RapidCheck

Property based testing for C++

But what is property based testing?

Unit testing

```
TEST_CASE(”concatenates two strings") {
  const auto s = concat(“foo”, “bar”);
  REQUIRE(s == “foobar”);
}
```

Unit testing

```
1 of 1 tests passed.
```

Unit testing

```
TEST_CASE(”given ‘foo’ and ‘bar’,”
          ” yields ‘foobar’") {
  const auto s = concat(“foo”, “bar”);
  REQUIRE(s == “foobar”);
}
```

Properties of `concat`

- Given input strings a and b returning c:
  - c starts with a
  - c ends with b
  - `c.size() == a.size() + b.size()`

A property

```
c.size() == a.size() + b.size()
```

Property as a function

```
bool property(const std::string &a,
              const std::string &b) {
  const auto c = concat(a, b);
  return c.size() == a.size() + b.size()
}
```

How do we convince ourselves?

- Just try random stuff!
- Middle ground between exhaustive and “as many as I can stand to write”
- Yes, it really works

QuickCheck

QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs, Koen Claessen and John Hughes, ICFP 2000

QuickCheck

```
prop_concatsize a b =
   length (concat a b) == length a + length b
```

So I created RapidCheck

- Basic concepts more or less stolen from Haskell/Erlang QC (credits to Hughes and Claessen)
- Very low on boilerplate
- Fully-featured
  - Lots of generators/combinators
  - Test case shrinking
  - Stateful testing framework

The property

```
bool property(const std::string &a,
              const std::string &b) {
  const auto c = concat(a, b);
  return c.size() == a.size() + b.size()
}
```

RapidCheck

```
rc::check(&property);
```

RapidCheck

```
rc::check([](const std::string &a,
             const std::string &b) {
  const auto c = concat(a, b);
  return c.size() == a.size() + b.size()
});
```

RapidCheck

```
Falsifiable after 21 tests and 17 shrinks

std::tuple<std::string, std::string>:
("", "aaaaaaaaaaaaaaaa")
```

Shrinking

```
[0, 234, 34, 3436, 56, 45, 234,
456, 56, 345, 345, 56, 23, 3, 3,
56, 7567, 567, 345, 57, 23, 1, 0,
8, 9, 56, 345, 576, 345, 678,
345, 67, 345, 645, 234, 24, 678,
234, 0, 1, 23, 345, 5767, 34, 23,
5, 78, 3, 2, 8, 12, 34, 56, 123,
78, 90, 56, 0, 0, 23, 6, 56, 123,
3490, 45, 77567, 345, 234, 56, 3]
```

Shrinking

```
[0, 234, 34, 3436, 56, 45, 234,
456, 56, 345, 345, 56, 23, 3, 3,
56, 7567, 567, 345, 57, 23, 1, 0,
8, 9, 56, 345, 576, 345, 678,
345, 67, 345, 645, 234, 24, 678,
234, 0, 1, 23, 345, 5767, 34, 23,
5, 78, 3, 2, 8, 12, 34, 56, 123,
78, 90, 56, 0, 0, 23, 6, 56, 123,
3490, 45, 77567, 345, 234, 56, 3]
```

