TUAnalyzer—Analyzing Templates in C++ Code

Thomas Gschwind[*]
Technische Universität Wien
Institut für Informationssysteme
Argentinierstraße 8, A-1040 Wien

Martin Pinzger[†]
Technische Universität Wien
Institut für Informationssysteme
Argentinierstraße 8, A-1040 Wien

Harald Gall
Universität Zürich
Institut für Informatik
Winterthurerstrasse 190, CH-8057 Zürich

Abstract

In this paper, we present TUAnalyzer, a novel tool that extracts the template structure of C++ programs on the basis of the GNU C/C++ Compiler's internal representation of a C/C++ translation unit. In comparison to other such tools, our tool is capable of supporting the extraction of function invocations that depend on the particular instantiation of C++ templates and to relate them to their particular template instantiation. TUAnalyzer produces RSF format output that can be easily fed into existing visualization and analysis tools such as Rigi or Graphviz. We motivate why this kind of template analysis information is essential to understand real-world legacy C++ applications. We present how our tool extracts this kind of information to allow others to build on our results and further use the template information. The applicability of our tool has been validated on real code as proof of concept. The results obtained with TUAnalyzer enable us and other approaches and tools to perform detailed studies of large (open source) C/C++ projects in the near future.

1. Introduction

The quality of reverse engineering results strongly depends on underlying data that is obtained through fact extraction techniques. Frequently, existing reverse engineering approaches extract these facts by parsing the source code that needs to be reverse engineered. Parsing source code of large complex software systems, however, needs highly sophisticated parsing approaches, especially when dealing with language and compiler specific features such as C++ templates.

Templates are a very powerful C++ programming language feature that makes it intrinsically hard to implement analyzers that are able to parse such code while at the same time evaluating the template instantiations. Currently available reverse engineering and parsing tools that concentrate on C/C++ source code such as Imagix-4D [9], SourceNavigator [13], Columbus [4], GCC_XML [6], or CPPX [2] lack of sufficiently handling C++ templates. Although, some of these tools extract basic information about templates, they do not fully handle template instantiations or template parameter analysis.

In this paper, we present the TUAnalyzer approach to extract facts from C/C++ source code with focus on C++ templates and virtual method calls. We use the GNU C/C++ compiler as front-end to parse the C/C++ code that needs to be reverse engineered. GCC's translation unit is then used to analyze function and class templates as well as inheritance relationships and virtual function calls. This allows us to find out the specific functions to which the invocation resolves during compilation. The output of TUAnalyzer is in FAMIX [12] compliant Rigi Standard Format (RSF) [16]. As a results, we can feed that resolved function information back into our other external reverse engineering tools and produce complete function call graphs or dependence graphs. To evaluate our tool and compare it to other currently available tools, we developed a number of small code samples that exhibit the challenges of analyzing C++ templates and virtual method calls. These examples are based on our experiences in analyzing the source code of the Mozilla web browser [18] and in teaching the C++ programming language at our university.

In Section 2, we present some common C++ template examples taken from the C++ Standard Library and the Mozilla open source web browser, respectively, that cannot be completely analyzed using existing C++ analysis tools. In Section 3, we provide a short overview on the approaches we have worked out to analyze C++ template instantiations. Section 4 explains in detail how these instantiations can be derived. The analysis of records, inheritance relationships, and the resolution of virtual member function invocations is presented in Section 5. Section 6 presents the validation of TUAnalyzer. We compare our work to other approaches in this field in Section 7 and draw our conclusions in Section 8.

* Presently at: IBM Research, Zurich Research Laboratory, Säumerstrasse 4, CH-8803 Rüschlikon.

† Presently at: Universität Zürich, Institut für Informatik, Winterthurerstrasse 190, CH-8057 Zürich.

2. Background

The background and motivation of our approach is in the analysis of software systems that heavily use C++ templates. For instance, we work on analyzing the Mozilla open source web browser. Applying existing parsing tools (Imagix-4D [9], GCC_XML [6], Columbus [4], CPPX [2]) we discovered that important information about C++ templates has not been extracted by these parsers inhibiting a complete analysis. The missed information is related to method calls of Mozilla components that have been implemented using the Mozilla component model XPCOM which heavily uses C++ templates.

