STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LV**, Number 1, 2010

UNIT TESTING OF C++ TEMPLATE METAPROGRAMS

ÁBEL SINKOVICS

Abstract. Unit testing, a method for verifying a piece of software, is a widely used technique and a good practice in software development. A unit, the smallest testable element of a software is verified in isolation. Tools for the development and execution of unit tests are available for a number of programming languages, such as C, C++, Java, C#, Python, Perl, Erlang, etc.

Unfortunately, the development of C++ template metaprograms has only limited support for this technique. To improve software quality, creation and maintenance of unit tests should be part of the development of C++ template metaprogramming libraries. In this paper we present how unit tests can be built and executed for C++ template metaprograms and how they fit into the development process of C++ libraries and applications. We present tools that help the developers in building unit tests for C++ template metaprograms. We also present how we applied them on a number of C++ template metaprogramming libraries.

1. Introduction

In 1994 Erwin Unruh demonstrated that the C++ compiler can execute algorithms by using templates in a special way. [1] This technique is called C++ template metaprogramming and it became a widely used technique. C++ template metaprograms form a Turing-complete sub-language of C++. [2] This capability was not taken into account during the development of the C++ standard, thus the syntax of template metaprograms is not friendly. It is hard to write, understand and maintain them, their development is a highly error-prone process. In many cases, C++ template metaprograms are written for a specific purpose, they are validated against that one special case and no further testing is done. As a result, hidden bugs may remain uncovered until the template metaprogram is reused in other situations or used in a different environment, such as another platform or compiler. Because of their complex

Received by the editors: 14th February 2011.

2010 *Mathematics Subject Classification.* code, code.

1998 *CR Categories and Descriptors.* code [**D.2.5**]: Testing and Debugging – *Testing tools*; code [**D.3.2**]: Language Classification – *C++*.

Key words and phrases. template metaprogramming, C++, testing, boost.

syntax, debugging template metaprograms [3] later, months or maybe years after they're written is a difficult thing which should be prevented with proper testing.

The rest of the paper is organised as follows. In section 2 we present our solution for the problem, in section 3 we evaluate it and compare it to the technique used in the Boost template metaprogramming library. We summarise our results in section 4.

2. Our approach

We have built a unit testing framework for C++ template metaprograms. In this section we present how our solution works, what the major difficulties are with unit testing metaprograms in general and how we have solved them. We have published our framework as an open-source project [5].

A C++ template metaprogram is executed at compile time. To test such a code, the test has to be evaluated at compile time as well. Template metaprogramming follows the functional paradigm, metaprograms consist of template metafunctions and the execution of a metaprogram is the evaluation of a metafunction, which invokes a number of other metafunctions. Thus, code testing template metaprograms has to be a template metafunction. The test is executed by evaluating the test template metafunction. Assume that we have a metafunction, `plus` that calculates the sum of two numbers. Here is the declaration of it:

```
template <class a, class b> struct plus;
```

We can write the following code that tests it:

```
typedef boost::mpl::equal_to<
  plus< boost::mpl::int_<11>, boost::mpl::int_<2> >::type,
  boost::mpl::int_<13>
> TestPlus;
```

`equal_to` is a metafunction that compares two values, `int_` is a wrapper for integer values. They are part of the Boost library [6]. `TestPlus` is a nullary metafunction, which is a metafunction taking zero arguments. It is evaluated when its nested class called `type` is accessed [7]. The metafunction evaluates `plus` with two arguments, 11 and 2, and verifies that it evaluates to 13. The value of `TestPlus` is a boolean value, the result of the test. When the test is successful, the value is `true`. When the expected and actual results differ, the value is `false`.

We have seen so far how test code for template metaprograms can be implemented. The metafunction implementing the test can evaluate to a boolean value, which is `true` when the test is successful and `false` when it isn't. The major difficulty is how the result can be represented in a structured way.