Shrinking

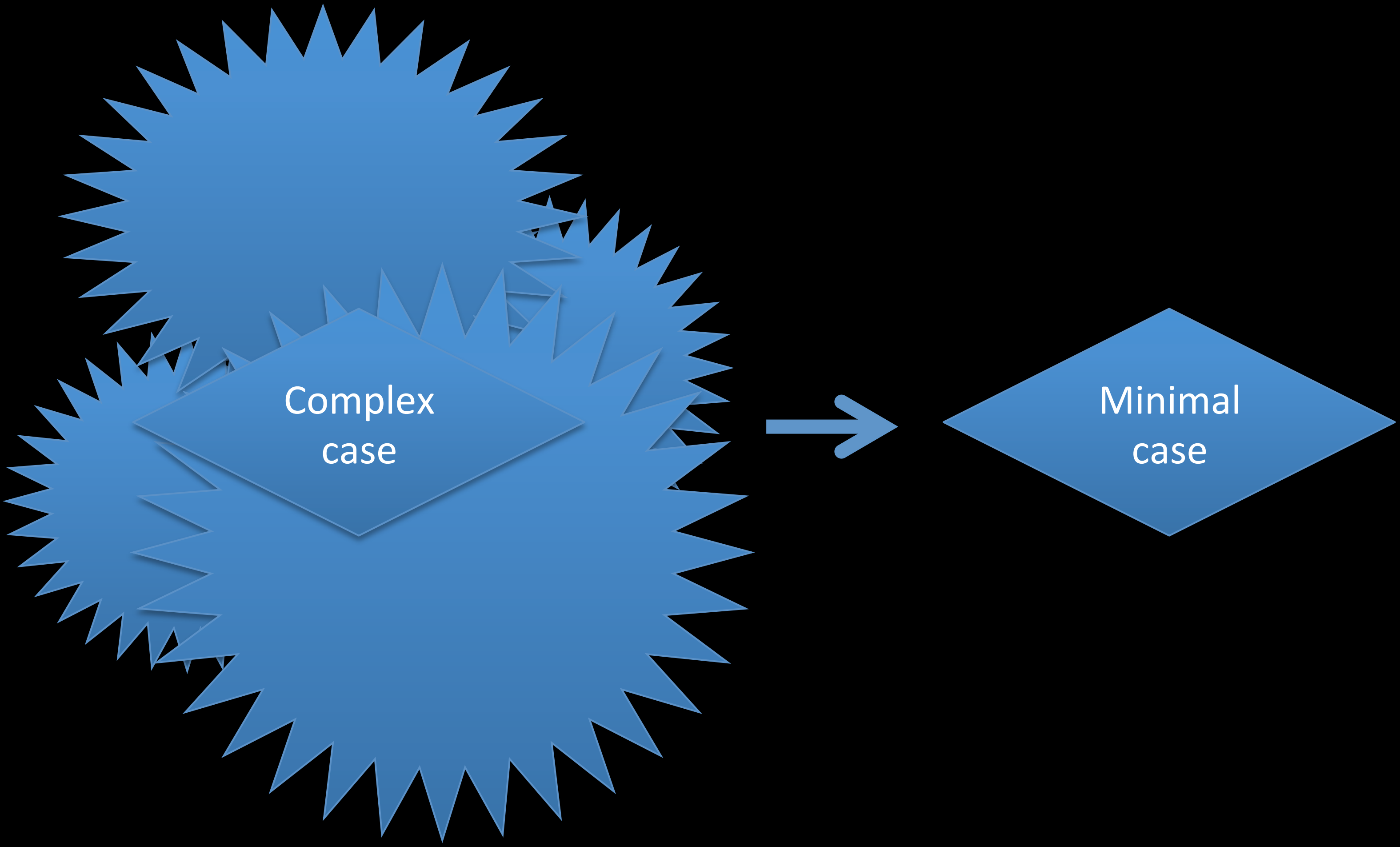
[77567]

Shrinking

```
[65536]
```

Oddly specific…

Shrinking



Advantages of property based testing

- It can find bugs in places you didn't even consider – not biased
- More coverage for less code
- Minimal counterexamples point you toward the bug
- Encourages you to think about what code *should* be doing, not what it *does*

But how is data formed?