Basically, C++ templates are a mechanism that allows the compiler to statically check the type compatibility of polymorphic functions or classes whose use can be resolved during compile time. Additionally, since the function invocations can be determined at compile time, the C++ compiler can perform many more optimizations than would be available otherwise.

The disadvantage of templates, however, is that in many cases they are hard to write or to understand and that any reverse engineering tool that wants to create a call graph has to understand all the details of C++'s template instantiation mechanism which has been shown to be Turing Complete [17].

In the following, we present three small code examples that are typical uses of C++ templates and explain how currently available tools deal with these examples. We also used these examples for the validation of TUAnalyzer's template analysis capabilities.

Example 1. Figure 1 shows a simplified version of the `find_if` function found in the C++ Standard Template Library [10, 15]. If this function is invoked with an integer pointer as first and second argument and a callback function as third argument, the C++ compiler generates a specialized version of this function that operates on integer pointers and the callback function provided. If it is invoked elsewhere in the program with a pointer to doubles and another callback function, the C++ compiler also generates such a specialization.

```
template <typename I, typename Op>
I find(I begin, I end, Op op) {
  while(begin!=end) {
    if(op(*begin)) break;
    ++begin;
  }
  return begin;
}
```

Figure 1. A Simple Find Example

When we analyzed the previous source code example with Imagix-4D, we identified that Imagix-4D retrieves the template function find and its parameters. However, it misses the call of `op(*begin)`, because it does not interpret the template parameter `Op`. Columbus identified op as a callback function but could not identify which function is actually being called. The quality of results produced by related parsing tools were even worse.

Example 2. We use a simplified version of the C++ STL's distance function shown in Figure 2. This function computes the distance between two elements pointed to by iterators. Depending on the type of iterator a different algorithm can be used. In case of a simple forward iterator, the iterator may only be advanced by one element and hence in order to count the number of elements we have to count how often we can advance the iterator until we have reached the second iterator. In case we have a random access iterator, it is possible to subtract the iterators from each other to compute the number of elements in between.

The first two functions compute the difference for an input iterator and a random access iterator as indicated by the function's third argument. Since the argument is not used within the function, we do not have to specify a name for the parameter. The third function is a generic wrapper that determines the type of iterator it is parameterized with through the `iterator_traits` class which returns for a given iterator the type of iterator. Then, this function creates an object of the corresponding type and lets the compiler statically determine the function to be called through its overload resolution mechanism.

```
template<class I> inline
int dist(I first, I last,
         input_iterator_tag) {
  int n=0;
  while(first!=last) { ++first; ++n; }
  return n;
}

template<class I> inline
int dist(I first, I last,
         random_access_iterator_tag) {
  return last-first;
}

template<class I> inline
int dist(I first, I last) {
  return dist(first,last,typename
  iterator_traits<I>::category());
}
```

Figure 2. Algorithm Selection

```
NS_IMETHODIMP ScrollbarsPropImpl::GetVisible(PRBool *aVisible)
{
  nsCOMPtr<nsIScrollable> scroller(do_QueryInterface(docshell));
  scroller->GetDefaultScrollbarPreferences(
    nsIScrollable::ScrollOrientation_Y, &prefValue);
  ...
}
```

Figure 3. A Code Example Taken from the Mozilla Open Source Web Browser

Analyzing the code snippets with existing analysis tools lacks information about the call to an instantiation of the respective `dist`/3 function template in the `dist`/2 function template because currently available parsing tools do not resolve the type of the third parameter due to template parameter dependencies. Hence they cannot statically determine which `dist` function is being called.

Example 3. XPCOM uses templates for the implementation of a kind of smart pointer (`nsCOMPtr`). This pointer is subsequently used to maintain a reference count of its components. In order to understand function invocations on these pointers, it is imperative that the reverse engineering tool understands the template instantiations performed by the C++ compiler. Figure 3 shows a sample taken from Mozilla's source code.

