: \mathbf{Int},\quad \mathbf{reverse}\ [x] == [x] \text{ should be } \mathbf{True}$$

So, we write

```
prop_reverseSingleton :: Int -> Bool
prop_reverseSingleton x =
  reverse [x] == [x]
```

Properties

- Inputs are considered to be universally quantified over their types
- Thus, writing QuickCheck properties is like writing a formal spec

Example (`prop_reverseReverse`)

Observe that

$\forall$ `xs :: [Int],` `reverse (reverse xs) == xs` should be **True**

So, we write

```
prop_reverseReverse :: [Int] -> Bool
prop_reverseReverse xs =
  reverse (reverse xs) == xs
```

Properties

- Properties can be defined for multiple inputs
- In Haskell, the syntax looks like "`in1 -> in2 -> out`"

Example (prop_reverseAppended)

```
prop_reverseAppended :: [Int] -> [Int] -> Bool
prop_reverseAppended xs ys =
  reverse (xs ++ ys) = reverse ys ++ reverse xs
```

For instance,

```
reverse ([1,2] ++ [3,4])
== reverse [1,2,3,4]
== [4,3,2,1]
== [4,3] ++ [2,1]
== reverse [3,4] ++ reverse [1,2]
```

Verifying Properties

- Instead of proving $\forall$, we show $\exists$ for a large number of cases
- By default, 100 random tests are generated
- Can use `quickCheck` function at Haskell prompt

Example (Running QuickCheck)

```
Main> quickCheck prop_reverseAppended
OK: passed 100 tests.
```

- Not much to look at when property passes. . .
- What if code is buggy?
- What if property is wrong?

Verifying Properties

Typo In Property

```
prop_reverseAppended xs ys =
  reverse (xs ++ ys) = reverse xs ++ reverse ys
```

Example

```
Main> quickCheck prop_reverseAppended
Falsifiable, after 1 tests:
[2]
[-2,1]
```

That is,

reverse ([2] ++ [-2,1]) != **reverse** [2] ++ **reverse** [-2,1]

Implication

- Many properties are conditional
- $A \implies B$: "If A is true, then B must be true"

Example

Suppose we define `insort :: Int -> [Int] -> [Int]`:

- `insort x xs` inserts `x` into `xs` in order (`xs` must stay sorted)

Define the predicate

```
sorted :: [Int] -> Bool
sorted [] = True
sorted [x] = True
sorted (x1:x2:xs) = x1 <= x2 && sorted (x2:xs)
```

Then,
$\forall$ `x :: Int`, `xs :: [Int]`, `sorted xs` $\implies$ `sorted (insort x xs)`

Implications in Properties

- QuickCheck defines an infix **==>** operator
- Only count test case if antecedent is **True**
- Otherwise, regenerate the test data

Example (prop_insortStaysSorted)

```
prop_insortStaysSorted :: Int -> [Int] -> Property
prop_insortStaysSorted x xs =
  sorted xs ==> sorted (insort x xs)
```

Property is a new return type for this “retrying” behavior

Generating Matching Tests

- What about "dumb" antecedents?
  - Always **`False`**
  - Rarely **`True`**—e.g., `bothEvenAndPrime`
- Don't want to get stuck regenerating
- Impose a limit of 1000 "do-overs"

Example

```
Main> quickCheck prop_insortStaysSorted
Arguments exhausted after 97 tests.
```

- Random lists are rarely ordered
- Is passing 97 tests enough?

Monitoring Test Cases

- 97 tests may pass. . .
- . . . But is the data well-distributed?
- Impossible to select uniformly randomly from infinite sets
  - $\mathbb{Z}$
  - Arbitrary lists
  - All binary trees
  - . . .
- Need to know what sort of data gets generated
  - Specifically, what gets through to the right-hand side of **==>**?

