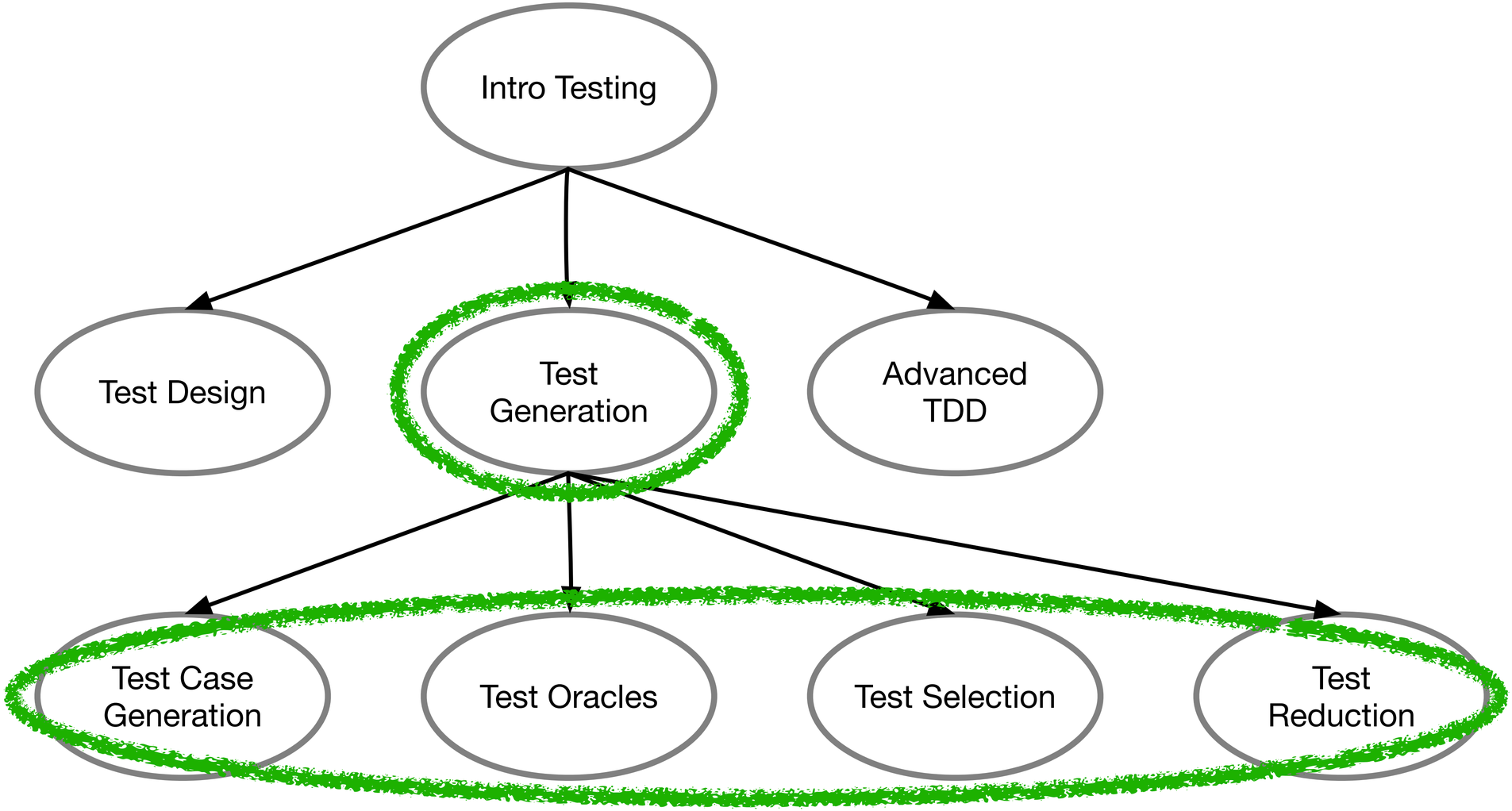
A Glympse of Testing Research

Guille Polito — CRIStAL — CNRS — RMoD - Inria Lille
@guillep

Test Generation



The problem of tests

- Do we have too few?
- Do we have too many?
- Are they redundant?
- Do they take long to run?
- Are they complex to debug?
- Are they good tests? Do they catch bugs?

The problem of tests

- **Do we have too few?**
- Do we have too many?
- Are they redundant?
- Do they take long to run?
- Are they complex to debug?
- Are they good tests? Do they catch bugs?

