Acta Univ. Sapientiae, Informatica, **2,** 2 (2010) 154–167

Testing by C++ template metaprograms

Norbert Pataki
Dept. of Programming Languages and Compilers
Faculty of Informatics, Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest,
Hungary
email: patakino@elte.hu

Abstract. Testing is one of the most indispensable tasks in software engineering. The role of testing in software development has grown significantly because testing is able to reveal defects in the code in an early stage of development. Many unit test frameworks compatible with C/C++ code exist, but a standard one is missing. Unfortunately, many unsolved problems can be mentioned with the existing methods, for example usually external tools are necessary for testing C++ programs.

In this paper we present a new approach for testing C++ programs. Our solution is based on C++ template metaprogramming facilities, so it can work with the standard-compliant compilers. The metaprogramming approach ensures that the overhead of testing is minimal at runtime. This approach also supports that the specification language can be customized among other advantages. Nevertheless, the only necessary tool is the compiler itself.

1 Introduction

Testing is the most important method to check programs' correct behaviour. Testing can reveal many problems within the code in development phase. Testing is cruicial from the view of software quality [5]. Many purposes of testing can be, for instance, quality assurance, verification and validation, or reliability estimation. Nonetheless, testing is potentially endless. It can never completely

Computing Classification System 1998: D.2.5
Mathematics Subject Classification 2010: 62N03
Key words and phrases: testing, C++, template metaprogramming

identify all the defects within the software. The main task is is to deliver faultless software [20].

Correctness testing and reliability testing are two major areas of testing. However, many different testing levels are used. In this paper we deal with unit tests that is about correctness. The goal of unit testing is to isolate each part of the program and to show that the individual parts are correct. A unit test provides a strict, written contract that the piece of code must satisfy. As a result, it affords several benefits. Unit tests find problems early in the development phase. Unfortunately, most frameworks need external tools [10].

A testing framework is proposed in [3, 4] which is based on the C++0x – the C++ forthcoming standard. The framework takes advantage of *concepts* and *axioms*. These constructs support the generic programming in C++ as they enable to write type constraints in template parameters. By now, these constructs are removed from the draft of the next standard. Metaprogram testing framework has already been developed [16] too, but it deals with metaprogams, it is just the opposite of our approach.

C++ template metaprogramming is an emerging paradigm which enables to execute algorithms when ordinary C++ programs are compiled. The style of C++ template metaprograms is very similar to the functional programming paradigm. Metaprograms have many advantages that we can harness. Metalevel often subserves the validation [8].

Template metaprograms run at compilation-time, whereupon the overhead at runtime is minimal. Metaprograms' "input" is the runtime C++ program itself, therefore metaprograms are able to retrieve information about the hosting program. This way we can check many properties about the programs during compilation [12, 14, 21, 22].

Another important feature of template metaprograms is the opportunity of *domain-specific languages*. These special purpose languages are integrated into C++ by template metaprograms [7, 9]. Libraries can be found that support the development of domain-specific languages [11]. New languages can be figured out to write C++ template metaprograms [18]. Special specification languages can be used for testing C++ programs without external tools.

In this paper we present a new approach to test C++ code. Our framework is based on the metaprogramming facility of C++. We argue for testing by meta-level because of numerous reasons.

The rest of this paper is organized as follows. In Section 2 C++ template metaprograms are detailed. In Section 3 we present the basic ideas behind our approach, after that in Section 4 we analyze the advantages and disadvantages of our framework. Finally, the future work is detailed in Section 5.

2 C++ template metaprogramming

The template facility of C++ allows writing algorithms and data structures parametrized by types. This abstraction is useful for designing general algorithms like finding an element in a list. The operations of lists of integers, characters or even user defined classes are essentially the same. The only difference between them is the stored type. With templates we can parametrize these list operations by type, thus, we have to write the abstract algorithm only once. The compiler will generate the integer, double, character or user defined class version of the list from it. See the example below:

```
template<typename T>
struct list
{
  void insert( const T& t );
  // ...
};

int main()
{
  list<int> l;      //instantiation for int
  list<double> d;   // and for double
  l.insert( 42 );   // usage
  d.insert( 3.14 ); // usage
}
```

The list type has one template argument `T`. This refers to the parameter type, whose objects will be contained in the list. To use this list we have to generate an instance assigning a specific type to it. The process is called *instantiation*. During this process the compiler replaces the abstract type `T` with a specific type and compiles this newly generated code. The instantiation can be invoked explicitly by the programmer but in most cases it is done implicitly by the compiler when the new list is first referred to.

