MosquitoNet Documentation

Release 0.0.0

Simon Bourne

December 06, 2015

CONTENTS

CHAPTER

ONE

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) {
            })
        )
    )
);
```

CHAPTER

TWO

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.

CHAPTER

THREE

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 *true* if the check passes, *false* otherwise'.

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));
```

3.4 Containers

MosquitoNet provides some utility functions for working with containers. For example, you can check the length of a container:

```
vector<int> v{1,2,3};
check("the length of a vector is 3", length(VAR(v)) == 3u);
```

and it will output:

```
Then : the length of a vector is 3
    (length(v) == 3)
        v = [1, 2, 3]: file "/work/build/MosquitoNet/cpp/test/src/Examples/AllExamples.cpp", line 96
```

Other functions are:

- *countEqual(container, value)* gives the number of elements equal to *value*.
- *countMatching(container, predicate)* gives the number of elements matching *predicate*.
- *allOf(container, predicate)* is true iff all elements match *predicate*.
- *anyOf(container, predicate)* is true iff any elements match *predicate*.
- *noneOf(container, predicate)* is true iff no elements match *predicate*.
- *length(container)* gives the length of *container*.
- *startsWith(container, prefixContainer)* is true iff *container* starts with *prefixContainer*.
- *endWith(container, postfixContainer)* is true iff *container* starts with *prefixContainer*.
- *contains(container, subSequenceContainer)* is true iff *subSequenceContainer* is a sub-sequence of *container*.

3.5 Customizing Assertions

You can customize assertions to use you own functions. If you have a non-overloaded function and you don't need template argument deduction, you can just use the function as-is in assertions. For example, given:

```
int multiply(int x, int y) { return x * y; }
```

multiply can be used directly in assertions:

```
check("3 squared is 9", VAR(multiply) (3, 3) == 9);
```

Arguments can be recorded as variables if required:

```
int three = 3;
check("3 squared is 9", VAR(multiply) (VAR(three), 3) == 9);
```

If you have an overloaded function or require template argument deduction, or just want a cleaner syntax, you can provide a wrapper for your function. For example, given:

```
template<typename Value>
Value multiplyOverloaded(Value x, Value y) {
    return x * y;
}

template<typename Value>
Value multiplyOverloaded(Value x, Value y, Value z) {
    return x * y * z;
}
```

You can provide the wrapper:

```
template<typename... Args>
auto multiplyOverloadedProxy(Args&&... args) {
    return makeFunction(
             "multiplyOverloaded",
             [] (auto&&... args) { return multiplyOverloaded(forward<decltype(args)>(args)...); }
    )(forward<Args>(args)...);
}
```

then the check:

```
check("3 cubed is 27", multiplyOverloadedProxy(VAR(three), 3, 3) == 27);
```

will output:

```
Then : 3 cubed is 27
     (multiplyOverloaded(three, 3, 3) == 27)
         three = 3: file "/work/build/MosquitoNet/cpp/test/src/Examples/AllExamples.cpp", line 104.
```

CHAPTER

FOUR

BDD TESTS

CHAPTER

FIVE

PARAMETERIZED TESTS

CHAPTER

SIX

MODEL CHECKING

CHAPTER

SEVEN

PRETTY PRINTING YOUR TYPES

CHAPTER

EIGHT

TEST HARNESS COMMAND LINE

CHAPTER

NINE

CUSTOM REPORTERS

CHAPTER

TEN

SUPPORTED COMPILERS

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

CHAPTER

ELEVEN

SOURCE CODE FOR EXAMPLES

```
using std::vector;
using std::runtime_error;
using std::forward;

// 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");
}

// Assertion example 11:
int multiply(int x, int y) { return x * y; }
// Assertion example 11 end.

// Assertion example 14:
template<typename Value>
Value multiplyOverloaded(Value x, Value y) {
    return x * y;
}

template<typename Value>
Value multiplyOverloaded(Value x, Value y, Value z) {
    return x * y * z;
}
// Assertion example 14 end.

// Assertion example 15:
template<typename... Args>
auto multiplyOverloadedProxy(Args&&... args) {
    return makeFunction(
            "multiplyOverloaded",
            [] (auto&&... args) { return multiplyOverloaded(forward<decltype(args)>(args)...); }
    )(forward<Args>(args)...);
```

```
}
// Assertion example 15 end.

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.

        // Assertion example 10:
        vector<int> v{1,2,3};
        check("the length of a vector is 3", length(VAR(v)) == 3u);
        // Assertion example 10 end.

        // Assertion example 12:
        check("3 squared is 9", VAR(multiply) (3, 3) == 9);
        // Assertion example 12 end.

        // Assertion example 13:
        int three = 3;
        check("3 squared is 9", VAR(multiply) (VAR(three), 3) == 9);
        // Assertion example 13 end.

        // Assertion example 16:
        check("3 cubed is 27", multiplyOverloadedProxy(VAR(three), 3, 3) == 27);
        // Assertion example 16 end.
    })
));
```