Imagix-4D correctly identifies the constructor call in order to instantiate the scroller object but it does not correctly identify the call to the `GetDefaultScrollbarPreferences` method because the object returned by the `->` operator depends on the type with which the pointer has been parameterized. Since Imagix-4D does not analyze the interdependencies between argument types and the instantiated templates, it overlooks the information about the type returned by the `->` operator and hence cannot identify the method call as has been confirmed by Imagix-4D engineers. Another problem illustrated by this code is that this code invokes a virtual function that are frequently used within the XPCOM implementation. Hence, our analysis tool not only has to be able to deal with templates but also with the invocation of virtual functions.

3. GCC's Translation Unit

Our first approach to identify the function calls that are being generated through template expansion was to use the object file generated by the compiler. Although this approach worked perfectly fine for identifying function calls generated by templates, using this approach, we were unable to identify the virtual method calls performed by the application such as the one shown in our Mozilla sample code. This is because virtual method calls are invoked through a virtual method table. Identifying the index of the function in the virtual method table and subsequently the function to be invoked on the basis of the disassembled code is very complex. Merging the information that can be obtained using this approach with the information provided by existing tools does not work either. By doing so, we are still missing virtual method invocations on objects which depend on a given template parameter. Hence, in order to extract the complete structure of the underlying program, our approach has to provide support for all of the features also provided by other fact extraction tools. Hence, we looked for a different approach.

Our next step was to look at GCC, one of the best known C++ compilers available today. GCC internally stores the abstract syntax tree (AST) as an über-union (a union that may store any node of GCC's AST) that is hard to understand. GCC, however, allows maintainers to dump its representation of the abstract syntax tree using one of several compiler switches. This functionality has been included to allow for the debugging of GCC itself [14]. Since we did not want to fiddle around with the implementation of GCC code directly we wrote a program that analyzes the dump of GCC's abstract syntax tree.

Using this approach, a C++ program can be easily analyzed by building the program using GCC and adding the necessary compiler switches. This compiles the program and generates files representing GCC's representation of the individual translation units. Finally, our analyzer uses these files and generates FAMIX compliant RSF format. These facts then are integrated into the repository generated by a traditional reverse engineering tool and thereby complete the fact base (i.e. source code model graph).

The interesting GCC compiler switches to supply during the build process of a given program are:

--fdump-translation-unit dumps almost all the information stored within GCC's abstract syntax tree for a given translation unit. It provides all the template declarations and all the template instantiations. Deriving the layout of the virtual method table on the basis of this information is non-trivial but possible. A short excerpt of this tree is given in Figure 4.

```
@1      namespace_decl   name: @2       srcp: <internal>:0
                         dcls: @3
@2      identifier_node  strg: ::       lngt: 2
@3      function_decl    name: @4       type: @5       srcp: find_tmpl.cc:30
                         chan: @6       C              extern
                         body: @7
@4      identifier_node  strg: main     lngt: 4
@5      function_type    size: @8       algn: 64       retn: @9
                         prms: @10
@6      function_decl    name: @11      mngl: @12      type: @13
                         srcp: find_tmpl.cc:26         chan: @14
                         args: @15      extern         body: @16
@11     identifier_node  strg: iszero   lngt: 6
@14     template_decl    name: @26      type: @27      srcp: find_tmpl.cc:18
                         chan: @28      rslt: @29      spcs: @30
                         prms: @31
@26     identifier_node  strg: find     lngt: 4
```

Figure 4. GCC Translation Unit

--fdump-tree-original dumps the AST of each individual method. The advantage of this approach is that a lot of information pertaining to a given method can be extracted from the tree without a deep understanding of GCC's AST. Since GCC, however, only dumps the information that is absolutely necessary to generate the code for the method, only partial information about records is being dumped and one cannot extract a record's virtual method table, unused methods or fields. The switch generates a dump similar to that given in Figure 4 for each function or method defined in the translation unit.

--fdump-class-hierarchy dumps the class hierarchy as well as the virtual method tables of the individual classes. Using this class representation allows developers to infer the methods that correspond to the individual virtual method table elements more easily.

The translation unit shown in Figure 4 shows the typical structure of GCC's representation of a translation unit. It first gives a namespace declaration (node `@1`) which references all the declarations within that namespace through the `dcls` attribute. The declarations itself are linked through `chan` links. This allows us to traverse through all the declaration in a translation unit.