The template mechanism of C++ enables the definition of partial and full *specializations*. Let us suppose that we would like to create a more space efficient type-specific implementation of the `list` template for the `bool` type. We may define the following specialization:

```
template<>
struct list<bool>
```

```
{
  //type-specific implementation
};
```

The implementation of the specialized version can be totally different from the original one. Only the names of these template types are the same. If during the instantiation the concrete type argument is `bool`, the specific version of `list<bool>` is chosen, otherwise the general one is selected.

Template specialization is an essential practice for template metaprogramming too [1]. In template metaprograms templates usually refer to other templates, sometimes from the same class with different type argument. In this situation an implicit instantiation will be performed. Such chains of recursive instantiations can be terminated by a template specialization. See the following example of calculating the factorial value of 5:

```
template<int N>
struct Factorial
{
  enum { value=N*Factorial<N-1>::value };
};

template<>
struct Factorial<0>
{
  enum { value = 1 };
};

int main()
{
  int result = Factorial<5>::value;
}
```

To initialize the variable `result` here, the expression `Factorial<5>::value` has to be evaluated. As the template argument is not zero, the compiler instantiates the general version of the `Factorial` template with 5. The definition of `value` is `N * Factorial<N-1>::value`, hence the compiler has to instantiate `Factorial` again with 4. This chain continues until the concrete value becomes 0. Then, the compiler chooses the special version of `Factorial` where the `value` is 1. Thus, the instantiation chain is stopped and the factorial of

5 is calculated and used as initial value of the `result` variable in `main`. This metaprogram "runs" while the compiler compiles the code.

Template metaprograms therefore stand for the collection of templates, their instantiations and specializations, and perform operations at compilation time. The basic control structures like iteration and condition appear in them in a functional way [17]. As we can see in the previous example iterations in metaprograms are applied by recursion. Besides, the condition is implemented by a template structure and its specialization.

```
template<bool cond,class Then,class Else>
struct If
{
  typedef Then type;
};

template<class Then, class Else>
struct If<false, Then, Else>
{
  typedef Else type;
};
```

The `If` structure has three template arguments: a boolean and two abstract types. If the `cond` is false, then the partly-specialized version of `If` will be instantiated, thus the `type` will be bound to `Else`. Otherwise the general version of `If` will be instantiated and `type` will be bound to `Then`.

With the help of `If` we can delegate type-related decisions from design time to instantiation (compilation) time. Let us suppose, we want to implement a `max(T,S)` function template comparing values of type T and type S returning the greater value. The problem is how we should define the return value. Which type is "better" to return the result? At design time we do not know the actual type of the `T` and `S` template parameters. However, with a small template metaprogram we can solve the problem:

```
template <class T, class S>
typename If<sizeof(T)<sizeof(S),S,T>::type
  max( T x, S y)
  {
    return x > y ? x : y;
  }
```

Complex data structures are also available for metaprograms. Recursive templates store information in various forms, most frequently as tree structures, or sequences. Tree structures are the favorite forms of implementation of expression templates [24]. The canonical examples for sequential data structures are `typelist` [2] and the elements of the `boost::mpl` library [11].

We define a typelist with the following recursive template:

```
class NullType {};

typedef Typelist<char,Typelist<signed char,
    Typelist<unsigned char,NullType> > >
  Charlist;
```

In the example we store the three character types in a typelist. We can use helper macro definitions to make the syntax more readable.

```
#define TYPELIST_1(x)
          Typelist< x, NullType>
#define TYPELIST_2(x, y)
          Typelist< x, TYPELIST_1(y)>
#define TYPELIST_3(x, y, z)
          Typelist< x, TYPELIST_2(y,z)>
// ...
typedef
TYPELIST_3(char,signed char,unsigned char)
  Charlist;
```

Essential helper functions – like `Length`, which computes the size of a list at compilation time – have been defined in Alexandrescu's Loki library [2] in pure functional programming style. Similar data structures and algorithms can be found in the metaprogramming library [11].

