Coverage-guided property-based testing

Stevan A

14.3.2025, BOBkonf (Berlin)

Background

- Coverage-guided property-based testing
  - Property-based testing (PBT)
  - Coverage-guided fuzzing (CGF)

Overview of this talk

- History of PBT and CGF
- How PBT and CGF work
- What the difference is
- How to combine the two

History of PBT

- John Hughes and Koen Claessen at Chalmers (1999)
- Testing formal specifications, type theory connection
- QuickCheck
- Mathematical properties (proof by structural induction)
- More “functional”

History of CGF

- Fuzzing (without CG), dial-up modem and rain, Barton Miller (class project 1988, University of Wisconsin)
- Fuzz UNIX command-line utilities
- Combine fuzzing with evolutionary algorithms (2007)
- AFL by Michał Zalewski (2013).
- More “imperative”

How does PBT work?

Example

- Nutshell: generate, check, if failed then shrink

Success

```
prop_reverseReverse : Property (List a)
prop_reverseReverse xs = reverse (reverse xs) == xs

test = check (genList 8 genInt) 123
>>> test prop_reverseReverse
Passed -- (100 tests generated)
```

Failure

```
prop_badReverse xs = reverse xs == xs
>>> test prop_badReverse
Failed (Cons 0 (Cons 1 Nil)) -- (Shrunk 6 times)
```

How does PBT work?

Generate

```
type Gen a = Prng -> a

genInt : Gen Int
genInt prng = random prng

genList : Length -> Gen a -> Gen (List a)
genList 0 gen prng = Nil
genList n gen prng =
  let
     (prng', prng'') = split prng
     x = gen prng'
   in
     Cons x (genList (n - 1) gen prng'')
```

How does PBT work?

Check

```
type Property a = a -> Bool

type Result a = Passed | Failed a

check : Gen a -> Seed -> Property a -> Result a
check gen seed prop = go 100 (newPrng seed)
  where
    go 0 prng = Passed
    go n prng =
      let
        (prng', prng'') = split prng
        x = gen prng'
      in
        if prop x then go (n - 1) prng''
                  else Failed (shrink prop x)
```

How does CGF work?

Example

- Nutshell: start with some corpus of inputs, pick one input, mutate it, check coverage, promote mutations that increase coverage, repeat until crash

Programs

```
>>> fuzz "/bin/ls" (Cons "foo" (Cons "\0" Nil))
```

Functions

```
>>> fuzz f (Cons "bar" Nil)
```

```
func f (input []byte) {
    if input[0] == 'b' {
        if input[1] == 'a' {
            if input[2] == 'd' {
                if input[3] == '!' {
                    error "input must not be bad!" } } } }
```

How does CGF work?

```
fuzz : ProgramOrFunction -> List Bytes -> Bytes
fuzz p corpus = go corpus (initEnergy corpus) noCoverage
  where
    go corpus energies coverage =
      let
        input = choose corpus energies
        input' = mutate input
        coverage' = execute p input'
      in if crashed coverage' then return input' else
        if isInteresting input' coverage coverage'
        then
          let
            corpus' = append corpus input'
            energies' = assignEnergy corpus' energies
          in
            go corpus' energies' coverage'
        else
         go corpus energies coverage
```

Difference between PBT and CGF?

- Generation
  - PBT requires you to write generators for your input datatypes
  - CGF merely requires some sample inputs and will mutate from there
- Test execution time
  - PBT typically takes subsecond to a minute to run
  - CGF can run for hours or even days
- Test depth/coverage
  - PBT depends on how well designed the generators are, doesn't have "memory"
  - CGF has "memory", learns and improves from past tests
- Correctness
  - CGF typically only checks if the program crashes
  - PBT let's you specify an arbitrary relation between inputs and outputs

Combining PBT and CGF

Motivation and plan

- PBT-style generators to speed up test exeuction
- CGF-style coverage-guidance to get "deeper" coverage
- Dan Luu's post (2015)
- Most progress have been from "CFG to PBT" (parse random bytes into data)
- Today I'd like to show more of a "PBT to CFG" solution
  - Key insight: use PBT's built-in coverage annotations
  - No need to query compiler for coverage

Combining PBT and CGF

PBT's built-in coverage (1/2)

```
- type Property a = a -> Bool
+ type Property a = a -> Result a

type Result a =
  { ok             : Bool
  , labels         : Set String
  , counterExample : Maybe a
  }


property : Bool -> Result a
property bool =
  { ok             = bool
  , labels         = Set.empty
  , counterExample = Nothing
  }
```

Combining PBT and CGF

PBT's built-in coverage (2/2)

```
label : String -> Result a -> Result a
label s result = result.labels += s

classify : Bool -> String -> Result a -> Result a
classify True  s result = label s result
classify False s result = result
```

Combining PBT and CGF

Example

```
bad : Property String
bad s =
  classify (s[0] == 'b') "Found b at first position!" (
  classify (s[1] == 'a') "Found a at second position!" (
  classify (s[2] == 'd') "Found d at third position!" (
  classify (s[3] == '!') "Found ! at fourth position!" (
  if s == "bad!"
  then property False
  else property True))))
```

```
>>> check (genList 4 genASCIIByte) 123 bad
Passed

>>> guided (genList 4 genASCIIByte) 123 bad
Failed "bad!"
```

Combining PBT and CGF

Evolutionary algorithm

```
guided : Gen a -> Seed -> Property (List a) ->
         Result (List a)
guided gen seed prop = go 100 (newPrng seed) Nil Set.empty
  where
    go 0 prng xs cover = { ok = True, labels = cover }
    go n prng xs cover =
      let
        (prng', prng'') = split prng
        x = gen prng'
        result = prop (append xs x)
        cover' = Set.union cover result.labels
      in
        if ok result
        then if Set.size cover < Set.size cover'
             then go (n - 1) prng'' (append xs x) cover'
             else go (n - 1) prng'' xs cover
        else result.cE = Just (shrink prop (append xs x))
```

Combining PBT and CGF

plain PBT vs CGPBT

- 1 ASCII character $= 7$ bits $\Rightarrow 2^7 = 128$ possibilities
- Probability of generating "bad!" using plain PBT: $\frac{1}{128} \times \frac{1}{128} \times \frac{1}{128} \times \frac{1}{128} = 3.72529 \times 10^{-7}\%$
- Probability of winning the lottery: 1 in 292.2 million $= 3.42231 \times 10^{-7}\%$
- Probability of generating "bad!" using coverage-guidance: $\frac{1}{128} + \frac{1}{128} + \frac{1}{128} + \frac{1}{128} = 3.125\%$
- In order words coverage-guidance turns an exponential problem into a polynomial problem!

Recap

- History of PBT and CGF
- How PBT and CGF works
- How they are different
- How to combine them
- What the benefit of combining them is: from exponential to polynomial!

Conclusion, further work and questions

- First version of QuickCheck (1999) has all the pieces to enable this!
- For more details and a full working implementation, see: https://stevana.github.io/talk/bobkonf-2025.html
- Can get stuck in local maxima, backtrack when no progress is made (`assignEnergy`)
- Thanks for listening! Questions?