Generators

- All generated data in RapidCheck comes from generator
- The customization point for supporting your own types

Supported out of the box

- All primitive types
- std::array<T, N>
- std::vector<T>
- std::deque<T>
- std::forward_list<T>
- std::list<T>
- std::set<T>
- std::map<K, V>
- std::multiset<T>
- std::multimap<K, V>
- std::unordered_set<T>
- std::unordered_map<K, V>
- std::unordered_multiset<T>
- std::unordered_multimap<T>
- std::basic_string<T>
- std::pair<T1, T2>
- std::tuple<Ts...>
- std::chrono::time_point
- std::chrono::duration
- boost::optional<T>

Create and combine

- `gen::arbitrary`
- `gen::construct`
- `gen::makeUnique`
- `gen::makeShared`
- `gen::build`
- `gen::container`
- `gen::just`
- `gen::lazy`
- `gen::distinctFrom`
- `gen::exec`
- `gen::maybe`
- `gen::inRange`
- `gen::nonZero`
- `gen::positive`
- `gen::negative`
- `gen::nonNegative`
- `gen::element`
- `gen::elementOf`
- `gen::weightedElement`
- `gen::sizedElementOf`
- `gen::sizedElement`
- `gen::oneOf`
- `gen::weightedOneOf`
- `gen::sizedOneOf`
- `gen::character`
- `gen::string`
- `gen::map`
- `gen::join`
- `gen::suchThat`
- `gen::cast`
- `gen::resize`
- `gen::scale`
- `gen::noShrink`
- `gen::withSize`
- `gen::tuple`
- `gen::pair`
- `gen::unique`
- `gen::uniqueBy`

Positive integers

```
using namespace rc;
const auto myGen = gen::positive<int>();
```

Vector of positive integers

```
const auto myGen =
    gen::container<std::vector<int>>(
        gen::positive<int>());
```

…but only with even length

```
using namespace rc;
const auto myGen =
    gen::suchThat(
        gen::container<std::vector<int>>(
            gen::positive<int>()),
        [](const auto &v) {
          return (v.size() % 2) == 0;
        });
```

…joined as a string

```
using namespace rc;
const auto myStringGen = gen::map(
    myGen, [](const auto &v) {
      return joinElements(v, “, “);
    });
```

Stateful testing

- What if the code is not a pure function?
- Input becomes a sequence of operations
- Validate against model

More at:
labs.spotify.com/2015/06/25/rapid-check/

That’s what we did

```
Using configuration: seed=11317088442510877731
Falsifiable after 76 tests and 30 shrinks

std::vector<std::shared_ptr<const Command<spotify::player::PlayerModel, spotify::player::PlayerSystem>>>:
ToggleRepeatingContext()
PreparePlay({
  "tracks": [{
      "uid": "a",
      "uri": "spotify:track:aaaaaaaaaaaaaaaaaaaaaa"
  }]
})
Play(0)
AddTrack(0, 0, {
  "uid": "b",
  "uri": "spotify:track:aaaaaaaaaaaaaaaaaaaaaa"
})
SkipToNextTrack()

../spotify/player/cpp/properties/main.cpp:94:
RC_ASSERT(track == expected_track)

Expands to:
{
  "uid": "a",
  "uri": "spotify:track:aaaaaaaaaaaaaaaaaaaaaa"
} == {
  "uid": "b",
  "uri": "spotify:track:aaaaaaaaaaaaaaaaaaaaaa"
}
```

That’s what we did

```
ToggleRepeatingContext()

PreparePlay({
  "tracks": [{
      "uid": "a",
      "uri": "spotify:track:aaaaaaaaaaaaaaaaaaaa"
  }]
})

Play(0)

AddTrack(0, 0, {
  "uid": "b",
  "uri": "spotify:track:aaaaaaaaaaaaaaaaaaaa"
})

SkipToNextTrack()
```

That’s what we did

```
RC_ASSERT(track == expected_track)

Expands to:
{
  "uid": "a",
  "uri": "spotify:track:aaaaaaaaaaaaaaaaaaaa"
} == {
  "uid": "b",
  "uri": "spotify:track:aaaaaaaaaaaaaaaaaaaa"
}
```

Learnings at Spotify

- High coverage with little code
- Good for testing very tricky things
  - Makes it at all practical to test some very tricky things
- Makes you really think about how things are supposed to work
- Found some very surprising bugs in existing code

Thanks!

GitHub:

github.com/emil-e/rapidcheck

Blog:

labs.spotify.com/2015/06/25/rapid-check