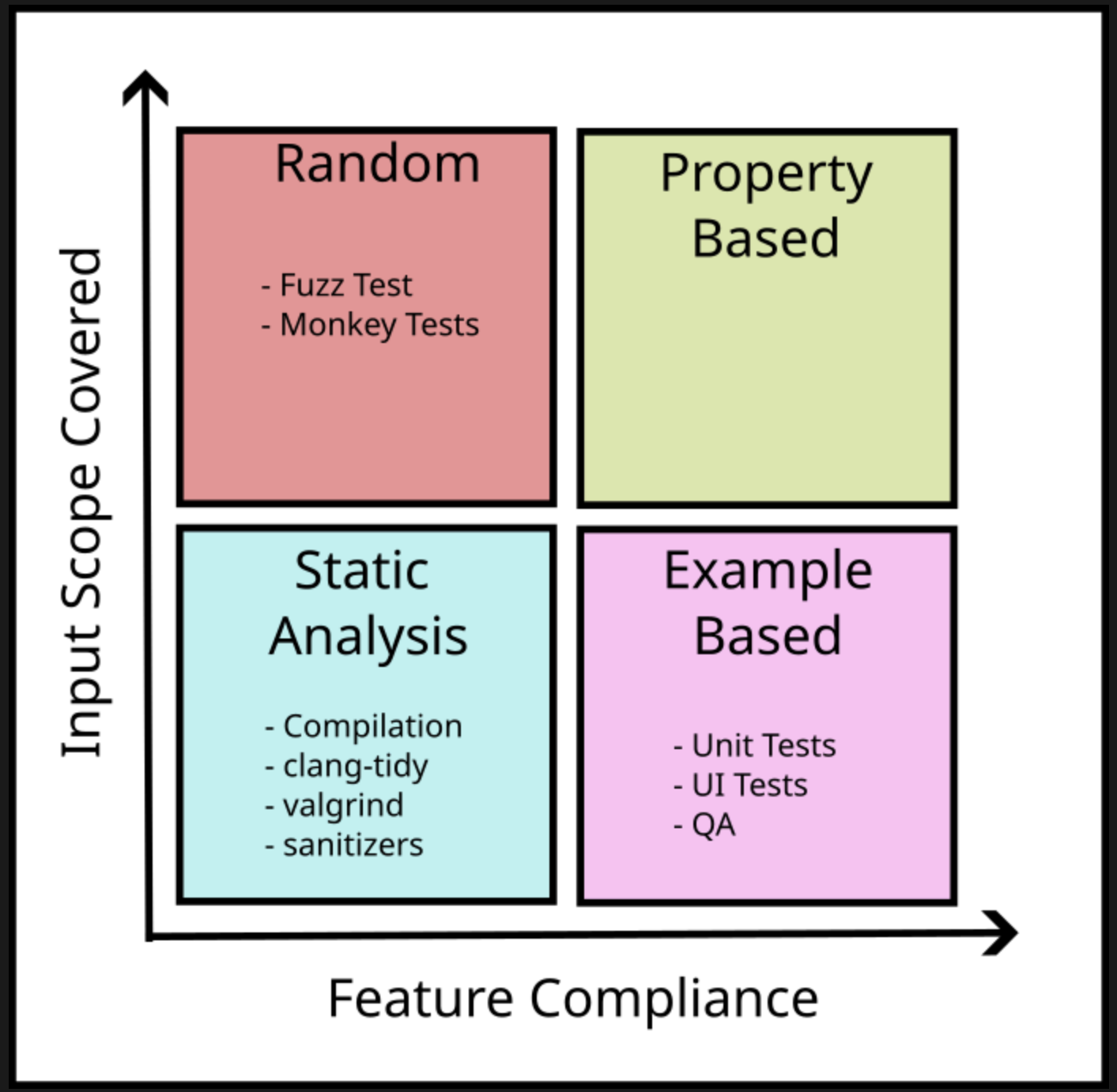
Property Based Testing

2025-06-17

How to Prove Correctness of the Code?


Random
- Fuzz Test
- Monkey Tests
Property Based
Static Analysis
- Compilation
- clang-tidy
- valgrind
- sanitizers
Example Based
- Unit Tests
- UI Tests
- QA
Input Scope Covered
Feature Compliance

Existing Issues I

- Static Analysis is for coding standards/compliance
- Unaware of domain and use cases
- Prone to false positives/false negatives
- Unable to detect runtime issues

Existing Issues II

- Random Tests are often hard to set up
- Time consuming to run
- Unable to catch logic flaws

Existing Issues III

- Unit tests only test the inputs we create
- Limited Code coverage
- Often focuses on implementation details (brittle)

Property Based Testing

> PBT is a testing method that verifies general properties or invariants by utilizing *randomly* generated input data.

Benefits of PBT

- Encourages thinking about code invariants
- Discovers logic flaws
- Cover the scope of all possible inputs
- Smart about selecting values
- Give minimal failing example (shrink)
- Deterministic

History Lesson

- First mentioned around 1990
- Popular Library: QuickCheck (Haskell) around 2000
- Growing acceptance for various languages/frameworks

Rapidcheck I

> QuickCheck clone for C++ with the goal of being simple to use with as little boilerplate as possible.

Rapidcheck II

- Integration with gtest, catch2 and others
- Support for (most) STL types
- Generators for custom types
- Shrinking
- Stateful testing
- Easy to set up

Example 1

String Concatenation

Finding Properties

Finding properties is the most difficult part

What is a Property?

- A logical assertion that remains true for all inputs
- Focus on general behavior, not specific input/output

Cool Sort

```
template<typename... Args>
void cool_sort(Args &&... args) {
    std::ranges::sort(std::forward<Args>(args)...);
}
```

What are possible properties?

Example 2

Sort Function

Shrinking

- *Randomly* generated input can be complex
- A failing test is often hard to interpret
- Shrinking provides the minimal failing example

Example 3

Shrinking

Generators for Fundamental Types

- Often Fundamental types are not enough
- Limit range

```
auto const i = *rc::gen::inRange(0, 10);
```

Generators for Custom Types

```
namespace rc {
    template<>
    struct Arbitrary<Vec2i> {
        static Gen<Vec2i> arbitrary() {
            return gen::build<Vec2i>(
                // clang-format off
                gen::set(&Vec2i::x),
                gen::set(&Vec2i::y)
                // clang-format on
            );
        }
    };
}
```

Example 4

Vec2i (part 1)

Printing Custom Types

```
void showValue(CustomType const &v, std::ostream &os) {
        os << ... << std::endl;
}
```

Example 4

Finding Bugs - Vec2i (part 2)

Advanced Features I

- Tagging:
  - `RC_TAG`
  - `RC_CLASSIFY`
- Preconditions
  - `RC_PRE`

Example 5

Advanced

Advanced Features II

Configuration

- `seed` - random seed used for generating values
- `max_success` and `max_size` - Tweak the number of runs
- `noshrink` and `verbose_shrinking` - configure shrinking

Summary

- No replacement for Unit Tests, but a great addition
- Many existing unit tests can benefit from using generated input

References

- An introduction to property based testing | F# for fun and profit