Automatic Test Generation

- How do we automatically generate tests?
  - Cover different code regions/branches/paths
  - Represent production code
- How do we ensure a generated test runs ok?
  - Discern between success, failure, error, crashes…

Automatic Test Generation

- **How do we automatically generate tests?**
  - Cover different code regions/branches/paths
  - Represent production code
- How do we ensure a generated test runs ok?
  - Discern between success, failure, error, crashes…

Automatic Test Generation

- When?
  - we have too few tests
  - always, software is complex, we cannot think about all possibilities
- Two main approaches
  - 1. Generate program inputs (requests, invocations…)
  - 2. Generate code (example programs, tests)

Fuzzing / Fuzz testing

- Goal: generate stimulus
- Try to execute different cases

```
SetTest >> testAdd

  | aSet |
  "Context"
  aSet := Set new.

  "Stimulus"
  aSet add: 5.
  aSet add: 5.

  "Check"
  self assert: aSet size equals: 1.
```

Fuzzing / Fuzz testing

- Goal: generate stimulus
- Try to execute different cases

- Example 1:
  - generate lots of cases

SetTest >> testAdd

```
| aSet |
"Context"
aSet := Set new.

"Stimulus"
1 to: 100 do: [:i |
  aSet add: i.
  aSet add: i. ]

"Check"
self assert: aSet size equals: 1.
```

Fuzzing / Fuzz testing

- Goal: generate stimulus
- Try to execute different cases

- Example 1:
  - generate lots of cases
- Example 2:
  - randomise

SetTest >> testAdd

```
| aSet |
1 to: 100 do: [:i | | i1 i2 |
 aSet := Set new.
 i1 := 100 atRandom.
 i2 := 100 atRandom.

 "Stimulus"
 aSet add: i1.
 aSet add: i2.

 "Check"
 self assert: (i1 = i2 and: [ aSet size = 1 ])
       or: [ aSet size = 2 ]
]
```

Property-based Testing

Example: Haskell's QuickCheck

- Developers define function *properties*
- Example, we say inserting into a sorted list should keep it sorted

```
InsertIsSorted x xs = ordered xs => ordered (insert x xs)
```

- ***Generators* build random input values** and validate that properties hold
- Pros: Lightweight and simple solution
- Cons: Expensive to explore border cases

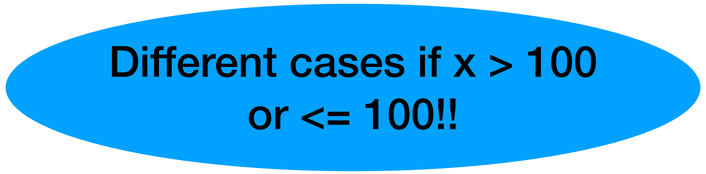
Claessen Koen, Hughes John. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. ICFP '00

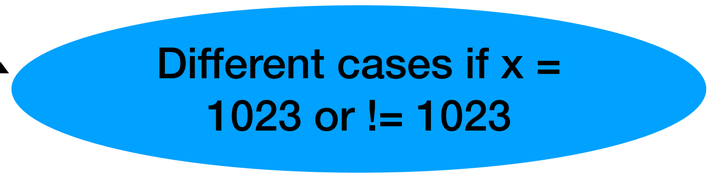
Directed Random Test Generation

Example: DART, CUTE

- White-box approach: guide test generation by looking at the implementation

```
int f(int x, int y){
  if (x > 100){
    if (y == 1023){
      segfault(!!)
    }
  }
}
```





Godefroid et al. DART: Directed Automated Random Testing. PLDI' 05
Set et al. CUTE: a concolic unit testing engine for C. FSE'05

Automatic Test Generation

- How do we automatically generate tests?
  - Cover different code regions/branches/paths
  - Represent production code
- **How do we ensure a generated test runs ok?**
  - Discern between success, failure, error, crashes…

The Test Oracle Problem

- Goal: generate assertions
- Much more complex than generating random inputs!!

