# Chapter 5

# g$^4$re – Tool Chain for Reverse Engineering C++ Programs

In this chapter we present the design and implementation of the g$^4$re tool chain [Kraft et al. 2005a; 2006b]. g$^4$re is a *tool chain*, because it is constituted by applications and libraries that are used either individually, or in concert. We designed g$^4$re with a GXL-based pipe-filter architecture; each constituent application or library in the chain takes, as input, the output of the preceding application or library in the chain. An important benefit of this architecture is that g$^4$re consists of a set of loosely coupled, reusable modules: the *ASG module*, the *schema and serialization modules*, the *transformation module*, the *linking module*, and the *API module*. We wrote all modules in ISO C++.

In Section 5.1 we present the architecture of g$^4$re. In this section, we also include an overview of the CppInfo schema. In Section 5.2 we present a sample usage of g$^4$re.

## 5.1 Architecture

In Figure 5.1, we provide an overview of the packages in the tool chain.We illustrate implementation artifacts, which we indicate with bold text, and third party libraries, which we indicate with italic text, at the left of the figure. We illustrate the *ASG module*, generic, as a package in the large g4re package at the right of the figure, and describe it in Section 5.1.1.

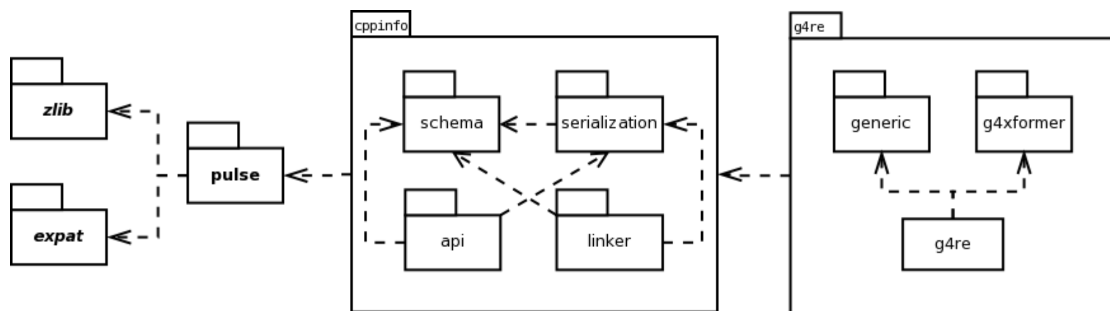


Figure 5.1: Overview of g$^4$re. *Dashed lines represent "use" dependencies. Bold text indicates an implementation artifact. Italic text indicates a third party library.*

We illustrate the *schema and serialization modules*, schema and serialization, as packages in the large cppinfo package at the center of the figure, and describe them in Section 5.1.2. We illustrate the *transformation module*, g4xformer, as a package in the large g4re package at the right of the figure, and describe it in Section 5.1.3. Finally, we illustrate the *linking module* and the *API module*, linker and api, as packages in the large cppinfo package at the center of the figure, and describe them in Sections 5.1.4 and 5.1.5, respectively.

### 5.1.1 The ASG module: generic

In the generic package, we provide parsing, storage, traversal, and serialization facilities for working with the GENERIC ASG representation of *gcc*. The input to this package is a `tu` file, or a GXL encoding of a `tu` file. The output of this package is a gzipped GXL encoding of the input file, or an in-memory representation of the GENERIC ASG.

We implemented two parsers: a `tu` file parser that uses a scanner generated by *flex*, and a GXL file parser that uses the *expat* XML parsing library and the *zlib* compression library via the pattern and utility library with standard extensions (*pulse*). We also implemented a simple node list representation for storage of the parsed ASG, and several parameterized methods for traversing the leftmost child right sibling (LCRS) tree that underlies the ASG. Finally, we implemented an extensible serialization facility that we use to create GXL encodings of `tu` files.

