Visualization of C++ Template Metaprograms*

Zoltán Borók-Nagy Viktor Májer József Mihalicza Norbert Pataki
Zoltán Porkoláb
Dept. of Programming Languages and Compilers
Eötvös Loránd University, Faculty of Informatics
Pázmány Péter sétány 1/C
H-1117 Budapest, Hungary
boroknagyz@gmail.com, majer_v@inf.elte.hu, jmihalicza@gmail.com, patakino@elte.hu
gsd@elte.hu

Abstract

Template metaprograms have become an essential part of today's C++ programs: with proper template definitions we can force the C++ compiler to execute algorithms at compilation time. Among the application areas of template metaprograms are the expression templates, static interface checking, code optimization with adaptation, language embedding and active libraries. Despite all of its already proven benefits and numerous successful applications there are surprisingly few tools for creating, supporting, and analyzing C++ template metaprograms. As metaprograms are executed at compilation time they are even harder to understand. In this paper we present a code visualization tool, which is utilizing Templight, our previously developed C++ template metaprogram debugger. Using the tool it is possible to visualize the instantiation chain of C++ templates and follow the execution of metaprograms. Various presentation layers, filtering of template instances and step-by-step replay of the instantiations are supported. Our tool can help to test, optimize, maintain C++ template metaprograms, and can enhance their acceptance in the software industry.

1. Introduction

Templates are key elements of the C++ programming language [18]. They enable data structures and algorithms to be parameterized by types thus capturing commonalities of abstractions at compilation time without performance penalties at runtime [21]. *Generic programming* is recently a popular programming paradigm [14, 16], which enables the developer to implement reusable codes easily. Reusable components – in most cases data structures and algorithms – are implemented in C++ with the heavy use of templates. The most notable example is the Standard Template Library [9] is now an unavoidable part of professional C++ programs.

In C++, in order to use a template with some specific type, an *instantiation* is required [22]. This process can be initiated either implicitly by the compiler when a template with a new type argument is referred, or explicitly by the programmer. During instantiation the template parameters are substituted with the concrete arguments, and the generated new code is compiled.

This instantiation mechanism enables us to write smart template codes that execute algorithms at compilation time. In 1994 Erwin Unruh wrote a program in C++ that did not compile, however, the error messages emitted by the compiler during the compilation process displayed a list of prime numbers [20]. Unruh used C++ templates and the template instantiation rules to write a program that is "executed" as a side effect of compilation. It turned out that a cleverly designed C++ code is able to utilize the template mechanism of the language and force the compiler to execute a desired algorithm [23]. These compile-time programs are called C++ *template metaprograms*. Later, template metaprograms' Turing-completeness has been proved [3].

Template metaprogramming is now an emerging direction in C++ programming for executing algorithms at compilation time. We write metaprograms for various reasons, like *expression templates* [24] replacing runtime computations with compile-time activities to enhance runtime performance, *static interface checking*, [10, 15, 26], which increases the ability of the compile-time to check the requirements against template parameters, i.e. they form constraints on template parameters, *active libraries* [4], acting

*Supported by TÁMOP-4.2.1/B-09/1/KMR-2010-0003 dynamically during compile-time, making decisions based on programming contexts and making optimizations, and many others.

As template metaprograms became popular in industrial applications, their maintenance turned to be a key issue. Unfortunately, here programmers have experienced major problems. C++ templates have been designed to express genericity on data structures and algorithms – i.e. parametric polymorphism. Template metaprogramming has been discovered almost as a side effect, and template syntax that time has already been formulated. That syntax is far to be expressive regarding template metaprograms. As a result, C++ template metaprograms are often difficult to read, and hard to understand.

Moreover, basic program comprehension techniques, like step by step execution or debugging of template metaprograms are troublesome. When compiling, the standard C++ compiler acts as an interpreter which executes a template metaprogram. Debugging the compiler is useless: we want to collect information about the interpreted code (i.e. the template metaprogram) not about the interpreter (the compiler) itself. Even printouts applied in selected points of metaprogram execution are challenging. As a result, maintenance of template metaprograms currently is an art rather than an engineering process.

In this paper we discuss code comprehension techniques of C++ template metaprograms. We present a visual debugger for *Templight* framework. Using the tool we can reveal the structure of a template metaprogram as it was executed by the compiler in a certain platform. We are able to replay the instantiation process in a step by step manner, both forward and backward direction. Usual debugger functionalities: step into/out/over instantiations are implemented. Break-points can be placed at source level. In the debugger syntax highlighting, watch windows and other conveniences help us to understand metaprograms. A second program – *Templar*, a code visualisation and comprehension tool – provides a graphical representation of template metaprograms. Various layouts and visualisations help us to understand template metaprograms displayed in graph representation. A filtering function is available to eliminate uninteresting nodes from the visual representation, thus we can concentrate on the important template instantiations.