Monitoring Test Cases

Trivial Data

- Random data may contain duplicates
- Some data is so simple that it's hardly worth considering

Example (trivial)

`insort x [] == [x]` is trivially sorted, so...

```
prop_insortStaysSorted :: Int -> [Int] -> Property
prop_insortStaysSorted x xs =
  sorted xs ==>
    trivial (null xs) (sorted (insort x xs))

Main> quickCheck prop_insortStaysSorted
OK, passed 100 tests (43% trivial).
```

Monitoring Test Cases

Details

- `trivial` is just binary—"trivial" or "non-trivial"
- What about whole distribution of data? (Over what variable?)

Example (`collect`)

```
prop_insortStaysSorted :: Int -> [Int] -> Property
prop_insortStaysSorted x xs =
  sorted xs ==>
    collect (length xs) (sorted (insort x xs))
```

Monitoring Test Cases

Details

- `trivial` is just binary—"trivial" or "non-trivial"
- What about whole distribution of data? (Over what variable?)

Example (`collect`)

```
Main> quickCheck prop_insortStaysSorted
OK, passed 100 tests.
49% 0.
32% 1.
12% 2.
4% 3.
2% 4.
1% 5.
```

Monitoring Test Cases

Generalizing

- `trivial` and `collect` are doing the same sort of things
  - Keep track of a property
  - Return test result *unchanged*—can compose observations

Example (`classify`)

```
prop_insortStaysSorted :: Int -> [Int] -> Property
prop_insortStaysSorted x xs =
  sorted xs ==>
    collect (length xs)                   $
    classify (sorted (x:xs))     "at-head" $
    classify (sorted (xs++[x])) "at-tail" $
    sorted (insort x xs)
```

Monitoring Test Cases

Generalizing

- `trivial` and `collect` are doing the same sort of things
  - Keep track of a property
  - Return test result *unchanged*—can compose observations

Example (`classify`)

```
Main> quickCheck prop_insortStaysSorted
OK, passed 100 tests.
58% 0, at-head, at-tail.
22% 1, at-tail.
13% 2.
4% 1, at-head.
3% 3.
```

Fixing the Distribution

Tools

- `==>` can skew data, so try generating *desired* data
  - ...But automation is complicated
- Instead, QuickCheck gives us easy random selection functions
  - `choose (a,b)`—random number between `a` & `b`
  - `oneof xs`—pick item with uniform probability
  - `frequency [(w1, x1), ...]`—pick item with weighted probability
- Main "interface": the `arbitrary` function
  - Tied to Haskell's type system (beyond our scope)
  - Generates arbitrary data for a given type
  - Instances already defined for built-ins: integers, booleans, lists, functions, etc.
- `sized`—gives access to a size parameter
  - Parameter is an upper bound on datum's size (e.g., list length)
  - Automatically increases as more tests pass
  - Thus, if there are failures, they happen on smaller data

Fixing the Distribution

In Action

- Needn't define a whole new data type to randomly generate
- Can just define a generator function

Example (forAll)

- Suppose we have an sortedList generator, defined roughly like

```
sortedList = sort (arbitrary :: [Int])
```

- forAll supplies the generator's data to the test
- Thus, we're guaranteed to generate 100 sorted lists

```
prop_insortStaysSorted :: Int -> Property
prop_insortStaysSorted x =
  forAll sortedList (\xs -> sorted (insort x xs))
```

What We've Learned

- Automatically generating test data is hard
- QuickCheck combines formal specs with random tests
- Can still get effective coverage from random tests
  - Formal properties force programmers to *think* about code...
  - ...But, needn't *fully* specify program, which is tedious
  - The more random tests we run, the more we're sure our code—on average—won't fail (like a Monte Carlo approximation of π)
- Randomized testing has been used effectively
  - Simple ideas port to many languages
  - Testing gets done faster or more completely in limited time
  - E.g., Ericsson team discovered bugs in *already* well-tested product
    - See "Testing Telecoms Software with Quviq QuickCheck" by Arts, Hughes, Johansson, and Wiger