The examples presented in this section expose the different approaches of template metaprograms and ordinary runtime programs. Variables are represented by static constants and enumeration values, control structures are implemented via template specializations, functions are replaced by classes. We use recursive types instead of the usual data structures. Fine visualizer tools can help a lot to comprehend these structures [6].

3 Testing framework

In this section we present the main ideas behind our testing framework which takes advantage of the C++ template metaprogramming.

First, we write a simple type which makes connection between the compilation-time and the runtime data. This is the kernel of the testing framework. If the compilation-time data is not equal to the runtime data, we throw an exception to mark the problem.

```
struct _Invalid
{
  // ...
};

template < int N >
class _Test
{
  const int value;

public:

  _Test( int i ) : value( i )
  {
    if ( value!=N )
      throw _Invalid();
  }

  int get_value() const
  {
    return value;
  }
};
```

Let us consider that a runtime function is written, that calculates the factorial of its argument. This function is written in an iterative way:

```
int factorial( int n )
{
  int f = 1;
  for( int i = 1; i <= n; ++i)
```

```
  {
    f *= i;
  }
  return f;
}
```

It is easy to test the factorial function:

```
template <int N>
_Test<Factorial<N>::value> factorial_test( const _Test<N>& n )
{
  return factorial( n.get_value() );
}
```

When `factorial_test` is called, it takes a compile-time and runtime parameter. The constructor of `_Test` guarantees, that the two parameters are equal. We take advantage of the parameter conversions of C++. When an integer is passed as `_Test`, it automatically calls the constructor of `_Test` which tests if the runtime and compilation time parameters are the same. If the runtime and compilation time parameters disagree, an exception is raised. The return type of `factorial_test` describes that it must compute the `Factorial<N>`. When it returns a value, it also calls the constructor of `_Test`. At compilation time it is computed what the return should be according to the metaprogram specification – e.g. what the `Factorial<N>` is. Because the `factorial_test` takes a `_Test` parameter, two parameters cannot be different. When the `factorial_test` returns it is also evaluates if the result of compilation time algorithm is the same with the result of the runtime algorithm, and an exception raised if it fails. So, we have a runtime and compilation time input, first we calculate the result at compilation time from the compilation time input. At runtime we have the very same input and a runtime function, and evaluates if the runtime algorithm results in the very same output. If it fails an exception is thrown.

Of course, we have to call the `factorial_test` function:

```
int main()
{
  factorial_test< 6 >( 6 );
}
```

In this case, we write `Factorial` metafunction that counts the factorial at compilation time, but we do not have to write this metafunction with metapro-

grams. This metaprogram can be generated by the compiler from a specification that can be defined in EClean [18, 17], Haskell, or other domain-specific language [15].

Instead of return value, references are often used to transmit data to the caller:

```
void inc( int& i )
{
  ++i;
}
```

At this point, we cannot wrap the call of this function into a tester function. Hence, in this case we deal with a new local variable to test.

```
template < int N >
_Test<N+1> inc_test( const _Test<N>& n )
{
  int i = n.get_value();
  inc( i );
  return i;
}
```

Since doubles cannot be template arguments we have to map doubles to integers. The natural way to do this mapping is the usage of the significand and exponent. Here is an example, that presents this idea:

```
template <int A, int B>
struct MetaDouble
{
};


double f( double d )
{
  return d*10;
}

template < int A, int B >
class _Test
{
```

```
  const double d;
public:
  _Test( double x ): d( x )
  {
    if ( get_value() != d )
      throw _Invalid();
  }

  double get_value() const
  {
    return A * pow( 10.0L, B );
  }
};

template <int A, int B>
_Test<A,B+1> f_test(MetaDouble<A, B> d)
{
  double dt = A*pow( 10.0L, B );
  return f( dt );
}
```

This framework can be easily extended in the way of C++ Standard Template Library (STL) [19]. We may use functor objects instead of the equality operator to make the framework more flexible because it can test more relations. We can take advantage of default template parameters of template classes. The following code snippet can be applied to the integers:

```
template <int N, class relation = std::equal_to<int> >
class _Test
{
  const int value;
  const relation rel;

public:
  _Test( int i ) : value( i )
  {
    if ( rel(N, value) )
      throw _Invalid();
  }
```

```
  int get_value() const
  {
    return value;
  }
};
```

4 Evaluation

In this section we argue for our approach. We describe pros and cons and present scenarios where our method is more powerful then the existing ones.