Node `@3`, for instance, shows the declaration of the main function. The function's type is referred to by the `type` and the body by the `body` attribute. To identify all of the function calls or accesses to records, all the statements and expressions inside the body need to be analyzed. For brevity reasons, we omit this discussions for simple function calls and focus our discussion on virtual function calls and templates.

4. Template Analysis

The strength of using GCC's representation of a translation unit becomes clear when C++ templates need to be analyzed. As we will see in the next section, interestingly, it is easier to obtain this information on the basis of GCC's translation units then the information about virtual method calls because they are called indirectly through the virtual method table.

4.1. Function Templates

Within GCC's translation unit, a template declaration is represented as a `template_decl` node. This node contains pointers to the original definition of the template, its parameters and to the instantiations of the template.

Figure 4 shows this node (`@14`) for the `find` function presented in Section 2. The `type` attribute points to a `function_type` node that provides the generic representation of the function's type and the `rslt` attribute points to the function's generic implementation.

The `find` function takes two template parameters: a forward iterator (i.e., a class that supports the comparison, dereference operator, and the pre-increment operator) and a function or class providing the `()` operator (both taking an argument of the type returned by `*begin`).

For analysis purposes, the `find` function poses several challenges. If the analysis tool is unable to interpret template parameters, it will be unable to identify which function will be called by the different instantiations of the function. Another challenge is that if the template parameter `I` is a class implementing the above mentioned operators, we would like to see the operator invocations reported since an

```
@30     tree_list        purp: @51      valu: @52      chan: @53
@52     function_decl    name: @26      mngl: @83      type: @84
                         srcp: find_tmpl.cc:18         args: @85
                         extern         body: @86
@53     tree_list        purp: @87      valu: @88
@88     function_decl    name: @26      mngl: @133     type: @134
                         srcp: find_tmpl.cc:18         args: @135
                         extern         body: @136
```

Figure 5. Instantiated `find` Functions

operator is basically a function. On the other hand if it only represents a pointer type, no user-defined function is called and hence no such call should be reported.

The following code creates two different instances of the `find` function: one parameterized with `<int*,in_range_class<int>>` and the other with `<int*,int(*)(int)>`.

```
int main() {
  int p[8]={17,7,0,4,39,45,-1,7}, *q=0;
  in_range_class<int> op=in_range(2,6);
  q=find(p,p+8,op);
  q=find(p,p+8,iszero);
}
```

The functions instantiated from the above code are shown in Figure 5. These functions are linked in a list of `tree_list` nodes. The `chan` attribute gives a link to the next instantiation, the `valu` attribute points to the actual instantiation, and the `purp` attribute specifies the types with which the function has been parameterized with.

4.2. Class Templates

Records and classes are other interesting programming language constructs that need to be analyzed. GCC does not differentiate between records and classes. The following shows a simple function object (`in_range_class`) that can be used in combination with our `find` function and a helper function (`in_range`) that enables the function object's transparent instantiation (see [15] for details).

```
template <class T>
struct in_range_class {
  T x, y;
  in_range_class(T a, T b): x(a),y(b) {}
  int operator()(int z) {
    return x<z && z<y;
  }
};

template <class T> inline
in_range_class<T> in_range(T a, T b) {
  return in_range_class<T>(a,b);
}
```

GCC's representation of this code is shown in Figure 6. As for function declarations, the template declaration node points to the generic type (`type`), the generic implementation (`rslt`), and the individual instantiations (`inst`) of the record.

Based on the record's generic representation (`@68`), the fields of a given record can be easily identified through the `flds` attribute of the `record_type` node. This node attribute points to a list of `var_decl` nodes which are linked through a series of `chan` attributes. Member functions can be obtained by following the `record_type`'s `fncs` attribute. In case of our function object, we can identify first the record's constructor (`@70`). In this case, it is the default constructor generated (hence, the node is marked as `artificial`) by GCC since none has been provided by us. By following the `chan` attributes one can identify our own constructor, as well as the function call operator `()`.

The instantiations of the class template are structured exactly the same way as its generic counterpart except that the type parameters are replaced with concrete types and generic function invocations with function invocations corresponding to the template parameter types.