There are only a few available tools supporting C++ template metaprogramming. Sánchez at al. published a poster [13] about visualizing template instantiations, but we did not find any continuation. Steven Watanabe created a template profiler, which is currently in the `boost` sandbox [28]. His work is useful to measure compilation times (run time of template metaprograms), but does not help code comprehension.

The rest of the paper is organized as follows. In Section 2 we overview C++ template metaprogramming and argue for an adequate code comprehension tool. We explain the *Templight* template metaprogram debugger – the foundation of our code visualisation tool in Section 3. In Section 4 we present our debugging and visualization tools. In Section 5 we evaluate the tool using examples. We conclude our paper in Section 6.

2. Understanding template metaprograms

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); 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. That 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 either 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 `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 essential practice for template metaprogramming too. 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 [8]. 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 `boost::mpl` metaprogramming library [8].

The examples presented in this Section are far more trivial than the real life applications of template metaprograms. However, they already show why template metaprograms are hard to understand. 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.

Professional C++ programmers are eager for tools which support them to maintain template metaprograms. Such tools should provide the following features:

- Reveal the actual instantiation process executed by the compiler.
- Replay the process in a step by step (both forward and backward manner).
- Usual debugger functionality step into/out/over instantiations.

- Break-points can be placed to help fast forward/stop replay process.
- Watch windows to inspect various template features.
- Present the state of the template (before/under/after) instantiation.
- Filter out irrelevant/not in interest instantiation steps.
- Help to understand the overall structure with proper visualisation.

In this paper we present our toolset which meets these requirements. The *Templar* template metaprogram visualisation tool presents the template metaprogram as a graph, where nodes represent the types generated from templates, and the edges show the instantiation requests. The user is able to replay the instantiation process either step by step or in a fast forward mode stopping at breakpoints. Backward mode is also available. The graph visualizes the actual state of each nodes.

There are cases when we are not interested in certain instantiations, i.e. "utility templates". Such nodes can be removed from the view, and their in- and outcoming edges are merged to the remaining graph. This filtering is based on template names of the source code.

We want to emphasize that pure static analysis techniques have their certain limitations in the space of template metaprogram comprehension. The standard of the C++ programming language strictly defines which templates should be instantiated, and how the generated code behaves. There are very good code visualisation tools for large-scale C++ programs based on static code analysis [19]. However, in the case of metaprograms it is up to the implementation how the instantiation process is implemented, how *memoization* is handled [1] and what is the resource requirement (both in factor of time and memory) for the code generation. Such details are compiler dependent, and a general static analysis tool would fail to discover the differences.

Therefore our method is not based purely on static analysis, but on the actual compilation process on the target compiler and platform. We instrument the source code using *Templight*, our template metaprogram debugger [11], and generate a trace file containing all the necessary information on the template metaprogram execution. Templar, the code visualisation tool, utilizes this information and presents it to the user graphically.

Figure 1 shows the factorial program in the state when `Factorial<4>` starts to instantiate `Factorial<3>`. The specialization `Factorial<1>` has already been created.

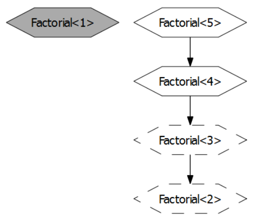


Figure 1. The factorial sample

3. Templight framework

With the Templight framework [11] we can generate a template instantiation log of the compilation process. We refer to this log as *trace*. This trace is a chronological list of events like

- instantiation of template `A` began
- type definition found
- instantiation of template `A` ended
- compilation warning

Having this log generated, we are able to reconstruct the compilation process from a template metaprogramming perspective. The order of template instantiation begin and end events determine the *instantiation stack* for every moment. As crashdumps allow post-mortem debugging of runtime programs, traces allow post-compilation analysis of template metaprograms. Similarly to many functional languages, the constructs of C++ template metaprograms have a stateless nature. Therefore the debugging process is not a one way road any more, we can freely step back and forth in time.

This Section shows the basic idea behind Templight and describes the elements of the tool chain.

