Testing debugging & Verification - Property based testing (Stateless)

Atze van der Ploeg

Property based - Testing

- Writing unit test takes a lot of effort!
- More unit test = more certainty
- Automate!
- Generate *random* inputs and check property

Example

```
int[] sort(int[] input)
```

Specification:
Requires: A non-null array as input
Ensures: A new array that is sorted in ascending order, that is a permutation of the input array

Property based Testing

- Generate a *random* input that satisfies the *precondition*
- Feed it to the function
- Check that a property on the ouput holds

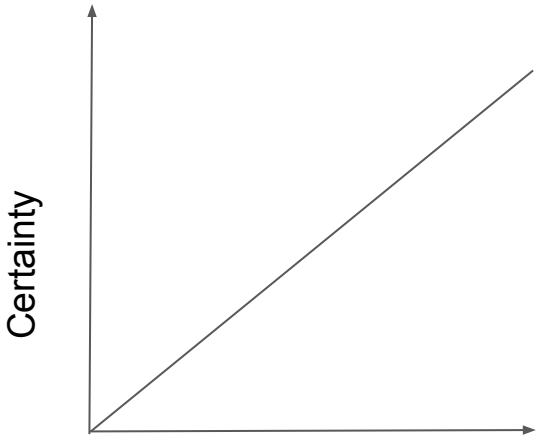
Property : post-condition holds

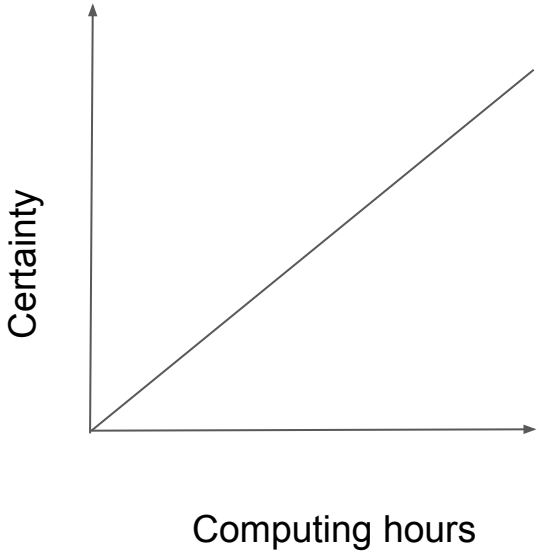
```
bool singleTest(){
  int[] input  = generateRandomArr();
  int[] output = sort(input);
  return isSorted(output) && isPermutationOf(output, input);
}
```

Doesn't have to be efficient! Run many times!

Unit vs Property based testing

Unit testing





Man hours = expensive, Compute time = cheap

Random Input generation

```
int[] random(int maxsize) {



}
```

Random Input generation

```
int[] randomArray(int max, int maxsize) {
   Random r = new Random();
   size = r.nextInt(maxsize);
   int[] res = new int[size];
   for(int i = 0 ; i < size ; i++) {

      res[i] = r.nextInt(max);

   }

   return res;

}
```

Algebraic properties

```
int[] reverse(int[] input)
```

Specification:
Requires: A non-null array as input
Ensures: A new array that contains the elements of the input in reverse order

Option 1: Check postcondition
Option 2: Check algebraic properties:

```
reverse({x}) = {x}

reverse(reverse(x)) = x
```

Algebraic properties: inverses

```
JavaAST parse(String source)
```

```
String prettyPrint(JavaAST program)
```

What property should hold here?

parse(prettyPrint(x)) = x 

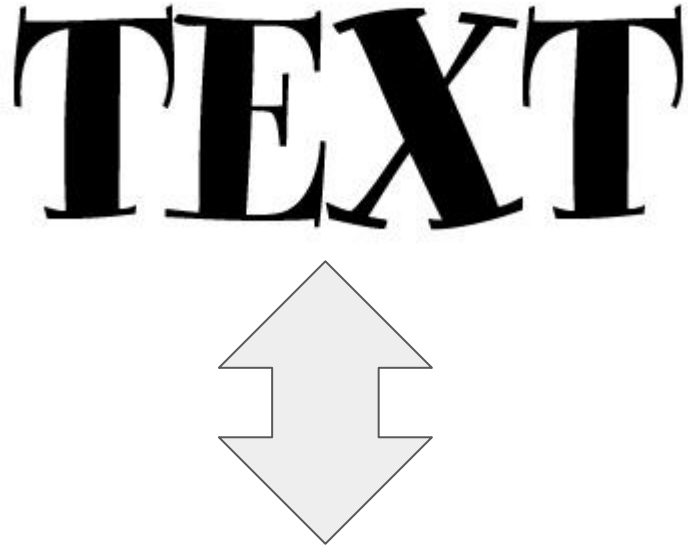
prettyPrint(parse(x)) = x 

More inverses

Context: Some systems only allow text:

- USENET
- Email
- URL
- XML

We want to use binary files in these systems anyway...
Solution: Binary 2 Text via Base64 encoding!




```
String base64Encode(byte[] data)
```

```
byte[] base64Decode(String data)
```

base64Decode(base64Encode(x)) = x

All binary data can be encoded, but not all strings can be the result of such an encoding...

base64Encode(base64Decode(x)) = x

What property should hold here?