5. Inheritance

Another challenge that we have shown in Section 2 is that even virtual function calls may depend on the instantiation of C++ templates. Since traditional tools are unable to analyze template instantiations, they also miss these virtual method calls. Hence, in order to be able to analyze complex projects such as Mozilla, TUAnalyzer not only has to cope with templates but also with the invocation of virtual functions.

Discussing the extraction of virtual function calls on basis of the Mozilla example that we have given in Figure 3 would be rather long due to the additional template instantiations and would not give additional insights into the analysis of C++ programs using our approach. Hence, we use the following more simple example that shows a class hierarchy (similar to the one implemented in TUAnalyzer). This example also allows to explain how inheritance relationships

```
@32     identifier_node  strg: x         lngt: 1
@45     template_decl    name: @67       type: @68       srcp: find_tmpl.cc:2
                         chan: @69       rslt: @70       inst: @71
                         prms: @72
@67     identifier_node  strg: in_range_class            lngt: 14
@68     record_type      name: @70       size: @105      algn: 8
                         struct          flds: @106      fncs: @107
                         binf: @108
@70     type_decl        name: @112      type: @68       srcp: find_tmpl.cc:2
                         artificial      chan: @113
@71     tree_list        purp: @114      valu: @115      chan: @116
@106    field_decl       name: @32       type: @157      scpe: @68
                         srcp: find_tmpl.cc:4            chan: @158
                         public          algn: 1
@107    function_decl    name: @67       type: @159      scpe: @68
                         srcp: find_tmpl.cc:5            chan: @160
                         member          public          constructor
                         pseudo tmpl     args: @161      extern
                         body: @162
@115    record_type      name: @169      size: @8        algn: 32
                         struct          flds: @170      fncs: @171
                         binf: @172
```

Figure 6. Class Template (`@45`) and Instantiations (`@71`)

and virtual function calls can be derived from GCC's representation of a translation unit:

```
class Decl {
  // ...
  virtual const char *getDeclName();
  virtual void dump();
};

class Expr {
  // ...
  virtual const char *getExpr()=0;
  virtual void dump() { /* ... */ }
};

struct VarDecl
: public Decl, public Expr {
  VarDecl(const char *v) : Decl(v) {}
  /* default getDeclName is OK */
  virtual const char *getExpr() {
    return getDeclName();
  }
  virtual void dump() { /* ... */ }
};

int main() {
  // ...
  decl->dump();
}
```

The interesting parts of GCC's representation of this translation unit are shown in Figure 7. The figure shows, how inheritance relationships can be identified using GCC. For instance, the record described by node `@69` shows the base classes specified using the `base` attribute. An interesting thing to note here is that the `base` attribute may occur multiple times.

Figure 7 also shows the `binf` attribute which points to a structure that may allow reverse engineers to derive the layout of the virtual method table. Analyzing this structure is fairly complex since it requires a deep understanding of GCC's internal data structures. Fortunately, GCC can also dump the class hierarchy and the layout of the virtual method tables in a format that can be understood more easily (using the --fdump-class-hierarchy switch). The output for the `VarDecl` class is shown in Figure 10.

As mentioned previously, the extraction of virtual function calls was the most challenging part in the implementation of TUAnalyzer. Within GCC's translation unit, these method calls are stored in a form suitable for the subsequent code generation phase but not suitable for any kind of code analysis purposes. That is, they are invoked indirectly by accessing the class's artifical virtual method table field, computing the function's entry, and by calling the address stored at that entry.

Figure 9 shows GCC's representation of a function call (`@524`). Since this is a virtual function, the `fn` attribute points to an `indirect_ref` node (instead of a `function_decl` node) which points to a plus expression.

```
@53     type_decl        name: @68      type: @69      srcp: virtual_mult.cc:17
                         artificial     chan: @70
@68     identifier_node  strg: VarDecl  lngt: 7
@69     record_type      name: @53      size: @82      algn: 32
                         vfld: @83      base: @84      public
                         base: @85      public         struct
                         flds: @86      fncs: @87      binf: @88
@84     record_type      name: @112     size: @32      algn: 32
                         vfld: @113     struct         flds: @113
                         fncs: @114     binf: @115
@85     record_type      name: @91      size: @111     algn: 32
                         vfld: @116     struct         flds: @116
                         fncs: @117     binf: @118
```

