Property based testing in C++

How to write 1000s of tests in one sitting?

Patryk Małek

`https://github.com/pmalek`

Let’s talk about tests

Suppose we have the following **add** function signature:

```
int add(int x, int y);
```

Let’s talk about tests

Suppose we have the following **add** function signature:

```cpp
int add(int x, int y);
```

How would you approach testing it?

Let’s talk about tests

Suppose we have the following **add** function signature:

```
int add(int x, int y);
```

How would you approach testing it?

```
TEST(AddTest, OnePlus3Equals4){
  EXPECT_EQ(4, add(1,3));
}
```

Let’s talk about tests

Suppose we have the following **add** function signature:

```
int add(int x, int y);
```

How would you approach testing it?

```
TEST(AddTest, OnePlus3Equals4){
  EXPECT_EQ(4, add(1,3));
}
```

```
TEST(AddTest, OnePlus0Equals1){
  EXPECT_EQ(1, add(1,0));
}
```

Let’s talk about tests

Suppose we have the following **add** function signature:

```
int add(int x, int y);
```

How would you approach testing it?

```
TEST(AddTest, OnePlus3Equals4){
  EXPECT_EQ(4, add(1,3));
}
```

```
TEST(AddTest, OnePlus0Equals1){
  EXPECT_EQ(1, add(1,0));
}
```

```
TEST(AddTest, OnePlusMinus1Equals0){
  EXPECT_EQ(0, add(1,-1));
}
```

Let’s talk about tests

Suppose we have the following **add** function signature:

```
int add(int x, int y);
```

How would you approach testing it?

```
TEST(AddTest, OnePlus3Equals4){
  EXPECT_EQ(4, add(1,3));
}
```

```
TEST(AddTest, OnePlus0Equals1){
  EXPECT_EQ(1, add(1,0));
}
```

```
TEST(AddTest, OnePlusMinus1Equals0){
  EXPECT_EQ(0, add(1,-1));
}
```

```
TEST(AddTest, BigNumbersAreCorrectlyAdded){
  EXPECT_EQ(186321, add(87556,98765));
}
```

Let’s talk about tests

What if the implementation looked like...

Let’s talk about tests

What if the implementation looked like…

… this

```
int add(int x, int y){
  if( x == 7 ) return 1;
  return x + y;
}
```

Let’s talk about tests

What if the implementation looked like…

… this

```
int add(int x, int y){
  if( x == 7 ) return 1;
  return x + y;
}
```

ehh …

How do you define a good test?

How can we test addition?

How do you define a good test?

How can we test addition?

Let’s think about its properties

How do you define a good test?

How can we test addition?

Let’s think about its properties

- Commutativity:

```
Test: "result shouldn't depend on order of parameters"
    int x = random int
    int y = random int
    ASSERT(add(x,y) == add(y,x))
```

How do you define a good test?

How can we test addition?

Let’s think about its properties

- Commutativity:

```
Test: "result shouldn't depend on order of parameters"
    int x = random int
    int y = random int
    ASSERT(add(x,y) == add(y,x))
```

- Identity:

```
Test: "adding 0 to any number yields the same number"
    int x = random int
    ASSERT(add(x,0) == x)
```

How do you define a good test?

How can we test addition?

Let’s think about its properties

- Commutativity:

```
Test: "result shouldn't depend on order of parameters"
    int x = random int
    int y = random int
    ASSERT(add(x,y) == add(y,x))
```

- Identity:

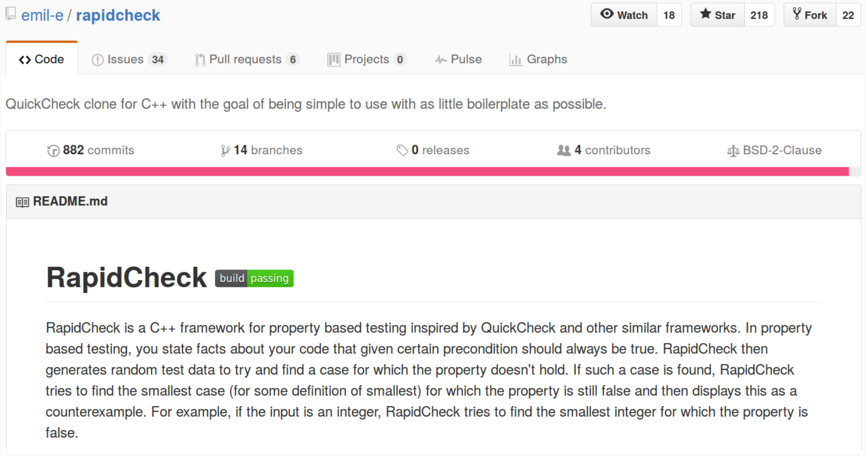
```
Test: "adding 0 to any number yields the same number"
    int x = random int
    ASSERT(add(x,0) == x)
```