The basic input unit of Templight is a compilation unit. Compilation units are processed independently by the compiler, consequently generating the trace for them one by one does not hurt generality. Trace file generation from a compilation unit has three main stages: annotation, patch and extraction. *Annotation* analyses the (preprocessed) source file and locates template metaprogram constructs in it. *Patch* then inserts special code fragments at these locations, which have the following properties:

- they do not change the semantics of the code at all
- compilation warnings are printed whenever the compiler processes them

Throughout the whole process we should be very careful not to alter the compilation process logically. At any point if the compiler processes the templates in a different manner due to our modifications, our visualization tool is useless, as it does not show the real picture.

After patching our source file, we have to compile this modified unit and save the compiler output, which will contain the warning messages produced by the instrumented code fragments (on top of the originally existing ones). The *extraction* phase retrieves the necessary information from the special warning messages and generates the trace file.

To show the original source context of an event we have to transform the position reported in the warning back to its initial form. For this, it is important to always keep track of the position changes when the source is modified.

Preprocessing makes our life a lot easier. After that there is only one file, all the `#include`s and macros are processed and the file is a series of tokens.

This step utilizes `boost::wave`[8], a C++ preprocessor library to get a token sequence from the preprocessed source. The task in this step is to identify as many template constructs in the token sequence as possible and record their details. Currently these constructs are found by using regular expressions of the form like `template` *keyword* + *arbitrary tokens* + `class` *or* `struct` *keyword* + *arbitrary tokens* + {.

Unfortunately the context sensitive grammar of C++ allows expressions which are very difficult to disambiguate with such a simple tool as regular expressions. The many possible roles of commas and relational operators make the proper automatic identification of an enumeration type almost impossible.

The purpose of this phase is to transform the source code so that informative error messages will be emitted at important positions of each template construct when they are instantiated. The inserted code fragments must not change instantiation order, nor the size and other properties of the structures to preserve the original compilation flow. In the following `g++` example instrumentation points detected by the annotator are marked with `/**/`:

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

// After instrumentation:
static int TEMPLIGHT__begin(char*);
static int TEMPLIGHT__end(wchar_t*);
template<int N> struct Factorial {
  enum { TEMPLIGHT__patch_0
    = sizeof(TEMPLIGHT__begin("")) };
  enum { value=N*Factorial<N-1>::value };
  enum { TEMPLIGHT__patch_1
    = sizeof(TEMPLIGHT__end("")) };
};
```

During compilation whenever an instance of `Factorial` is instantiated, we get a warning both at the beginning and at the end of the given instance. The warning messages will refer to code positions in this modified source, so here it is again important to record the location changes to have an exact mapping.

Having our warning emitter snippets added, we can compile the patched code. The compiler output contains progress output, (name of file actually being processed), possible error messages, and warning messages (both original or artificial ones, triggered by instrumented code).

4. Debugging and Visualisation

To visualize template metaprograms, we build a directed graph where nodes represent instantiated templates (so called specializations) and edges represent the requests for instantiation. I.e. we have an edge from node `A` to node `B` if `A` is a template which instantiates `B`. The trace file which is generated by Templight has all the information we need. There are entries in a sequential order. An entry can be `TemplateBegin` which means a class template's instantiation is started. It also can be `TemplateEnd` that means a template class's instantiation has been finished. The entries are in the same order as they occured at compilation time.

An entry contains the name of the template class and file positions. This is enough to build the instantiation graph which is always a tree. We use a very simple algorithm for it. For example there is a `TemplateBegin` entry of type `A` followed by a `TemplateBegin` entry of type `B`. That means type `A` instantiates type `B`. If the next entry is a `TemplateEnd` of type `B`, and after that there is a `TemplateBegin` of type `C` that also means that type `A` instantiates type `C`. The trace has a custom XML format:

```
<TemplateBegin>
 <Position pos="fact.patched.cpp|9|1"/>
 <Context context="Factorial<5>"/>
</TemplateBegin>
<TemplateBegin>
 <Position pos="fact.patched.cpp|9|1"/>
 <Context context="Factorial<4>"/>
</TemplateBegin>
...
<TemplateEnd>
 <Position pos="fact.patched.cpp|13|1"/>
 <Context context="Factorial<4>"/>
</TemplateEnd>
<TemplateEnd>
 <Position pos="fact.patched.cpp|13|1"/>
 <Context context="Factorial<5>"/>
</TemplateEnd>
```

We transform the locations referring to the preprocessed and instrumented code back to their original source file and line positions. First, using the line map saved at instrumentation, the line numbers are mapped back to their non-instrumented variants. These line positions correspond to the preprocessed, but not yet patched code. Then, based on the location map containing the `#line` mapping, the original source code positions are determined.

This transformation is applied right before the trace file is written. We use the stack data structure to treat the instantiation relationships correct. If we read a `TemplateBegin` entry, we create a node. If the stack is not empty, we add an edge to the graph between the node of the top of the stack and the new node we created. After that, we push the new node to the stack. If we read a `TemplateEnd` entry, we call the pop member function on the stack.

4.1. Handling inheritance

Consider the following simple example:

```
template <typename T>
class Derived : public Base<T>
{ ... };
```

Given a concrete instance, to say with `T=int`, we would expect something like this in the trace file:

```
Derived<int> begin
  Base<int> begin
  Base<int> end
Derived<int> end
```

Without special inheritance handling, however, the basic annotation and instrumentation would give different results:

```
template <typename T>
class Derived : public Base<T>
{ /* instrumentation point
     for TemplateBegin */
  // ..
  /* instrumentation point
     for TemplateEnd */
};
```

Here the `TemplateBegin` event for `Derived<int>` will be reported only *after* `Base<int>` has been completely processed, thus producing the following sequence:

```
Base<int> begin
Base<int> end
Derived<int> begin
Derived<int>
```

which is not exactly what we expect. To solve the problem, an artificial extra first base class is instrumented into all base class list:

```
template <typename T>
class Derived :
  public TEMPLIGHT_BASE<Derived<T> >,
  public Base<T>
```

where `TEMPLIGHT_BASE` generates a warning right before the instantiation of `Base` begins.

4.2. The debugger

Initially Templight was only a set of command line utilities and the user had to apply the preprocess, instrumentation, warning translation etc. steps manually. The only move towards an interactive graphical user interface was a plugin to an existing IDE on a specific platform. We decided to create an appropriate frontend to our framework that makes template debugging not only possible but convenient. We kept platform independence in focus, our application has only such dependencies which are available on different platforms. Also, the internals of Templight have been changed to better handle large projects, the initial version had serious performance issues.

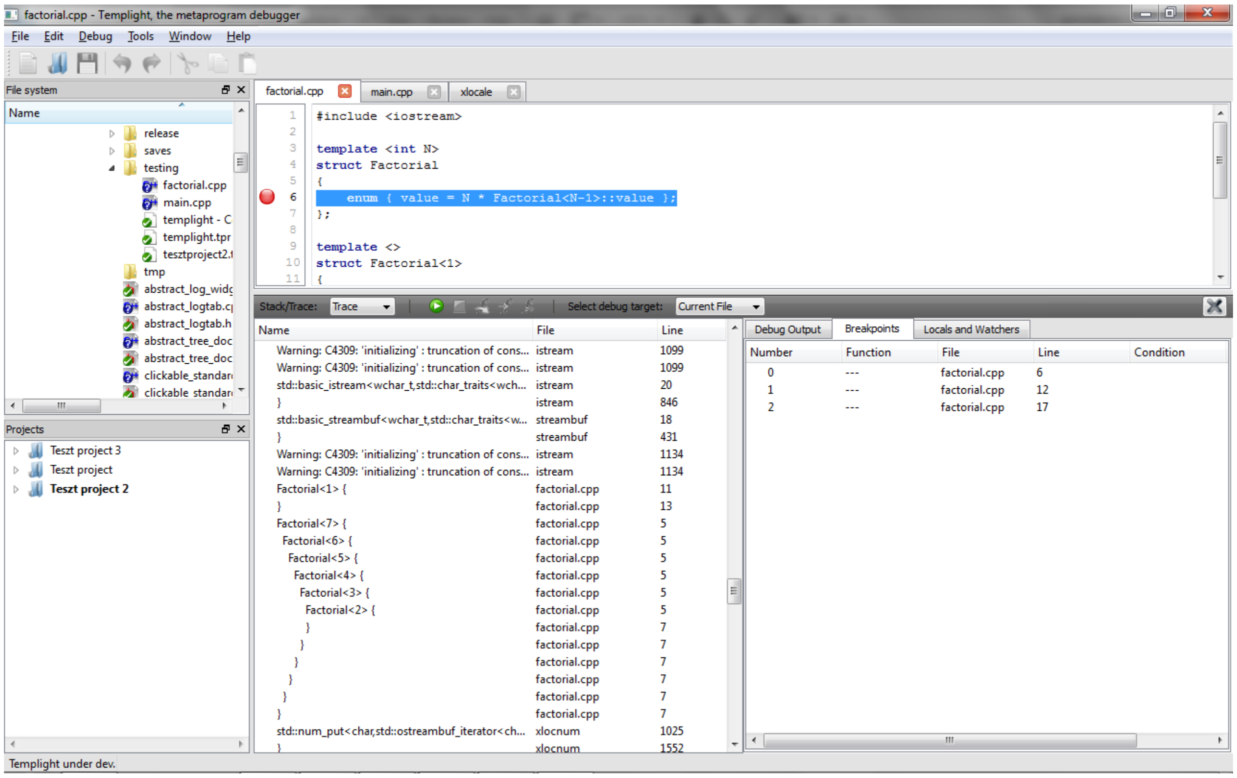
Our debugger provides basic file browsing, text editing functionality with syntax highlight and supports template debugging. There is a quick debug use case where the actually opened file can be debugged with a single click. This makes experimenting very convenient. The other use case allows us to set up projects with their corresponding compiler settings and debug their different files from time to time as the project evolves.

All the standard debugging features are available for templates to the extent of Templight's capabilities. The template instantiation stack shows the chain of template instantiations calling each other. We can interactively go through the compilation process with the very same commands as in normal debugging: run, continue, step in, step over, step out. We can set breakpoints inside templates, and we can check from what context(s) that specific template is instantiated.

There is also a trace window where the whole flow of the compilation can be observed.

4.3. Templar visualizer

We used the builder design pattern in the visualizer tool. At the time we have only one concrete builder class named GraphvizBuilder. It builds a graphviz graph but with this pattern it is easy to build any kind of graphs (e.g. BGL, Lemon [5]).