Figure 7. Inheritance Relationships (`@53`)

```
@108    identifier_node  strg: _vptr$Decl              lngt: 10
@109    identifier_node  strg: $vf      lngt: 3
@113    field_decl       name: @108     mngl: @109     type: @110
                         scpe: @84      srcp: virtual_mult.cc:1
                         artificial     chan: @148     public
                         size: @111     algn: 32       bpos: @81
@236    var_decl         name: @291     type: @256     scpe: @28
                         srcp: virtual_mult.cc:28      chan: @292
                         init: @293     size: @111     algn: 32
                         used: 1
@256    pointer_type     size: @111     algn: 32       ptd : @84
@291    identifier_node  strg: decl     lngt: 4
@630    nop_expr         type: @256     op 0: @690
@688    component_ref    op 0: @752     op 1: @113
@690    nop_expr         type: @256     op 0: @753
@752    indirect_ref     op 0: @630
@753    nop_expr         type: @256     op 0: @806
@806    non_lvalue_expr  type: @256     op 0: @236
```

Figure 8. Member Access (`@688`)

```
@63     void_type        name: @77      algn: 8
@524    call_expr        type: @63      fn  : @574     args: @575
@574    indirect_ref     op 0: @629
@575    tree_list        valu: @630
@629    plus_expr        type: @110     op 0: @688     op 1: @689
@689    integer_cst      type: @110     low : 4
```

Figure 9. Virtual Function Invocation (`@524`)

```
Vtable for VarDecl
VarDecl::_ZTV7VarDecl: 9 entries
0  0
4  &_ZTI7VarDecl
8  Decl::getDeclName()
12 VarDecl::dump()
16 VarDecl::getExpr()
20 0ffffff8
24 &_ZTI7VarDecl
28 VarDecl::_ZThn8_N7VarDecl7getExprEv()
32 VarDecl::_ZThn8_N7VarDecl4dumpEv()

Class VarDecl
   size=12 align=4
VarDecl (0x10094280) 0
    vptr=((&VarDecl::_ZTV7VarDecl)+8)
  Decl (0x100942c0) 0
      primary-for VarDecl (0x10094280)
  Expr (0x10094300) 8 nearly-empty
      vptr=((&VarDecl::_ZTV7VarDecl)+28)
```

Figure 10. Class Hierarchy Dump

The first argument to the plus expression is the address of the virtual function table field obtained through a member access shown in Figure 8. The reference to a field of a record is identified by a `component_ref` node. The `op 0` attribute refers to an object whose field is to be accessed and the `op 1` attribute refers to the name of the field which is to be accessed.

The second argument is the immediate value 4, the function's index in the table. This is dependent on the processor architecture. The size of function pointers and the alignment of functions, however, can also be identified by using GCC's representation of the translation unit. Hence, one does not have to know the architecture on which the dump has been generated.

Once the type of the object has been identified through the `component_ref`'s `op 0` attribute we can look up the method to be called on the basis of GCC's class hierarchy dump which is shown in Figure 10. One thing that has to be noted, is that the object's virtual function table pointer points already to the third element of the virtual table dumped by GCC. That is the `Decl::getDeclName()` entry in the case of our example.

6. Validation

We empirically validated TUAnalyzer using the examples presented throughout this paper. Figure 11 shows a simplified version of the RSF data generated by TUAnalyzer. The output has been simplified to make it more readable and reduced to show only the essential parts. Besides the information shown, we also extract classes, their members, and accesses to the fields of a class.

As shown by the output, the method signatures generated are slightly different from those expected by a C/C++ compiler. The output shows also how GCC internally represents constructors and methods. That is, the first parameter of a method is used to pass the parameter. This problem, however, does not affect the analysis of the program and will be dealt with in a future version of TUAnalyzer.

The effort to use and integrate TUAnalyzer to reverse engineer C/C++ programs is rather small: one has to use GCC on the source basis in a regular build and enable the additional compiler switches as described in Section 3. Then our TUAnalyzer is applied on the translation unit of GCC. The output that is generated contains all resolved template definitions and virtual method call information from the source code and is in FAMIX compliant RSF format.