The first parser we wrote provides functionality to parse a `tu` file and to store the corresponding ASG. After parsing a `tu` file, we perform a series of transformations on the stored ASG to remove extraneous information and to make it more suitable for reverse engineering tasks. In particular, we:

- remove fields that store internal information used by the *gcc* back end,
- mark methods as static if their parameter lists do not contain a *this* pointer,
- mark methods as const if their parameter lists contain a const *this* pointer,
- remove the *this* pointer from all method parameter lists.

We use this parser in conjunction with our serialization facility to create GXL instances of tu files.

The second parser we wrote provides functionality to parse a GXL file or gzipped GXL file and to store the corresponding ASG. Three advantages of this parser over the tu parser are:

1. reentrance,
2. the lack of post-parse transformation overhead,
3. the compression rate is higher for GXL files than for tu files.

We use the tu parser (in conjunction with our serialization facility) to create the GXL file(s) accepted by this parser; thus, there is a one-time cost associated with its use.

### 5.1.2 The Schema and Serialization Modules: schema and serialization

In the schema package, we provide a class library that implements the CppInfo schema[1] for the ISO C++ programming language. In the current implementation, we provide 72 classes, 42 of which are concrete, that provide information about C++ language elements. Language elements include declarations, such as classes (including class templates and class template instantiations); namespaces; functions (including function templates and function template instantiations); and variables, statements (including control statements and exception statements), and some expressions.

In the serialization package, we provide serialization facilities for working with instances of the schema representation of C++. We implemented a parser to read GXL encodings; gzipped files are also accepted. We implemented visitor [Gamma et al. 1995] classes to write gzipped GXL encodings. We used C++ templates to allow the package to read and write both individiual and linked instances of the schema representation.

[1] We describe the CppInfo schema in more detail in Section 4.2.

### 5.1.3 The Transformation Module: g4xformer

In the g4xformer package, we provide an implementation of the transformation from the ASG representation that we provide in the generic package to the representation that we provide in the schema package. The input to this package is a `tu` file, or a GXL encoding of a `tu` file; gzipped files of either type are also accepted. The output of this package is a gzipped GXL encoding of the instance of the schema representation that corresponds to the GENERIC ASG in the input file.

We implemented the transformation in three passes. In the first pass, we traverse the generic ASG in program order, and create the *core* of the instance of the schema representation. The core consists of all declaration, declarator, and statement nodes, as well as structural edges. In the second pass, we adorn the core with edges that indicate the use of a type; these edges include inheritance edges. In addition, in the second pass, we build all cv-qualified types. We also resolve uses of bound template template parameters to their template declarations.[2]. Finally, in the third pass, we adorn the graph that results from the second pass with edges that indicate uses of expressions, including function calls[3]

### 5.1.4 The Linking Module: linker

In the linker package, we provide an implementation of our linking algorithm. The input to this package is a set of GXL encodings of instances of the schema representation for all C++ translation units in a program; gzipped files are also accepted. The input files need not be created by the g4xformer package. The output of this package is a gzipped GXL encoding of the linked, or *unified*, instance of the schema representation for all C++ translation units in the program.

Programs written in C++ consist of multiple files, both header and source. A *C++ translation unit* consists of a source file and all files that it includes, either directly or transitively. A C++ compiler, such as *gcc*, operates on a single translation unit at a time;

[2]This task is not needed for compilation, and is not performed by *gcc*

[3] Calls to virtual functions are designated as such in `tu` files, but sets of possible targets are not identified. These sets are available via the *gcc* compiler flag `-fdump-class-hierarchy`.

the generated object code for all translation units in a program is linked by the system linker, e.g., *ld* on Unix systems. A C++ reverse engineering tool, such as g$^4$re, also operates on a single translation unit at a time; however, the generated output is not object code, but rather a program representation such as an ASG.

We perform linking $n-1$ times, where $n$ is the number of translation units, when $n$ is greater than one. Otherwise, we perform linking one time. We achieve linking by performing a traversal of the most recently constructed instance of the schema representation. We add or append a schema class instance to the unified instance of the schema representation if the class instance does not exist, or is incomplete, in the unified instance. A schema class instance is *incomplete* if it is missing a required element (as defined by the CppInfo schema) or contains another incomplete instance. Using our definition of incomplete, we resolve function declarations to their corresponding definitions.