A template metaprogram is executed as part of the compilation process, thus the template metaprogram runs inside the compiler and the input and output possibilities of it are limited. The only source of input data is the code under compilation. There are more possibilities for output data. Template metaprograms are template metafunctions, that evaluate to a class. That class can be used by the rest of the code, its static members become part of the compiled code. When there are errors during the evaluation of template metaprograms, the compiler emits warnings or errors. The latter causes the compilation process to fail. As a summary, the input of a template metaprogram is the compiled source code, the outputs of a template metaprogram are the warning and error messages and the compiled code. Tests come from the source code. We present how the different output channels can be used to display the result of the test execution.

2.1. **Displaying results as error messages.** One of the output channels is the list of error and warning messages. Boost provides a macro for static assertion, `BOOST_STATIC_ASSERT`, that takes a compile time boolean value as argument and emits a compile error, when the value is `false`. We can use it to emit a compile error when a test fails. We need to evaluate the test first. It evaluates to a boolean value: `true` when the test was successful and `false`, when it wasn't. This value can be passed to `BOOST_STATIC_ASSERT`. When the test was not successful, the compilation fails.

The drawback of static assertion is that failed tests are displayed as error messages, that are complex and difficult to interpret. The compiler is not prepared for displaying an easily readable summary of which tests failed and which ones passed.

2.2. **Displaying results using the generated code.** The other output of template metaprograms is the generated code. Template metaprograms can generate executable code that becomes part of the compiled program and can be executed at runtime.

We propose an approach where the test driver executes the tests at compile time and generates runtime code, that displays the results in a human readable format. Template metaprograms are evaluated at compile time, thus test execution has to happen at compile time. Once tests are executed, the results can be displayed at any stage of the compilation and program execution process. We have very limited control over the way the C++ compiler displays information as error and warning messages, but we have full control over how runtime code displays output.

Code generation from template metaprograms happens by adding static member functions to classes implementing template metafunctions. We always call this static member function `run`. For example, here is a template metafunction that takes a wrapped boolean value as an argument and generates code that displays the word `passed` or `FAILED` on the standard output depending on the value of the compile time argument:

```
template <class b>
struct display_result
{
  static void run()
  {
    std::cout << (b::value ? "passed" : "FAILED");
  }
};
```

When `display_result` is instantiated, the compiler generates a function by substituting `b` with the actual template argument in the code of `run`. The generated `run` function can be executed at runtime. The result of tests can be displayed using it.

We can execute tests at compile time and display the results at runtime. But the code executing tests and displaying the results has to be in the `main` function or it has to be called from the `main` function. Thus, when developers want to implement their tests in multiple compilation units, they have to explicitly call that code from `main`.

It can be done implicitly. Before the `main` function is executed, the static objects are constructed and their constructors are executed. [8] We can use this feature to execute code at runtime, that is not explicitly called from the `main` function. We create a class for every test execution and add the code displaying the result to the constructor of the class. For example we do the following for `TestPlus`:

```
struct RunTestPlus {
  RunTestPlus() {
    std::cout << "TestPlus: ";
    display_result<TestPlus::type>::run();
    std::cout << std::endl;
  }

  static RuntTestPlus staticInstance;
};
```

Every time `RunTestPlus` is instantiated, it displays the result of the test called `TestPlus`. Note that the test is executed exactly once at compile time regardless of how many times it is displayed at runtime. To display the test results when the compiled code is started, we created a static instance of

`RunTestPlus`. The only purpose of it is that when it is created, its constructor displays the result of `TestPlus`. The `main` function can remain empty, and can be in its own compilation unit. Things will still work.

This solution works for generating simple reports. To build more complex reports, such as an HTML summary, we need to generate a container of the results available at runtime and can be passed to runtime code generating the report.

We need a data-structure that describers the result of one test case. Here is an example implementation of it:

```
class TestResult {
public:
  TestResult(const std::string& name, bool result);
  const std::string& name() const;
  bool result() const;
  // ...
};
```

The instances of `TestResult` are immutable, because they describe the results of a test executed at compile time. Their values are not dependent on anything that happens at runtime. We can build a list of them at runtime:

```
std::list<TestResult> getResults()
{
  std::list<TestResult> results;
  results.push_back(TestResult("TestPlus", TestPlus::value));
  // ...
  return results;
}
```

Every time `getResults` is called, it generates a runtime list containing the results of the test executions. This solution has the same problem we saw earlier. Every test case has to be added to `getResults` explicitly. The idea of using static objects and executing code in their constructors can be reused here. We can create classes for each test case. These classes can add the result of the test to the result list.