SetTest >> testAdd

```
| aSet |
1 to: 100 do: [:i | | i1 i2 |
 aSet := Set new.
 i1 := 100 atRandom.
 i2 := 100 atRandom.

 "Stimulus"
 aSet add: i1.
 aSet add: i2.

 "Check"
 self assert: (i1 = i2 and: [ aSet size = 1 ])
       or: [ aSet size = 2 ]
```

Property-based Testing Oracles

Example: Haskell's QuickCheck

- **Developers define function *properties***
- Example, we say inserting into a sorted list should keep it sorted

```
InsertIsSorted x xs = ordered xs => ordered (insert x xs)
```

- *Generators* build random input values and validate that properties hold
- Pros: Lightweight and simple solution
- Cons: Expensive to explore border cases

Claessen Koen, Hughes John. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. ICFP '00

Metamorphic Testing

- Functions can be tested as a blackbox if we understand their some underlying properties, called *metamorphic properties,* such that
  - if we change the function input, we can foresee how the output will change

- Examples:
  - if we know that *sort(xs)* is sorted, then *sort(subset(xs))* is sorted
  - sin(*π*-*x*) = sin(x)

T.Y. Chen; S.C. Cheung; S.M. Yiu. Metamorphic testing: A new approach for generating next test cases. 98’

Differential Testing

- Two systems that implement the same semantics can be used as oracles for each other

- Examples:
  - two compilers for the same language
  - two parsers/serialisers for the same format
  - two databases…

W. McKeeman. Differential testing for software. Digital Technical Journal, ‘98

The problem of tests

- Do we have too few?
- **Do we have too many?**
- **Are they redundant?**
- **Do they take long to run?**
- Are they complex to debug?
- Are they good tests? Do they catch bugs?

Test Case Minimisation and Prioritisation

When we have **too many** tests

- Test Minimisation: How do we discover and eliminate redundant tests?
- Test Selection: How do we choose a subset of relevant tests to run?
- Test Prioritisation: What is the ideal order of running tests?

Yoo Shin, Harman Mark. Regression Testing Minimisation, Selection and Prioritisation: a Survey. STVR'07

Test Case Selection in Industry

- Case Study in WorldLine
- Large test suite that takes hours
- Static approaches: build an application model and find dependencies
- Dynamic approaches: execute the tests and find runtime dependencies

- Dynamic approaches are more accurate than static ones
  - Polymorphism, dynamic binding and reflection harm dependency analysis

Blondeau et al. Test Case Selection in Industry: an Analysis of Issues Related to Static Approaches. Software Qual J **25** 2016

The problem of tests

- Do we have too few?
- Do we have too many?
- Are they redundant?
- **Do they take long to run?**
- **Are they complex to debug?**
- Are they good tests? Do they catch bugs?

Test Case Reduction

- Make a test **smaller** so that
  - it tests the *same feature/scenario* as the larger version
  - yet, it’s
    - simpler to debug
    - faster to run
    - has less dependencies
    - …

Delta Debugging

- Automatically reduce test input, until it does not have the same output
- Example, reduce the array that holds `sort.first.(==1)`
  - **`first(sort([5,4,1,2,3])) = 1`**
  - **`first(sort([1,2,3])) = 1`**
  - `first(sort([2,3])) = 2`
  - **`first(sort([1,2])) = 1`**
  - `first(sort([2])) = 2`

A. Zeller, R. Hildebrandt, Simplifying and isolating failure-inducing input. IEEE TSE, 2002

The problem of tests

- Do we have too few?
- Do we have too many?
- Are they redundant?
- Do they take long to run?
- Are they complex to debug?
- **Are they good tests? Do they catch bugs?**

Test Case Validation

Who watches the watchmen?

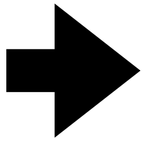
- Tests must detect bugs
  - validate results
  - validate state modifications
  - validate exception cases

- How can we detect *Weak* tests?

Mutation Testing

- Insert code modifications
- Tests should break!
- Otherwise, the modified functionality is not tested

```
if (a && b){
  c = 1;
} else {
  c = 17;
}
```



```
if (a || b){
  c = 1;
} else {
  c = 17;
}
```

Tim A. Budd, Mutation Analysis of Program Test Data. PhD thesis, Yale University New Haven CT, 1980

Rotten Green Tests

- Tests may have assertions, be green
  - And still not execute the assertion!
- Otherwise, the modified functionality is not tested
- Conditional code not executing a branch
- Iterating over an empty collection

```
class RottenTest {
    method testABC {
     if (false) then {self.assert(x)}
    }

}
```

Delplanque et al., Rotten Green Tests. ICSE'19

Conclusion

- Ensuring the validity and conformity of software systems is
  - complex
  - an active area of research
  - an interesting area of research

- Solutions often mix automatic code modification, static and dynamic analyses. They can be used in different contexts such as standard industrial setups or compilers