One limitation of the current version of TUAnalyzer is that it only extracts function calls and the access of a class's attributes whereas many of the commercial tools provide other information about a program such as the local variables used in a given routine as well as their types or other such information. This problem, however, can be solved by using our tool along with such commercial tools and by integrating the results of our tool into a fact database generated by a traditional reverse engineering tool and thereby completing the fact base (i.e. source code model graph). Consequently, the quality of the fact base is improved and more detailed analyses are facilitated.

Since many—even commercial—reverse engineering tools fail due to incomplete C/C++ source code parsing and representation, our TUAnalyzer provides an important solution for a broad range of tools that can use its results. For instance, we use grok [3] for querying and analyzing the more complete source code model graphs and Rigi [16] and the Graphviz tool kit [5] for graph visualization.

Our next steps will concentrate on large open source projects such as Mozilla, to evaluate our tool and the underlying technique in detail to distill benefits and constraints of our approach. For this paper, the proof of concept has been described, more evaluations are beyond the scope of this paper.

7. Related Work

Several commercial and publicly available parsing tools exist that address fact extraction from C/C++ source code. There is no doubt that there is at least one parser that handles nearly all C/C++ specific language constructs such as templates—that is the GCC. However, the output produced by GCC cannot directly be used by reverse engineering

a) `find.cc`

```
invokes "int main()" "*(int) find(*(int),*(int),in_range_class<int>)"
invokes "int main()" "*(int) find(*(int),*(int),*(int(int)))"
type "int main()" "Function"
invokes "*(int) find(*(int),*(int),*(int(int)))" "*(int(int))"
type "*(int) find(*(int),*(int),*(int(int)))" "Function"
invokes "*(int) find(*(int),*(int),in_range_class<int>)"
        "int operator(*(in_range_class<int>),int)"
type "*(int) find(*(int),*(int),in_range_class<int>)" "Function"
```

b) `dist_tmpl.cc`

```
invokes "int main()" "int dist(*(int),*(int))"
type "int main()" "Function"
invokes "int dist(*(int),*(int))" "int dist(*(int),*(int),random_access_iterator_tag)"
type "int dist(*(int),*(int))" "Function"
```

c) `virtual_mult.cc`

```
invokes "int main()" "void __comp_ctor(*(VarDecl),*(char))"
type "int main()" "Function"
invokes "int main()" "void dump(*(Decl))"
type "void __comp_ctor(*(VarDecl),*(char))" "Method"
type "void dump(*(Decl))" "Method"
```

Figure 11. Output of TUAnalyzer

tools. Furthermore, not all the information produced by the GCC is needed.

Imagix-4D [9] is a reverse engineering tool that concentrates on browsing, navigating, and analyzing C/C++ source code. The parser of Imagix-4D derives source code facts of reasonable quality. As we have explained, however, regarding C/C++ templates the parser lacks of important information about method/function calls which is due to the inability of interpreting template parameters. This drawback is handled by TUAnalyzer.

A related approach that uses the GCC translation units to extract C/C++ source code facts is CPPX [2]. CPPX derives and generates parsing results in the Graph eXchange Language (GXL) [8] format. Although, the resulting graphs include information about templates, the extraction of this information and virtual method call facts that conform to a fact meta model such as FAMIX or Datrix is not yet supported. Hence, the user has to reconstruct this information from the output graph in a way similar as presented in this paper.

Another tool that takes a similar approach is XOGASTAN [1]. A drawback of XOGASTAN is, however, that in its current stage it only converts GCC's translation unit dump file into an XML representation. Its analysis capability of the abstract syntax tree itself, however, is limited.

The approach of Fowler et al. presented in [11] also is based on GCC. It uses a modified version of bison (the parser generator used by GCC) that generates an XML enriched version of the compiler's parse tree. This helps tools to understand the structure of C/C++ programs. However, in order to perform analyses tasks such as analyzing the call or dependency graph, one has to analyze the whole GCC parse tree. To filter out information from the parse tree Fowler et al. introduced the gccXfront tool [7]. However, the filtering capabilities of this tool are limited and do not take into account template instantiations and virtual method calls.