The list containing the results has to be a static object, otherwise it would be initialised after the objects that register the test results. There is no guarantee in which order the static objects are initialised across compilation units [8].

This problem has already been solved by the singleton classes of the Boost library. It supports the creation of singleton instances of classes. We create a wrapper for `std::list<TestResult>` and a singleton instance of that class. We call the wrapper class `TestDriver` and add test-registration logic to it.

```
class TestDriver
{
public:
  typedef std::list<TestResult>::const_iterator iterator;

  iterator begin() const;
  iterator end() const;
  // ...
};
```

We provide the iterator interface through `const_iterator`s, so runtime code can't change the results.

We add a member function for executing tests. We can do it in two possible ways. One possibility is adding a member function that takes a boolean value, the result of the test execution. For example:

```
class TestDriver
{
public:
  void addTest(const std::string& name, bool result);
  // ...
};
```

The problem with this solution is that test execution happens *before* `addTest` is called, and it is called with result of the test execution. When test execution goes well, `addTest` is called with a boolean value. When the test case is invalid, for example when it is not a nullary metafunction or when its value is not a wrapper of a boolean value, the compiler displays an error message, that is difficult to interpret and no code, no report is generated.

This can be solved if `TestDriver` takes the *test itself* as argument and not only its result. The test itself is a nullary template metafunction, which is a class. `addTest` can be a template member function [8] taking a class as argument:

```
class TestDriver {
public:
  template <class testCase>
  void addTest(const std::string& name);
  // ...
};
```

When `addTest` is instantiated with a test case, the test case is not executed unless other code explicitly evaluated it. `addTest` can evaluate it and handle some cases when the evaluation of it would lead to a compilation error. `addTest` can verify if it has a nested class called `type` and if that class has a static constant member called `value`. When these things are missing, it doesn't break the compilation, `addTest` can register that the test failed. `TestResult` can be extended with a comment filed, where `addTest` can display the reason why the test case failed to simplify debugging.

3. Evaluation

We compare our testing framework to the unit testing tools used to unit test the Boost template metaprogramming library. The unit tests for that library are in separate files, each file includes a header containing a `main` function to display the results. Each file uses macros to wrap the unit tests and to do static asserts. The verification of the success or failure of the unit tests happens by using compile-time assertions, thus when a test case fails it breaks the compilation of that specific compilation unit. A separate executable is built from each compilation unit. Their `main` function display a summary, but since failing test cases break the compilation, these summaries are displayed only when there are no errors. In comparison, our framework provides a way to execute all unit tests regardless of the number of failures and display a report about the failing test cases.

We have measured the compilation time with both approaches. The compilation time contains the execution of the tests themselves, since test execution happens at compile time. We have done two types of measurements:

- We run the same test several times within one compilation unit. We were using different values to make sure the compiler has to instantiate everything in all cases. Every test execution requires the testing framework to deal with the results. Figure 1 shows the results. We can see that execution time grows in a linear scale for both testing frameworks. For our solution it grows a little faster, which means that processing the result of a test case happens slower in our solution.
- We didn't change the number of the tests, we changed their execution time. We were running a linear algorithm, linear search, with different input lengths. We were always searching the last element of the input to make sure that the search has to iterate over the entire input. The testing framework had to deal with the same number of results, regardless of the execution time of the test cases. Figure 2 shows the results. We can see that there is a constant difference between the compilation times with the two approaches, which means that the extra overhead our solution has for processing the results of a test case is constant.

We did our measurements on a Linux pc with an Intel Atom 1.6Ghz processor and 1 GB memory. We were using gcc version 4.4.3. We were using the `time` command to measure compilation time.