One the most fundamental advantages is that our framework does not need external tools, the only necessary tool is the compiler itself. Nevertheless, another important feature, that we compute the result at compilation time, so the runtime overhead is minimal. Of course, the compilation time is increased. The performance analysis of C++ template metaprograms is detailed in [13].

Our approach is able to detect and pursue the changes external APIs' interface. For instance, the type of return value has been changed, we do not need to modify the specifications. Just like the `max` example in 2 section, metaprograms can determine the type of return values, etc.

Domain-specific languages can be developed with the assistance of template metaprograms. Therefore, specification languages can be easily adopted to our approach. Users can select a specification language from the exisiting ones or develop new domain-specific languages for the specification [23]. The usual specification methods support only one specification language at all.

Moreover, metaprograms are written in an functional way, but runtime C++ programs are written in an imperative way. Therefore, testing approach and implementation is quite different. It is easy to focus on the results this way. A very useful advantage is that that our framework can be used for legacy code too.

Albeit there are some typical approaches which cannot be tested with our method. For instance, metaprograms cannot access database servers and metaprograms cannot deal with other runtime inputs. Files and requests cannot be managed with metaprograms. On the other hand, we can test the business logic of the programs: is the result correct if the input would be the specificated one. Also, calls of virtual methods cannot be handled at compilation time.

Our approach cannot facilitate the testing of multithreaded programs. Test-

ing concurrent programs is hard, but the compiler acts as a single-threaded non-deterministic interpreter.

5 Conclusions and future work

Testing is one of the most important methods to ensures programs' correctness. In this paper we argue for a new approach to test C++ programs. Our solution takes advantage of C++ template metaprogramming techniques in many ways. We have examined the pros and cons of our method.

After all, the most important task is to work out a set of special specification languages and generate standard compliant C++ metaprograms from these specifications.

In this paper we argue for a method that manages runtime entities at compilation time. With this method we tested runtime functions. Many other interesting properties should be managed in this way, for instance, the runtime complexity or the prospective exceptions.

Another important task is developing mapping between the runtime and compile time advanced datastructures. Node-based datastructures (like trees, linked lists) are also available in metalevel, but we have not mapped these structures to runtime akins. User-defined classes also may be mapped to the their compilation-time counterparts.

An other opportunity is that we take advantage of the metalevel and generate testcases at compilation time. In our approach the users specificate the test cases. It would be more convenient if the compiler could generate testcases which covers most of execution paths.

References

[1] D. Abrahams, A. Gurtovoy, *C++ template metaprogramming*, Addison-Wesley, Reading, MA, 2004. ⇒157

[2] A. Alexandrescu, *Modern C++ design: Generic programming and design patterns applied*, Addison-Wesley, Reading, MA, 2001. ⇒159

[3] A. H. Bagge, V. David, M. Haveraaen, The axioms strike back: Testing with concepts and axioms in C++, *Proc. 5th International Conference on Generative Programming and Component Engineering (GPCE'09)*, Denver, CO, 2009, The ACM Digital Library, 15–24. ⇒155

[4] A.H. Bagge, M. Haveraaen, Axiom-based transformations: Optimisation and testing, *Electronic Notes in Theoret. Comput. Sci.*, **238,** 5 (2009) 17–33. ⇒155

[5] M. Biczó, K. Pócza, Z. Porkoláb, I. Forgács, A new concept of effective regression test generation in a C++ specific environment, *Acta Cybern.*, **18** (2008) 481–512. ⇒154