Yet another tool taking this approach is GCC_XML [6]. GCC_XML derives an XML representation of the class, function, and namespace declarations that is easier to parse but does not take function/method bodies into account. Consequently, GCC_XML misses all call graph information that is needed for a detailed analysis of software systems.

8. Conclusions

Currently available reverse engineering tools such as Imagix-4D are only able to indicate an invocation through a template argument but are unable to resolve the function to which the invocation resolves during compilation time. As we have shown in this paper, our approach which we have implemented in TUAnalyzer solves this problem by analyzing GCC's internal representation of translation units and class hierarchies.

We have based TUAnalyzer on the GCC compiler since the compiler has to generate the machine code for each individual template instantiation. These instantiations can also be found in GCC's representation of a translation unit. We have described, how our tool makes use of this data to resolve template instantiations and virtual method calls. TUAnalyzer produces FAMIX compliant RSF format as output such that other tools and approaches can exploit our template analysis. For our reverse engineering tasks, we feed these results into tools such as Rigi or Graphviz for complete call and dependence graph visualization. In addition, the output format enables other approaches to understand GCC's representation of a translation unit and hence allow to build upon our template analysis results for their further analysis purposes.

We also discussed how large C++ applications such as the Mozilla open source browser or the C++ Standard Template Library, are using templates as well as the limitations of the currently available tools. We have used examples taken from these systems to validate our claim that TUAnalyzer template analysis capabilities are superior to those found in other reverse engineering tools. The validation results demonstrated that our tools is indeed able to identify invocations depending on template arguments.

In the future, we plan to use our tool to analyze the structure of the entire Mozilla web browser which makes heavy use of C++ templates and evaluate how our tool performs when used in combination with a large software project and to demonstrate how source code analysis can benefit from our template analysis capability.

References

[1] G. Antoniol, M. Di Penta, G. Masone, and U. Villano. XOgastan: XML-oriented GCC AST analysis and transformation. In *Proceedings of the Third International Workshop on Source Code Analysis and Manipulation (SCAM'03)*. IEEE, 2003.

[2] T. R. Dean, A. J. Malton, and R. C. Holt. Union schemas as a basis for a C++ extractor. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*. IEEE, Oct. 2001.

[3] H. Fahmy and R. C. Holt. Software architecture transformations. In *Proceedings of the International Conference on Software Maintenance*, pages 88–96, San Jose, CA, Oct. 2000. IEEE.

[4] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus—reverse engineering tool and schema for C++. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*. IEEE, 2002.

[5] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software-Practice & Experience*, 30(11):1203–1233, Sept. 2000.

[6] GCC_XML. `http://www.gccxml.org/HTML/Index.html`, May 2004.

[7] M. Hennessy, B. A. Malloy, and J. F. Power. gccXfront: Exploiting gcc as a front end for program comprehension tools via XML/XSLT. In *International Workshop on Program Comprehension (IWPC'03)*. IEEE, 2003.

[8] R. C. Holt, A. Walter, and A. Schürr. GXL: Toward a standard exchange format. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, pages 162–171. IEEE, Nov. 2000.

[9] Imagix Corporation. Imagix-4D 4.3.3. `http://www.imagix.com/`, May 2004.

[10] ISO/IEC. *ISO/IEC14882: Programming Languages—C++*, 1st edition, July 1998.

[11] J. F. Power and B. A. Malloy. Program annotation in XML: A parse-tree based approach. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*. IEEE, 2002.

[12] Software Composition Group, University of Berne. *The FAMIX 2.0 Specification*, 2.0 edition, Aug. 1999. `http://www.iam.unibe.ch/~scg/Archive/famoos/FAMIX/`.

[13] Source navigator 5.1.4. `http://sourcenav.sourceforge.net/`, June 2003.

[14] R. M. Stallman. *Using GCC: The GNU Compiler Collection Reference Manual*. GNU Press, Oct. 2003.

[15] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[16] S. R. Tilley, K. Wong, M.-A. D. Storey, and H. A. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, Dec. 1994.

[17] T. L. Veldhuizen. C++ templates are Turing Complete. Technical report, Indiana University, 2003.

[18] The Mozilla open source web browser. `http://www.mozilla.org/`, May 2004.