```cpp
#include <iostream>

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

template <>
struct Factorial<1>
{
```

```
Name                                                File            Line
  Warning: C4309: 'initializing' : truncation of cons...  istream         1099
  Warning: C4309: 'initializing' : truncation of cons...  istream         1099
  std::basic_istream<wchar_t,std::char_traits<wch...     istream         20
  }                                                       istream         846
  std::basic_streambuf<wchar_t,std::char_traits<w...     streambuf       18
  }                                                       streambuf       431
  Warning: C4309: 'initializing' : truncation of cons...  istream         1134
  Warning: C4309: 'initializing' : truncation of cons...  istream         1134
  Factorial<1> {                                          factorial.cpp   11
  }                                                       factorial.cpp   13
  Factorial<7> {                                          factorial.cpp   5
   Factorial<6> {                                         factorial.cpp   5
    Factorial<5> {                                        factorial.cpp   5
     Factorial<4> {                                       factorial.cpp   5
      Factorial<3> {                                      factorial.cpp   5
       Factorial<2> {                                     factorial.cpp   5
       }                                                  factorial.cpp   7
      }                                                   factorial.cpp   7
     }                                                    factorial.cpp   7
    }                                                     factorial.cpp   7
   }                                                      factorial.cpp   7
  }                                                       factorial.cpp   7
  std::num_put<char,std::ostreambuf_iterator<ch...       xlocnum         1025
  }                                                       xlocnum         1552
```

Number	Function	File	Line	Condition
0	---	factorial.cpp	6	
1	---	factorial.cpp	12	
2	---	factorial.cpp	17	

Figure 2. The Templight debugger displaying Factorial sample

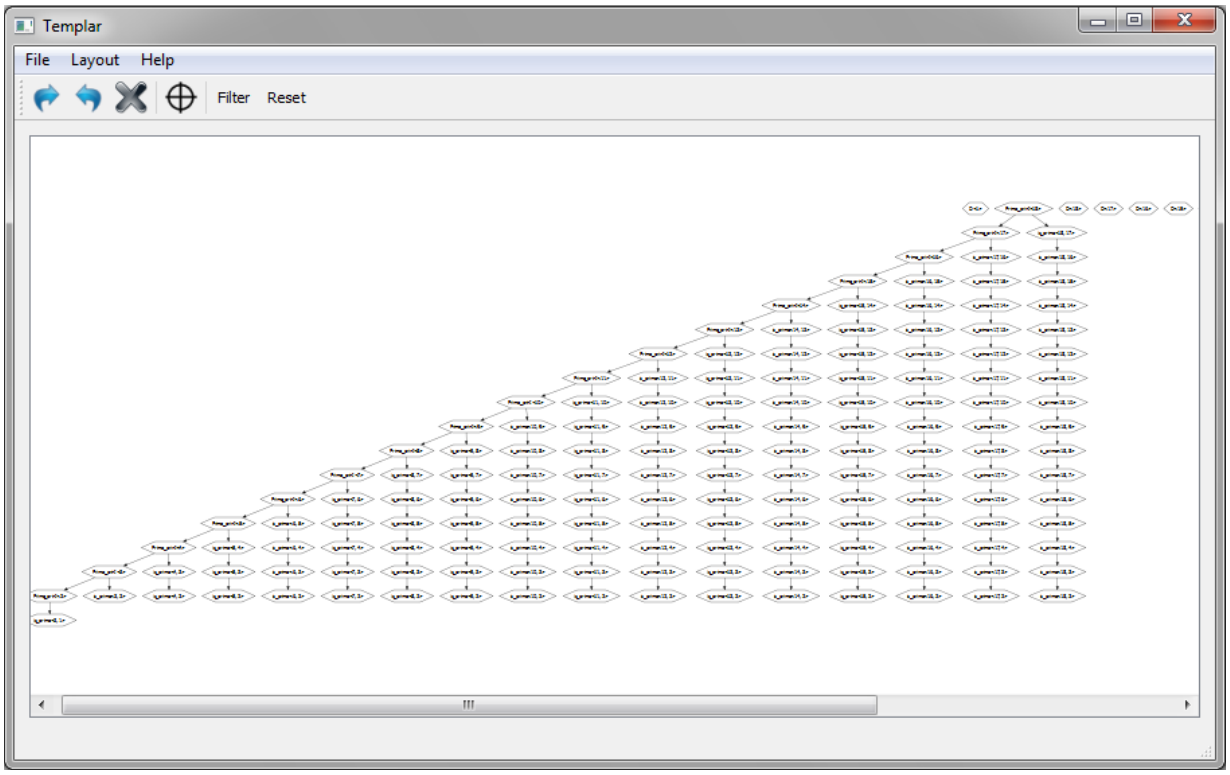


Figure 3. The Templar visualisation tool displaying Prime_print sample using dot layout

As we mentioned, we create a *graphviz* graph. Graphviz [27] is an open source graph visualization software that includes an open set of layouts. We have chosen Graphviz for the ease of simplicity, but there are many other possibilities to pick up a visualisation tool. We are currently evaluating the possible parametrization of the graphical visualisation tool and the possible candidates [6, 7].

By default, we use the dot layout because we found it suitable for directed hierarchical acyclic graphs. The layout of the graph is done by graphviz. Other graph layouts (*neato*, *fdp*, *twopi*, *circo*) can be also selected, but they seemed to be suitable for less use cases. We have added coloring features to widgets, to express the state of the nodes. We can also select a node, and there is an option to set the center of the view to the node whose color has changed.

To really understand what happens during compilation there is feature to replay the instantiation process step by step. Our GraphvizBuilder class stores every entry from the tracefile. To replay the process we iterate over that list and notify our QGraph class to visualize the steps forward and backward.

There is an option to focus to the node of which state has been changed. This small enhancement proved inevitable for practical use of the tool.

Template metaprograms frequently use utility templates from third party libraries. They are important for compilation but irrelevant or disturbing to understand the general structure of template metaprograms. Such utility templates are common in the `boost::mpl` and `loki` libraries. Similarly, “native” templates, like STL containers are better not to be presented in our charts. Templar can filter out these irrelevant nodes. User can define filtering patterns with *regular expressions*. This process regenarates the graph from the trace file ignoring those classes whose names match to any of the given regular expressions. Incoming and outcoming edges of filtered nodes are merged to remaining nodes.

We mentioned that there are file positions in the trace file. With that information it is also a feature to see the source code of the appointed template class.

Finally our tool can export the visualized graph to PNG. For small graphs it is done by graphviz, for big graphs it is done by Qt. We make monochromatic images from very large graphs.

5. Evaluation

In this Section we present the power of our tool based on a well-known metaprogram. With the assistance of the visualizer tool we understand a quite abstruse program.

Our sample code is the very first metaprogram created by Erwin Unruh in 1994 [20]. We present the code in an up-to-date version.