There is a special case for linking function parameters. A function parameter from a function declaration (prototype), is not always identical to the corresponding function parameter from the function definition. A function parameter may only have an initial value in a function declaration. In addition, the name of the function parameter may differ, e.g., anonymous function parameters are commonly used in header files.

### 5.1.5 The API Module: api

In the api package, we provide an abstract class that defines the interface for an API that provides access to information about language elements in a C++ program. In addition, we provide a concrete implementation of the API. The input to this package is a GXL encoding of a linked instance of the schema representation; gzipped files are also accepted. The input files need not be created by the linker package. The output of this package is an API, an in-memory representation of the linked instance that may be queried by a user program.

We designed the api package to provide a clear and flexible interface. We provide two points of access. The first point of access is a pointer to the global namespace, from which a user can traverse the ASG that underlies the API. We provide iterator classes, as well as an

abstract visitor class, to use when accessing the API in this fashion [Gamma et al. 1995]. The second point of access is a collection of lists that each contain instances of a particular schema class. We provide, in two forms, the lists for Namespace, Class, Enumeration, Enumerator, Function, Variable, and Typedef. The first form provides all instances of the particular schema class; the second form provides *filtered* instances of the particular schema class. *Filtered* instances are determined by user-provided *filter lists* that contain the names of source files from which schema class instances should be ignored. We provide a script that generates filter lists.

## 5.2 Sample Usage

In Figure 5.2 we provide an overview of API usage. We illustrate a GXL file containing a linked instance of the schema representation at the top of the figure. Next, we illustrate a sample user program, metrics, that instantiates then queries the API to compute metrics. We illustrate the API, the abstract class cppinfo::api::Interface, in the middle of the figure. Finally, we illustrate filter lists at the bottom of the figure.

The user program instantiates the API with the name of the GXL file; when the API is instantiated, it reads the filter lists. The user program queries the instantiated API to perform a reverse engineering task, such as a program analysis. In Section 5.2.1 we describe the process of acquiring the GXL files needed to instantiate the API. In Section 5.2.2 we present a sample user program that instantiates and queries an API to perform a simple program analysis.

### 5.2.1 Input

In Figure 5.3 we illustrate the process of using *gcc*, and optionally g$^4$re and/or *gzip*, to create a set of files that contain instances of the GENERIC schema. We show the input, a C++ source file, at the left of the figure. We show the output, a set of files to transform, at the right of the figure (see Subsection 5.1.3 for details). This set may contain any combination