- Associativity:

```
Test: "result shouldn't depend on order of operations"
    int x = random int
    int y = random int
    int z = random int
    ASSERT(add(z,add(x,y)) == add(add(x,y),z))
```

RapidCheck

`https://github.com/emil-e/rapidcheck`


emil-e / **rapidcheck**

Watch 18 Star 218 Fork 22

Code Issues 34 Pull requests 6 Projects 0 Pulse Graphs

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

882 commits **14** branches **0** releases **4** contributors BSD-2-Clause

README.md

RapidCheck

build passing

RapidCheck is a C++ framework for property based testing inspired by QuickCheck and other similar frameworks. In property based testing, you state facts about your code that given certain precondition should always be true. RapidCheck then generates random test data to try and find a case for which the property doesn't hold. If such a case is found, RapidCheck tries to find the smallest case (for some definition of smallest) for which the property is still false and then displays this as a counterexample. For example, if the input is an integer, RapidCheck tries to find the smallest integer for which the property is false.

RapidCheck addition tests

Let's use **RapidCheck** with our examples:

```
#include <rapidcheck.h>

int main() {
  rc::check("result shouldn't depend on order of parameters",
            [](int x, int y){
    RC_ASSERT( add(x,y) == add(y,x) );
  });

  rc::check("adding 0 to any number yields the same number",
            [](int x){
    RC_ASSERT( add(x,0) == x);
  });

  rc::check("result shouldn't depend on order of operations",
            [](int x, int y, int z){
    RC_ASSERT( add(z, add(x,y)) == add(add(x,y), z));
  });
}
```

RapidCheck output

Output we might expect on failure:

```
Using configuration: seed=1313473344045799863

- result shouldn't depend on order of parameters
Falsifiable after 18 tests and 1 shrink

std::tuple<int, int>:
(7, 0)

/home/patryk/workspace/git/rapidcheck_cmaketest/src/main.cpp:43:
RC_ASSERT(add(x,y) == add(y,x))

Expands to:
1 == 7

- adding 0 to any number yields the same number
Falsifiable after 18 tests

std::tuple<int>:
(7)

/home/patryk/workspace/git/rapidcheck_cmaketest/src/main.cpp:48:
RC_ASSERT(add(x,0) == x)

Expands to:
1 == 7

...
```

RapidCheck output

After fixing our implementation like so:

```
int add(int x, int y){
  return x + y;
}
```

RapidCheck output

After fixing our implementation like so:

```cpp
int add(int x, int y){
  return x + y;
}
```

All the tests pass:

```
Using configuration: seed=1912891779374620633

- result shouldn't depend on order of parameters
OK, passed 100 tests

- adding 0 to any number yields the same number
OK, passed 100 tests

- result shouldn't depend on order of operations
OK, passed 100 tests
```

RapidCheck configuration

`RapidCheck` has a lot of configuration options:

- `googletest`/`Boost.Test` integration

Google test integration

You can integrate RapidCheck with google test:

```
#include <gtest/gtest.h>
#include <rapidcheck.h>
#include <rapidcheck/gtest.h>

RC_GTEST_PROP(TestCase, inRange, (int first, int second))
{
  int x = *rc::gen::inRange(first, second);
  RC_ASSERT(x >= first);
  RC_ASSERT(x < second);
}

int main(int argc, char **argv)
{
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Google test integration

And you get familiar output:

```
[==========] Running 1 test from 1 test case.
[----------] Global test environment set—up.
[----------] 1 test from TestCase
[ RUN      ] TestCase.inRange
Using configuration: seed=4574822431460607532
[       OK ] TestCase.inRange (32 ms)
[----------] 1 test from TestCase (33 ms total)

[----------] Global test environment tear—down
[==========] 1 test from 1 test case ran. (33 ms total)
```

RapidCheck configuration

`RapidCheck` has a lot of configuration options:

- `googletest`/`Boost.Test` integration
- ”shrinking”

shrinking

Suppose we have the following output:

```
Using configuration: seed=1313473344045799863

- all numbers in vector have desired value
Falsifiable after 100 tests

std::vector<int>:
[-2319, 12, -223584, -2071, 4383, -3727, -7431, -123897]
```

shrinking

Suppose we have the following output:

```
Using configuration: seed=1313473344045799863

- all numbers in vector have desired value
Falsifiable after 100 tests

std::vector<int>:
[-2319, 12, -223584, -2071, 4383, -3727, -7431, -123897]
```

Is it clear what might be wrong with our implementation?

shrinking

Suppose we have the following output:

```
Using configuration: seed=1313473344045799863

- all numbers in vector have desired value
Falsifiable after 100 tests

std::vector<int>:
[-2319, 12, -223584, -2071, 4383, -3727, -7431, -123897]
```

Is it clear what might be wrong with our implementation?

How about now?

```
Using configuration: seed=1313473344045799863

- all numbers in vector have desired value
Falsifiable after 18 tests and 1 shrink

std::vector<int>:
[0, 0, 0, 0, 100, 0, 0, 0]
```

shrinking

Suppose we have the following output:

```
Using configuration: seed=1313473344045799863

- all numbers in vector have desired value
Falsifiable after 100 tests

std::vector<int>:
[-2319, 12, -223584, -2071, 4383, -3727, -7431, -123897]
```

Is it clear what might be wrong with our implementation?

How about now?

```
Using configuration: seed=1313473344045799863

- all numbers in vector have desired value
Falsifiable after 18 tests and 1 shrink

std::vector<int>:
[0, 0, 0, 0, 100, 0, 0, 0]
```

Implementation:

```
ASSERT(std::all_of(v.begin(), v.end(), [](int i){ return i<100; }));
```

RapidCheck configuration

`RapidCheck` has a lot of configuration options:

- `googletest`/`Boost.Test` integration
- ”shrinking”
- Reproducible failures/seed

Reproducible failures

Each time you get a failure with `RapidCheck` you'll get a similar information in the end of console output:

```
Some of your RapidCheck properties had failures. To reproduce these, run with:
RC_PARAMS="reproduce=C0SYkRWaudGIwACdvBSYulHIuVXbiVmcgkXalxGZzBCdoVGIzFWblBib11mYlJ3H+
35MMG+Aw_h_dODjhPA8f4fnzwY4DA_H+35MMG+AwDIIA0ADAAAAAEDchJXYtVGdlJ3cg8mckVmcgMGah52ZlBC
ZvV2cudCdgEmZmV2Y0BCdoVGIyV2c1xGdf4fnzwY4DA_H+35MMG+Aw_h_dODjhPA8f4fnzwY4DAPggAQDMAAAA
AA"
```

You can reproduce a failed test (with seed that was used to run it) by running your test binary with `RC_PARAMS` environment variable.

RapidCheck configuration

`RapidCheck` has a lot of configuration options:

- `googletest`/`Boost.Test` integration
- ”shrinking”
- Reproducible failures/seed
- Generators for user defined types

RapidCheck configuration

`RapidCheck` has a lot of configuration options:

- `googletest`/`Boost.Test` integration
- ”shrinking”
- Reproducible failures/seed
- Generators for user defined types
- Built in support for many STL types

RapidCheck configuration

`RapidCheck` has a lot of configuration options:

- `googletest`/`Boost.Test` integration
- ”shrinking”
- Reproducible failures/seed
- Generators for user defined types
- Built in support for many STL types
- Configurable number of tests to run

RapidCheck configuration

`RapidCheck` has a lot of configuration options:

- `googletest`/`Boost.Test` integration
- ”shrinking”
- Reproducible failures/seed
- Generators for user defined types
- Built in support for many STL types
- Configurable number of tests to run
- … and many more

Conclusion

Think about your systems under test as fellow human beings

Conclusion

Think about your systems under test as fellow human beings

Don’t think about them in terms of input and output pairs

Conclusion

Think about your systems under test as fellow human beings

Don’t think about them in terms of input and output pairs

Consider their properties and conditions that should hold

Examples source code

Examples available at:

`https://github.com/pmalek/rapidcheck_codedive.git`

References i

E. Eriksson.
Generating test cases so you don’t have to.
https://labs.spotify.com/2015/06/25/rapid-check/, 2015.

S. Wlaschin.
The lazy programmer’s guide to writing 1000’s of tests: An introduction to property based testing.
https://skillsmatter.com/skillscasts/6432-the-lazy-programmers-guide-to-writing-1000s-of-tests-an-introduction-to-property-based-testing, 2015.

Questions?