```
template <int p, int i>
struct is_prime {
  enum {
    prim = (p==2) ||
           (p%i) &&
           is_prime<(i>2?p:0),i-1>::prim
  };
};
template<>
struct is_prime<0,0> { enum {prim=1}; };
template<>
struct is_prime<0,1> { enum {prim=1}; };
template <int i>
struct D {
    D(void*);
};
template <int i>
struct Prime_print {
  Prime_print<i-1> a;
  enum
   { prim = is_prime<i,i-1>::prim };
  void f() {
      D<i> d = prim ? 1 : 0;
      a.f();
  }
};
template<>
struct Prime_print<1> {
  enum {prim=0};
  void f() {
      D<1> d = prim ? 1 : 0;
  }
};

#define LAST 18

int main() {
    Prime_print<LAST> a;
    a.f();
}
```

Strictly speaking, the code is incomprehensible. It is hard to understand what the program does and how it works. So, before we analyze the code, we try it. But when the program has been compiled, error diagnostics are emitted:

```
error:   initializing argument 1 of
      'D<i>::D(void*) [with int i = 17]'
error:   initializing argument 1 of
      'D<i>::D(void*) [with int i = 13]'
error:   initializing argument 1 of
      'D<i>::D(void*) [with int i = 11]'
error:   initializing argument 1 of
      'D<i>::D(void*) [with int i = 7]'
```

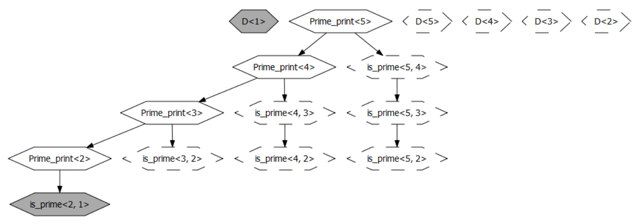


Figure 4. Prime numbers generation – instantiation of is_prime<2,1> is just completed.

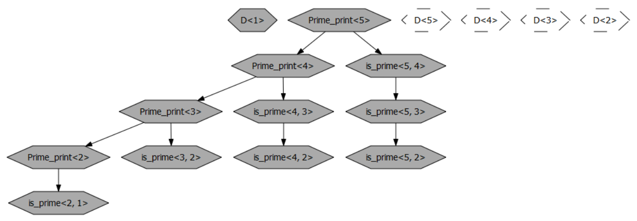


Figure 5. Prime numbers generation – all is_prime have been instantiated, generating instances of template D starts.