More algebraic properties

```
int[] append(int[] l, int[] r)
```

Specification:
Requires: non-null arrays as input
Ensures: A new array such that res[i] == i < l.length? l[i] : r[i - l.length]

Algebraic properties?

forall r. append({},r) == r
forall r. append(l, {}) == l
forall l r z. append(append(l,r),z) == append(l, append(r,z))

Monoid properties!

forall r. 0 + r = r
forall l. l + 0 = l
forall l r z . (l + r) + z = l + (r + z)

Property: pointwise equivalence of functions

```
int[] superSmartSort(int[] input)
```

Generate random input,
Check that they give the same output

∀x. superSmartSort(x) == simpleButSlowSort(x)

```
int[] simpleButSlowSort(int[] input)
```

Property: No runtime errors

```
String notReallySureWhatThisDoes(byte[] data)
```

Specification:
Requires: Something
Ensures: No idea

Can we test something in this situation?

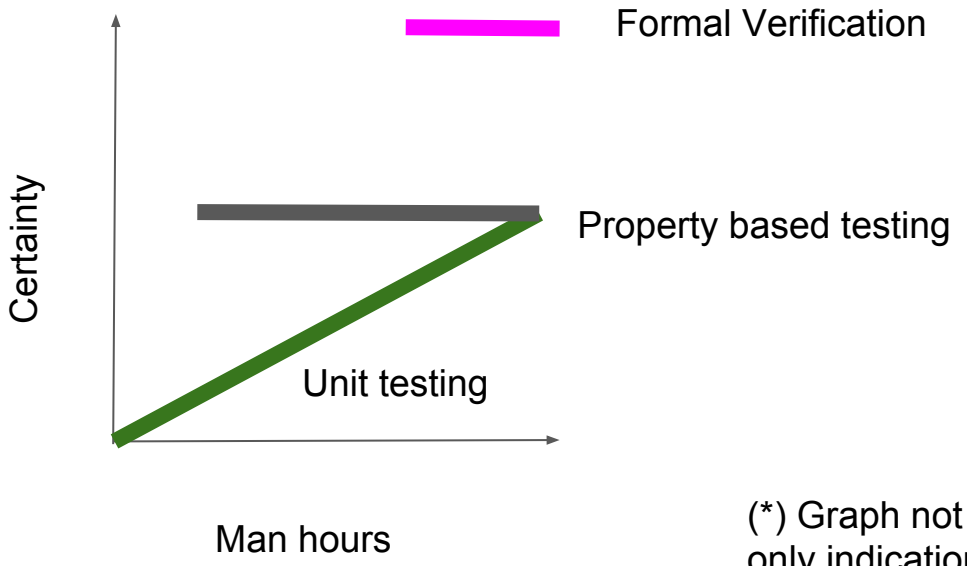
- Generate a *random* input that satisfies the precondition
- Feed it to the function

Will find runtime errors such as out-of-bounds errors, cast errors, non-termination (with timeout) etc.

Types of properties

- Post condition holds
- Algebraic properties
- (point-wise) Equivalence to other function
- No runtime errors

Unit vs Property based testing vs Verification (2)