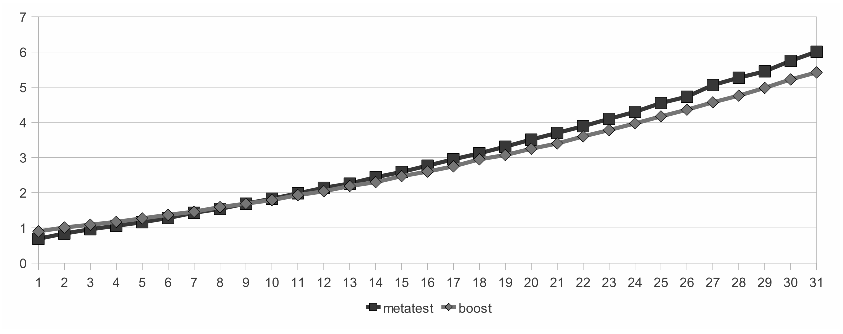

FIGURE 1. Increasing number of test cases

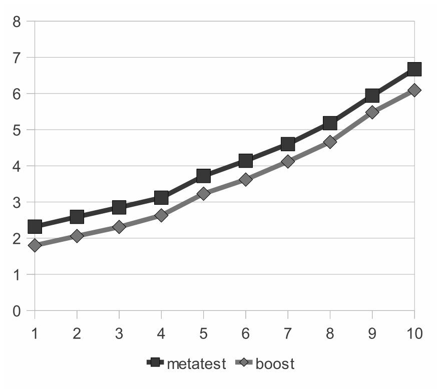

FIGURE 2. Increasing execution time of test cases

We have been using our framework in the development of a number of complex template metaprogramming libraries [5]. We have been unit testing our framework with itself as well.

4. SUMMARY

We have demonstrated a way of building unit testing frameworks for template metaprogramming generating meaningful summaries. We have built a framework using this method, which is available as an open-source project [5].

We have compared our solution to the tools the most widely used template metaprogramming library, `boost::mpl` is tested with. We have seen, that our approach gives a better report about success and failure of test cases.

In the future, these ideas can be reused in other metaprogramming contexts and add value to them as well. Another future work is adding introspection capabilities to the framework to detect how a specific test case works and give more meaningful error messages. For example detect when the test case compares the result of a tested function to another value and when the comparison fails, the error report could display the different values as well. Logical operators, such as `and`, `or`, etc in the test case could be detected and in case of a failure, the root cause of the failure could be pointed out more accurately by the framework. These changes can be done without changing existing test code.

References

[1] E. Unruh, Prime number computation, ANSI X3J16-94-0075/ISO WG21-462.

[2] K. Czarnecki, U. W. Eisenecker, Generative Programming: Methods, Tools and Applications, Addison-Wesley, 2000.

[3] Porkoláb, Z., Mihalicza, J., Sipos, Á.: Debugging C++ Template Metaprograms, in proc. of Generative Programming and Component Engineering (GPCE 2006), The ACM Digital Library pp. 255–264, (2006)

[4] Porkoláb, Z., Mihalicza, J., Pataki, N., Sipos, Á.: Analysis of profiling techniques for C++ template metaprograms, Annales Universitatis Scientiarum Budapestinensis de Rolando Etvs Nominatae, Sectio Computatorica, 30:97-116 (2009)

[5] The source code of mpllibs
`http://github.com/sabel83/mpllibs`

[6] The Boost libraries.
`http://www.boost.org`

[7] Ábel Sinkovics, Functional extensions to the Boost Metaprogram Library, In Porkolab, Pataki (Eds) Proceedings of the 2nd Workshop of Generative Technologies, WGT'10, Paphos, Cyprus. pp.56–66 (2010), ISBN: 978-963-284-140-3

[8] ANSI/ISO C++ Committee, *Programming Languages – C++*, ISO/IEC 14882:1998(E), American National Standards Institute, 1998.

Dept. of Programming Languages and Compilers, Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
E-mail address: `abel@elte.hu`