```
error:   initializing argument 1 of
      'D<i>::D(void*) [with int i = 5]'
error:   initializing argument 1 of
      'D<i>::D(void*) [with int i = 3]'
error:   initializing argument 1 of
      'D<i>::D(void*) [with int i = 2]'
```

Now we have figured out what the code does. This program generates primes numbers at compilation time, and prints them in form of error messages. But many questions are arisen. How does the code work? Where do the prime numbers come from? Why one gets error messages when the program is being compiled? Why are the primes printed in error messages? What does cause error messages?

Let us consider the previous code. The error message points to the constructor of class template D. This constructor takes a parameter of void*, but where the constructor is called integers are passed. In the code the integer is 0 or 1 and only the integer 0 can be passed as void*. If the argument of constructor is 1, it results in an error. But what is the problem with prime numbers?

Figure 4 and figure 5 make the flow much more clear. The LAST macro is set to 5. First, Prime_print<5> is instantiated which starts a loop, and instantiates Prime_print with 4, 3, 2 and 1. Every Prime_print<i> instantiates is_prime<i, i-1>. These is_prime templates start instantiate themselves, by decreasing their second argument one by one to 1. These class templates determine if their first argument is prime. In the Prime_print class templates use the class template D's constructor as passed is_prime information about primeness in the member function f. If it is prime, then an error message is generated as seen before.

Figure 3 also presents the characteristics of the program. However, we cannot read the "runtime" complexity of the program easily. It is very hard to analyze the previous code's runtime complexity. Figure 3 presents that the algorithm runs in quadratic time. This means that the compiler has to instantiate asymptotically ($N*N$) templates, where N stands for the LAST constant.

Figures have proven that our framework makes metaprograms easier to understand and maintain. With the help of our tool one can estimate the run time of metaprograms too.

6. Conclusion

In this paper we presented a toolset for C++ template metaprogram comprehension. Based on our previous Templight framework, we have developed a graphical user-interfaced debugger and visualizer tool. Using the tools we can reveal the structure of a template metaprogram as it was executed by the compiler on a certain platform.

We are able to replay the instantiation process in a step by step manner, both in forward and backward direction. Usual debugger functionalities: step into/out/over instantiations are implemented. Breakpoints can be placed in source level. In the debugger syntax highlighting, watch windows and other conveniences help us to understand metaprograms. Using the visualisation tool metaprograms appear in a graph based representation in different layouts. Template instantiations out of interest can be filtered out to focus on relevant instantiations. We believe that with the help of this toolset maintenance of template metaprograms is easier and more reliable.

It is up to the implementation how the instantiation process is implemented, memoization is handled and what is the resource requirement for the code generation. Such details are compiler dependent, and a general static analysis tool would fail to discover the differences. Our tool utilizes the results of the actual compilation section, therefore it is essential to reveal such implementation details.

References

[1] Abrahams, D., Gurtovoy, A.: *C++ template metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond*, Addison-Wesley, Boston, 2004.

