

---

# **MosquitoNet Documentation**

***Release 0.0.0***

**Simon Bourne**

December 06, 2015

## CONTENTS

|           |                                   |           |
|-----------|-----------------------------------|-----------|
| <b>1</b>  | <b>Introduction</b>               | <b>1</b>  |
| 1.1       | Features . . . . .                | 1         |
| 1.2       | Example . . . . .                 | 1         |
| <b>2</b>  | <b>Getting Started</b>            | <b>4</b>  |
| <b>3</b>  | <b>Assertions</b>                 | <b>5</b>  |
| 3.1       | Basic Assertions . . . . .        | 5         |
| 3.2       | Exceptions . . . . .              | 6         |
| 3.3       | Explicit Failures . . . . .       | 6         |
| <b>4</b>  | <b>BDD Tests</b>                  | <b>7</b>  |
| <b>5</b>  | <b>Parameterized Tests</b>        | <b>8</b>  |
| <b>6</b>  | <b>Model Checking</b>             | <b>9</b>  |
| <b>7</b>  | <b>Customizing Assertions</b>     | <b>10</b> |
| <b>8</b>  | <b>Pretty Printing Your Types</b> | <b>11</b> |
| <b>9</b>  | <b>Test Harness Command Line</b>  | <b>12</b> |
| <b>10</b> | <b>Custom Reporters</b>           | <b>13</b> |
| <b>11</b> | <b>Supported Compilers</b>        | <b>14</b> |
| <b>12</b> | <b>Source Code For Examples</b>   | <b>15</b> |

## INTRODUCTION

MosquitoNet is a C++14 unit testing and model checking framework for Linux, OS X and Windows.

### 1.1 Features

- Single header version makes it simple to get started.
- Only one macro, which only adds file and line arguments to a simple function call. This means you're always dealing with core C++ code, so you could generate tests programatically.
- Simple tests or BDD style tests.
- Parameterized tests.
- Model checking. Specify a model and the values each argument can have and MosquitoNet will check every combination of arguments against your model.
- Customizable assertions using natural C++ expressions.
- Tests can continue to run after the first failure.
- Nested test contexts.

### 1.2 Example

This example shows most of the features of MosquitoNet. For a gentler introduction, see [Getting Started](#).

```
#include "MosquitoNet.h"

#include <vector>
#include <set>

using namespace Enhedron::Test;
using std::vector;
using std::set;

// We'll use this later in some parameterized tests.
void checkVectorSize(Check& check, size_t size) {
    vector<int> v(size, 0);
    check(length(VAR(v)) == size);
}

static Suite u("Util",
    given("a very simple test", [] (auto& check) {
```

```

int a = 1;

// VAR is the only macro we need. If the name clashes, undef it
// and use M_ENHEDRON_VAR. Upon failure, this will log "a == 1",
// along with the value of `a`.
check(VAR(a) == 1);
}),

given("an empty set", [] (auto& check) {
    set<int> s;

    // Upon failure, this will log "length(s) == 1", along with
    // the contents of the set.
    check("it is initially empty", length(VAR(s)) == 0u);

    check.when("we add an element", [&] {
        s.insert(1);
        check("the size is 1", length(VAR(s)) == 1u);

        // This test will run twice. The first time, it will run the when
        // block labelled when("we add a different element"), but skip the
        // when block labelled when("we add the same element"). The second
        // time it runs, it will do the inverse. There can be an arbitrary
        // number of when blocks within each block, nested to an arbitrary
        // depth.
        check.when("we add a different element", [&] {
            s.insert(2);
            check("the size is 2", length(VAR(s)) == 2u);
        });

        check.when("we add the same element", [&] {
            s.insert(1);
            check("the size is still 1", length(VAR(s)) == 1u);
        });
    });
}),

// Parameterized tests. There can be any number or type of parameters.
given("a vector of size 0", checkVectorSize, 0),
given("a vector of size 10", checkVectorSize, 10),

// Model checking.
exhaustive(
    choice(0, 10, 20), // These are the 3 values for `initialSize`.
    choice(0, 5, 10, 15, 20, 25) // and these are the 6 for `resizeTo`.
    // This will run the test 3 * 6 = 18 times for every
    // combination of arguments.
).
given("a vector with some elements", [] (
    Check& check,
    size_t initialSize,
    size_t resizeTo
)
{
    vector<int> v(initialSize, 0);
    check("the initial size is correct",
        length(VAR(v)) == initialSize);
}