[6] Z. Borók-Nagy, V. Májer, J. Mihalicza, N. Pataki, Z. Porkoláb, Visualization of C++ template metaprograms, *Proc. Tenth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2010)*, Timişoara, Romania, 2010, pp. 167–176. ⇒159

[7] K. Czarnecki, J. T. O'Donnell, J. Striegnitz, W. Taha, DSL implementation in MetaOCaml, template Haskell, and C++, *Lecture Notes in Comput. Sci.*, **3016** (2004) 51–72. ⇒155

[8] G. Dévai, Meta programming on the proof level, *Acta Univ. Sapientiae Inform.*, **1,** 1 (2009) 15–34. ⇒155

[9] J. (Y). Gil, K. Lenz, Simple and safe SQL queries with C++ templates, *Science of Computer Programming*, **75,** 7 (2010) 573–595. ⇒155

[10] P. Hamill, *Unit test frameworks*, O'Reilly Media, Inc., Sebastopol, CA, 2004. ⇒155

[11] B. Karlsson, *Beyond the C++ standard library: An introduction to boost*, Addison-Wesley, Reading, MA, 2005. ⇒155, 159

[12] J. Mihalicza, N. Pataki, Z. Porkoláb, Á. Sipos, Towards more sophisticated access control, *Proc. 11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering*, Tampere, Finland, 2009, pp. 117–131. ⇒155

[13] Z. Porkoláb, J. Mihalicza, N. Pataki, Á. Sipos, Analysis of profiling techniques for C++ template metaprograms, *Ann. Univ. Sci. Budapest. Sect. Comput.* **30** (2009) 97–116. ⇒164

[14] Z. Porkoláb, J. Mihalicza, Á. Sipos, Debugging C++ template metaprograms, *Proc. 5th International Conference on Generative Programming and Component Engineering (GPCE'06)*, Portland, OR, 2006, The ACM Digital Library, 255–264. ⇒155

[15] Z. Porkoláb, Á. Sinkovics, Domain-specific language integration with compile-time parser generator library, *Proc. 9th International Conference on Generative Programming and Component Engineering (GPCE' 10)*, Eindhoven, The Netherlands, 2010, The ACM Digital Library, pp. 137–146. ⇒162

[16] Á. Sinkovics, Unit testing of C++ template metaprograms, *Abstracts 8th Joint Conference on Mathematics and Computer Science (MACS'10)*, Komárno, Slovakia, June 14–17, 2010. ⇒155

[17] Á. Sipos, Z. Porkoláb, N. Pataki, V. Zsók, Meta<Fun> - Towards a functional-style interface for C++ template metaprograms, *Proc. 19th International Symposium of Implementation and Application of Functional Languages (IFL 2007)*, Freiburg, Germany, 2007, 489–502. ⇒158, 162

[18] Á. Sipos, V. Zsók, EClean – An embedded functional language, *Electronic Notes in Theoret. Comput. Sci.*, **238,** 2 (2009) 47–58. ⇒155, 162

[19] B. Stroustrup, *The C++ programming language (Special edition)*, Addison-Wesley, Reading, MA, 2000. ⇒163

[20] Cs. Szabó, L. Samuelis, Observations on incrementality principle within the test preparation process, *Acta Univ. Sapientiae Inform.*, **1,** 1 (2009) 63–70. ⇒155

[21] Z. Szűgyi, N. Pataki, J. Mihalicza, Z. Porkoláb, C++ method utilities, *Proc. Tenth International Scientific Conference on Informatics (Informatics 2009)*, Herl'any, Slovakia, pp. 112–117. ⇒155

[22] Z. Szűgyi, N. Pataki, Sophisticated methods in C++, *Proc. International Scientific Conference on Computer Science and Engineering (CSE 2010)*, Kosice, Slovakia, pp. 93–100. ⇒155

[23] N. Vasudevan, L. Tratt, Comparative study of DSL tools, *Proc. Workshop on Generative Technologies 2010 (WGT 2010)*, Paphos, Cyprus, pp. 67–76. ⇒164

[24] T. L. Veldhuizen, Expression templates, *C++ Report*, **7,** 5 (1995) 26–31. ⇒159

Received: August 5, 2010 • Revised: October 15, 2010