(*) Graph not based on data, only indication

More certainty = more work

Generating constrained data

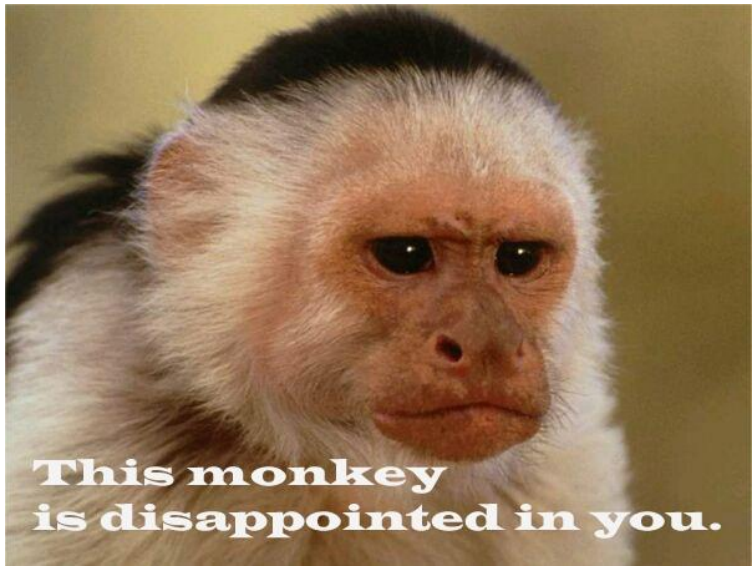
```
int binarySearch(int[] input, int elem)
```

Specification:
Requires: A sorted (non-decreasing) input array
Ensures: The output is -1 if elem does not occur in the input array,
or it is the lowest index 0 <= index < input.size such that input[index] == elem.

We need random *sorted* array

Generating constrained data

```
int[] randomSortedArray(int max, int maxsize) {
  int[] res;
  do {
     res = randomArray(max,maxSize);
  } while (!isSorted(res));
  return res;
}
```



Generating constrained data, smarter way

```
int[] randomSortedArray(int max, int maxsize) {
  int[] res;
  Random r = new Random();
  res = new int[r.next(maxSize)];
  int lowerBound = 0;
  for(int i = 0 ; i < res.length(); i++ ) {
   lowerBound = lowerBound + r.nextInt(max - lowerBound);
   res[i] = lowerBound;
  }
}
```

Algebraic properties

```
int[] merge(int[] left, int[] right)
```

Specification:
Requires: Left and right are non-null sorted in non-decreasing order
Ensures: Output is sorted in non-decreasing order and
forall x : merge(left,right) ++ left + right. count(x,merge(left,right)) == count(x,left) + count(x,right)
and the inputs are not modified

Property: forall left, right : sorted arrays.

isSorted(merge(left,right)) /\ occMap(merge(left,right)) == occMap(left).merge(occMap(right))

What info does it give

Let’s generate smaller numbers

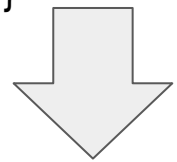
sort ({2560640248534041108,7988838958177884381,7580873405546173607, -3323190357898464145, -4926074542028600123, -6642162681233484839,8700679086630230326,4846869657323990532, -3031306809069141382, -6974789346487101705}) = {-6974789346487101705, -6642162681233484839, -4926074542028600123, -3323190357898464145, -3031306809069141382,2560640248534041108,4846869657323990532,7580873405546173607,7988838958177884381,8700679

sort ({67,74,3,4,80,89,74,82,40,7,51,68,49,94,69,35,43,51,67,13}) = {3,4,7,13,35,40,43,49,51,67,67,68,69,74,74,80,82,89,94}

This is not a *minimal* example!

From failing test case to minimal example

sort
({67,74,3,4,80,89,74,82,40,7,51,68,49,94,69,
35,43,51,67,13}) =
{3,4,7,13,35,40,43,49,51,67,67,68,69,74,74,
80,82,89,94}



sort({51,51}) = {51}

How do we do this?

Answer: Shrinking

This is debugging! Input minimalization!

- *Shrink* the failing input somehow
- See if the test still fails
- If so shrink more, otherwise backtrack

Many possible ways of shrinking!

For example: try all sub-lists

sort ({67,74,3,4,80,89,74,82,40,7,51,68,49,94,69, 35,43,51,67,13}) = {3,4,7,13,35,40,43,49,51,67,67,68,69,74,74, 80,82,89,94}

sort({51,51}) = {51}

Shrinking arrays: Greedy Binary search

```
  /* Requires : arr is a failing test case
   * Ensures : Res is a sublist of arr and res is a failing
               testcase
   */
static int[] shrinkHalfInsert(int[] arr){
    int[] left = leftHalf(arr);
    if(isFailing(left)){
     return shrinkHalfInsert(left);
    }
    int[] right = rightHalf(arr);
    if(isFailing(right)){
      return shrinkHalfInsert(right);
     }
    return arr;


}
```

Property based testing - Tool support

Quickcheck automates:

- Generating random stuff
- Shrinking
- Nice syntax for properties

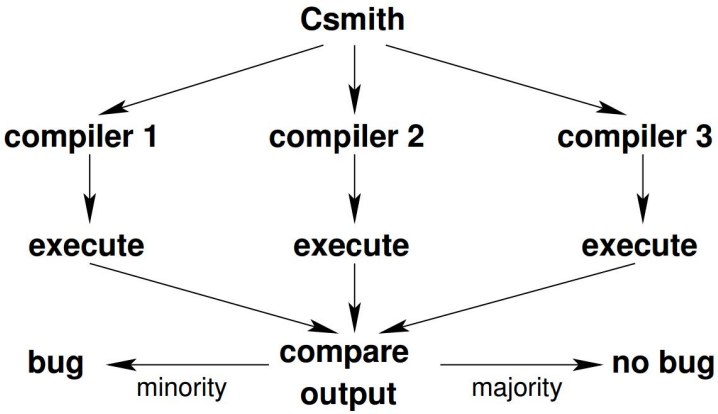
Originally for Haskell, but ports for:

C C++, Chicken Scheme,Clojure Common Lisp, D,Elm, Erlang, F#, Factor, Io,Java, JavaScript, Node.js, Objective-C, OCaml,Perl,Prolog, Python,R, Ruby, Rust, Scala, Scheme,Smalltalk,Standard ML, Swift

Randomized testing showcase: Csmith

Finding and Understanding Bugs in C Compilers (PLDI)

Generate random c-program with well-defined behavior (constrained data)
(according to specs, not everything specified)



GCC w O0, GCC w O1
LLVM ..

C compilers

Kahoot

GCC open tickets: 508
Clang open tickets: 3820

Still very trustworthy compared to most software.

Example bug

```
int foo (void) {
  signed char x = 1;
  unsigned char y = 255;
  return x > y;
}
```

Ubuntu GCC:
foo() == 0 (true),

Vanilla GCC and spec:
foo() == 1

CSmith - results

	GCC	LLVM
Front end	0	10
Middle end	49	75
Back end	17	74
Unclassified	13	43
Total	79	202

More C-compiler testing!

Compiler Validation via Equivalence Modulo Inputs

Take *existing unit test* (compilers have a lot of them), randomly transform program without changing semantics (i.e. outcome). For example:

- a + b -> b + a
- inline function
- remove dead code
- structs to variables and backwards

Run both through compiler and run.. same result?

Existing test

```
struct tiny { char c; char d; char e; };
f(int n, struct tiny x, struct tiny y, struct tiny z,
      long l) {
  if (x.c != 10) abort();
  if (x.d != 20) abort();
  if (x.e != 30) abort();
  if (y.c != 11) abort();
  if (y.d != 21) abort();
  if (y.e != 31) abort();
  if (z.c != 12) abort();
  if (z.d != 22) abort();
  if (z.e != 32) abort();
  if (l != 123) abort();
}
main() {
  struct tiny x[3];
  x[0].c = 10;
  x[1].c = 11;
  x[2].c = 12;
  x[0].d = 20;
  x[1].d = 21;
  x[2].d = 22;
  x[0].e = 30;
  x[1].e = 31;
  x[2].e = 32;
  f(3, x[0], x[1], x[2], (long)123);
  exit(0);
```

Semantically the same, but different result on Clang (LLVM)

```
struct tiny { char c; char d; char e; };
f(int n, struct tiny x, struct tiny y, struct tiny z,
      long l) {
  if (x.c != 10) /* deleted */;
  if (x.d != 20) abort();
  if (x.e != 30) /* deleted */;
  if (y.c != 11) abort();
  if (y.d != 21) abort();
  if (y.e != 31) /* deleted */;
  if (z.c != 12) abort();
  if (z.d != 22) /* deleted */;
  if (z.e != 32) abort();
  if (l != 123) /* deleted */;
}
main() {
  struct tiny x[3];
  x[0].c = 10;
  x[1].c = 11;
  x[2].c = 12;
  x[0].d = 20;
  x[1].d = 21;
  x[2].d = 22;
  x[0].e = 30;
  x[1].e = 31;
  x[2].e = 32;
  f(3, x[0], x[1], x[2], (long)123);
  exit(0);
```

Randomly mutate unit tests - results

	GCC	LLVM	TOTAL
Reported	111	84	195
Marked duplicate	28	7	35
Confirmed	79	68	147
Fixed	56	54	110

Serious testing - SQLite

SQLLite has some serious testing

- regular code: 112.8 KSLOC of C
- test code: 91555.1 KSLOC (811 x as much)
- 100% Statement, branch and MCDC coverage!
- Feed functions random input (fuzzing) to detect unexpected assertion errors
- Simulate I/O and memory errors
- Check consistency with and without optimizations

More property based testing later!

More shrinking next time!

Unit testing and property based good for testing *pure* code

What about stateful code, such as most Java objects?

Tune in next time!