```

```
        check.when("we resize it", [&] {
            v.resize(resizeTo);
            check("the new size is correct",
                length(VAR(v)) == resizeTo);

            check("the size <= the capacity",
                length(VAR(v)) <= v.capacity());
        });
    },

    context("we can also nest contexts",
        context("to an arbitrary depth",
            given("an empty test to illustrate that tests can go here",
                [] (auto& check) {
            })
        )
    )
);
```

## GETTING STARTED

Download the single header latest release of the single header, *MosquitoNet.h* from [here](#). Everything you'll need is in the *Enhedron::Test* namespace. In the same directory as *MosquitoNet.h*, create a file *Harness.cpp* with this code in it:

```
#include "MosquitoNet.h"

int main(int argc, const char* argv[]) {
    return Enhedron::Test::run(argc, argv);
}
```

Then compile it with *g++* (version 5 or later, but 4.9 will work with *-std=c++1y*):

```
g++ --std=c++14 -o test-harness Harness.cpp
```

Now run *.test-harness* and you should get this output:

```
Totals: 0 tests, 0 checks, 0 fixtures
```

Let's add a simple test. We'll just check the value of a variable. In the same directory again, create a file *MinimalTest.cpp* so it contains:

```
static Suite u("a minimal test suite",
    given("a very simple test", [] (auto& check) {
        int a = 1;

        check(VAR(a) == 1);
    })
);
```

and compile it with:

```
g++ --std=c++14 -o test-harness MinimalTest.cpp Harness.cpp
```

From now on, we'll give examples as code snippets from [Source Code For Examples](#).

## ASSERTIONS

### 3.1 Basic Assertions

When your test is run, it is passed a *Check* object which is your interface to MosquitoNet within the test. *Check* overrides *operator()* to provide basic checking. We'll concentrate on that for now. The test will continue even if any checks fail. The *VAR* macro indicates that we're interested in the value and name of a particular variable. Each check will return a *bool* with *true* iff the check passes.

This example will check the values of *a* and *b*.

```
static Suite s("examples", context("assertion",
    given("some constants to assert with", [] (auto& check) {
        int a = 0;
        int b = 1;

        check (VAR(a) == 0 && VAR(b) == 1);
    })
));
```

When we run the test harness with *-verbosity variables* it will print out each check along with the value of the variables in that expression:

```
examples/assertion
  Given: some constants to assert with
  Then : ((a == 0) && (b == 1))
         a = 0: file "/work/build/MosquitoNet/cpp/test/src/Examples/AllExamples.cpp", line 16.
         b = 1: file "/work/build/MosquitoNet/cpp/test/src/Examples/AllExamples.cpp", line 16.
```

Assertions can also have a description:

```
check("a is one and b is two", VAR(a) == 0 && VAR(b) == 1);
```

Will output:

```
Then : a is one and b is two
      ((a == 0) && (b == 1))
         a = 0: file "/work/build/MosquitoNet/cpp/test/src/Examples/AllExamples.cpp", line 27.
         b = 1: file "/work/build/MosquitoNet/cpp/test/src/Examples/AllExamples.cpp", line 27.
```

For a less technical spec, we can suppress the expression (when a description is provided) and the variables with *-verbosity checks*:

```
examples/assertion
  Given: some constants to assert with
  Then : a is one and b is two
```

We can also provide context variables that are not checked, but provide context in the output:

```
check("`c` and `d` are provided for context", VAR(a) == 0 && VAR(b) == 1, VAR(c), VAR(d));
```