[2] Alexandrescu, A.: *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, 2001.

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

[4] Czarnecki, K., Eisenecker, U. W., Glück, R., Vandevoorde, D., Veldhuizen, T. L.: *Generative Programming and Active Libraries*, Springer-Verlag, 2000.

[5] Dezső, B., Jüttner, A., Kovács, P: *LEMON – an Open Source C++ Graph Template Library*, in Proceedings of Workshop on Generative Technologies 2010, pp. 3–14.

[6] Holten, D.: *Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data*, IEEE Transactions on Visualization and Computer Graphics (TVCG; Proceedings of Vis/InfoVis 2006), Vol. 12, No. 5, pp. 741–748, 2006.

[7] H. Hoogendorp, O. Ersoy, D. Reniers, A. Telea: *Extraction and Visualization of Call Dependencies for Large C/C++ Code Bases: A Comparative Study*, In Proc. VISSOFT. IEEE, 2009. to appear.

[8] Karlsson, B.: *Beyond the C++ Standard Library: An Introduction to Boost*, Addison-Wesley, 2005.

[9] Musser, D. R., Stepanov, A. A.: *Algorithm-oriented Generic Libraries*, Software-practice and experience, 27(7) July 1994, pp. 623–642.

[10] McNamara, B., Smaragdakis, Y.: *Static interfaces in C++*, In First Workshop on C++ Template Metaprogramming, October 2000.

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

[12] Reis, G. D., Stroustrup, B: *Specifying C++ concepts*, Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2006: pp. 295–308.

[13] Sánchez, A. J., Dei-Wei, J.: *Towards a graphical notation to express the C++ template instantiation process (poster session)*, Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum), (OOPSLA 2000), Minneapolis, Minnesota, United States, January 2000, pp. 117–118.

[14] Siek, J.: *A Language for Generic Programming*, PhD thesis, Indiana University, 2005.

[15] Siek, J., Lumsdaine, A.: *Concept checking: Binding parametric polymorphism in C++*, In First Workshop on C++ Template Metaprogramming, October 2000.

[16] Siek, J., Lumsdaine, A.: *Essential Language Support for Generic Programming*, Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation, New York, USA, pp 73-84.

[17] Sipos, Á., Porkoláb, Z., Pataki, N., Zsók, V.: Meta<Fun> - *Towards a Functional-Style Interface for C++ Template Metaprograms*, in Proceedings of 19th International Symposium of Implementation and Application of Functional Languages (IFL 2007), pp. 489–502.

[18] Stroustrup, B.: *The C++ Programming Language Special Edition*, Addison-Wesley, 2000.

[19] Telea, A., Voinea, L.: *An interactive reverse engineering environment for large-scale C++ code*, In Koschke, R., Hundhausen, C., D., Telea, A. (Eds.): Proceedings of the ACM 2008 Symposium on Software Visualization, Ammersee, Germany, September 16-17, 2008. ACM 2008, ISBN 978-1-60558-112-5SOFTVIS pp. 67–76.

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

[21] Vandevoorde, D., Josuttis, N. M.: *C++ Templates: The Complete Guide* Addison-Wesley, 2003.

[22] Veldhuizen, T. L.: *Five compilation models for C++ templates*, In First Workshop on C++ Template Metaprogramming, October 2000.

[23] Veldhuizen, T. L.: *Using C++ Template Metaprograms*, C++ Report vol. 7, no. 4, 1995, pp. 36–43.

[24] Veldhuizen, T. L.: *Expression Templates*, C++ Report vol. 7, no. 5, 1995, pp. 26–31.

[25] Wills, G. J.: *NicheWorks – Interactive Visualization of Very Large Graphs*, Journal of Computational and Graphical Statistics, 8(3), June 1999, pp. 190–212.

[26] Zólyomi, I., Porkoláb, Z: *Towards a template introspection library*, LNCS Vol.3286, 2004, pp.266–282.

[27] *Graphviz, the Graph Visualization Software*, `http://www.graphviz.org/`

[28] *Watanabe's Template profiler in boost sandbox*, `https://svn.boost.org/svn/boost/sandbox/tools/profile_templates/`