Outputs:

```
Then : `c` and `d` are provided for context
      ((a == 0) && (b == 1))
      a = 0: file "/work/build/MosquitoNet/cpp/test/src/Examples/AllExamples.cpp", line 33.
      b = 1: file "/work/build/MosquitoNet/cpp/test/src/Examples/AllExamples.cpp", line 33.
      c = 2: file "/work/build/MosquitoNet/cpp/test/src/Examples/AllExamples.cpp", line 33.
      d = 4: file "/work/build/MosquitoNet/cpp/test/src/Examples/AllExamples.cpp", line 33.
```

Also note that you can use an expression inside the VAR macro and all variables don't have to be inside the VAR macro:

```
check("a is one and b is two", VAR(a) == 0 && b == 1 && VAR(a + c) == c);
```

Outputs:

```
Then : a is one and b is two
      (((a == 0) && true) && (a + c == 2))
      a = 0: file "/work/build/MosquitoNet/cpp/test/src/Examples/AllExamples.cpp", line 37.
      a + c = 2: file "/work/build/MosquitoNet/cpp/test/src/Examples/AllExamples.cpp", line 37.
```

## 3.2 Exceptions

This will check something derived from *std::exception* is thrown:

```
check.throws("an exception is thrown", VAR(throwRuntimeError()));
```

This will check something derived from *runtime\_error* is thrown:

```
check. template throws<runtime_error>("a runtime_error is thrown", VAR(throwRuntimeError()));
```

Again, we can provide context variables:

```
check.throws("`a` and `b` are provided for context", VAR(throwRuntimeError()), VAR(a), VAR(b));
```

## 3.3 Explicit Failures

We can explicitly fail by throwing an exception. We can also use the *Check* object if we'd like to continue the test:

```
check.fail(VAR("explicit failure"));
```

And with context variables:

```
check.fail(VAR("`a` and `b` are provided for context"), VAR(a), VAR(b));
```



**BDD TESTS**

## PARAMETERIZED TESTS

**MODEL CHECKING**

## **CUSTOMIZING ASSERTIONS**

## PRETTY PRINTING YOUR TYPES

## TEST HARNESS COMMAND LINE

## CUSTOM REPORTERS

## **SUPPORTED COMPILERS**

- g++ version 4.9 or later.
- clang version 3.6 or later.
- Microsoft Visual C++ 2015 or later.



## SOURCE CODE FOR EXAMPLES

```
using std::runtime_error;

// The comments in this file are used when building the documentation.

// Assertion example 1:
static Suite s("examples", context("assertion",
    given("some constants to assert with", [] (auto& check) {
        int a = 0;
        int b = 1;

        check(VAR(a) == 0 && VAR(b) == 1);
    })
));
// Assertion example 1 end.

void throwRuntimeError() {
    throw runtime_error("Expected exception");
}

static Suite t("examples", context("assertion",
    given("some constants to assert with", [] (auto& check) {
        int a = 0;
        int b = 1;
        int c = 2;
        int d = 4;

        // Assertion example 2:
        check("a is one and b is two", VAR(a) == 0 && VAR(b) == 1);
        // Assertion example 2 end.

        // Assertion example 3:
        check("`c` and `d` are provided for context", VAR(a) == 0 && VAR(b) == 1, VAR(c), VAR(d));
        // Assertion example 3 end.

        // Assertion example 4:
        check("a is one and b is two", VAR(a) == 0 && b == 1 && VAR(a + c) == c);
        // Assertion example 4 end.

        // Assertion example 5:
        check.throws("an exception is thrown", VAR(throwRuntimeError)());
        // Assertion example 5 end.

        // Assertion example 6:
        check.template throws<runtime_error>("a runtime_error is thrown", VAR(throwRuntimeError)()));
```

```
// Assertion example 6 end.

// Assertion example 7:
check.throws("`a` and `b` are provided for context", VAR(throwRuntimeError()), VAR(a), VAR(b));
// Assertion example 7 end.

// Assertion example 8:
check.fail(VAR("explicit failure"));
// Assertion example 8 end.

// Assertion example 9:
check.fail(VAR("`a` and `b` are provided for context"), VAR(a), VAR(b));
// Assertion example 9 end.
    })
});
```