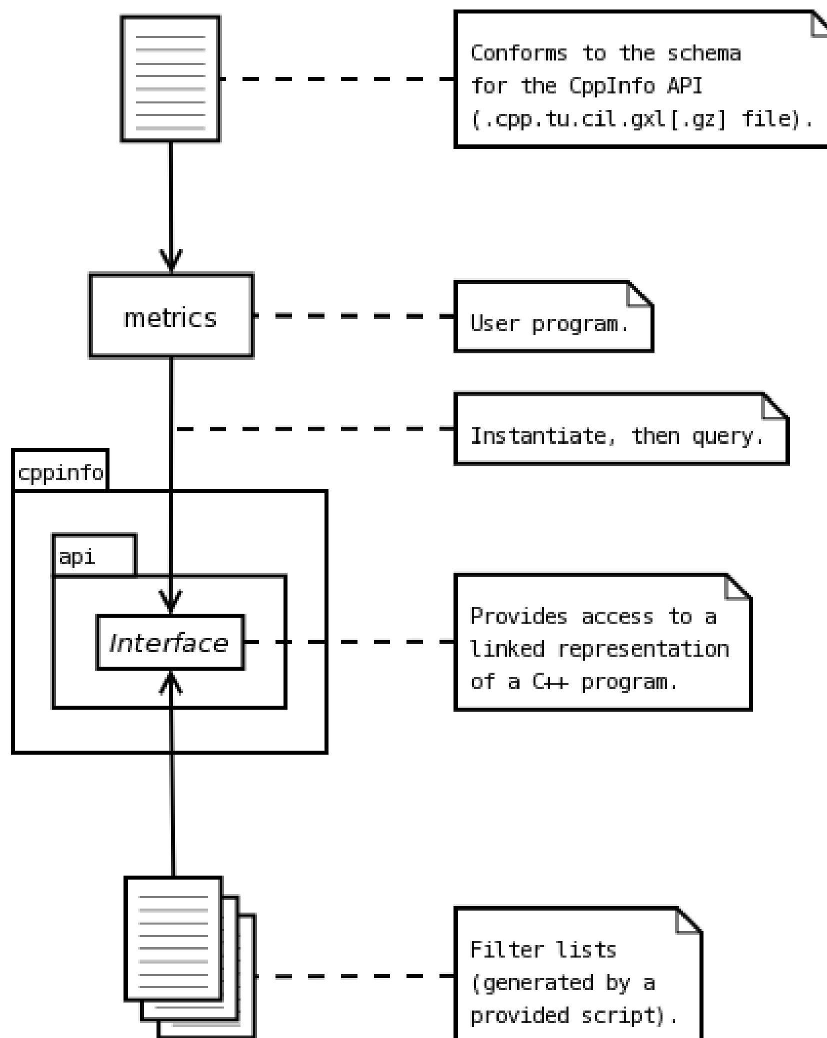


Figure 5.2: Overview of API usage. *Solid, directed lines show input, unless otherwise noted. Dashed lines show notes.*

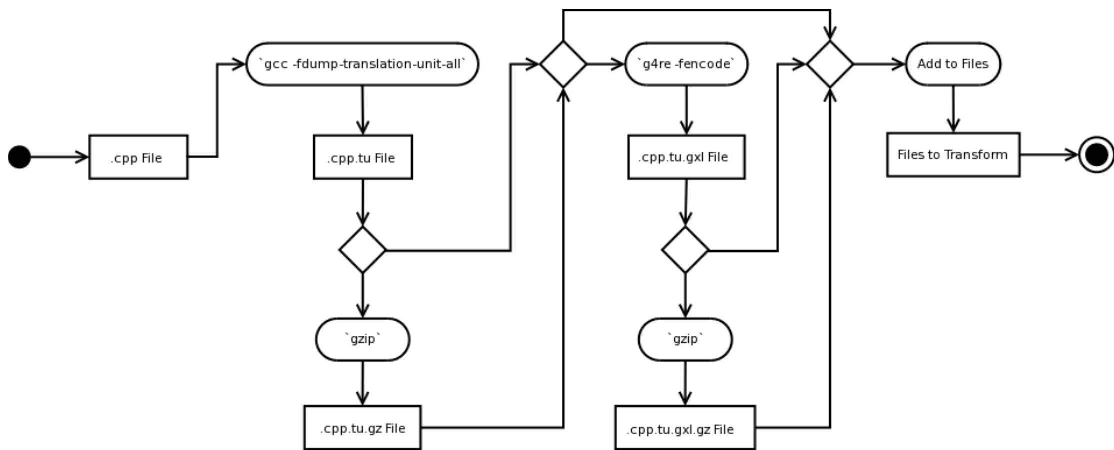


Figure 5.3: UML Activity Diagram for Transformer Input. *The process of creating a set of files to transform.*

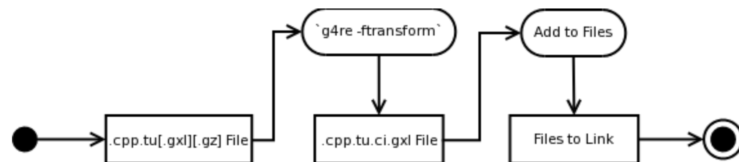


Figure 5.4: UML Activity Diagram for Linker Input. *The process of creating a set of files to link.*

of the four possible encodings of the input.

We show the use of the *gcc* command line flag `-fdump-translation-unit-all` to obtain a plain text representation of the GENERIC instance for each translation unit in a program. We show the creation of these representations, known as `tu` files, in the upper left of Figure 5.3. We use `tu` files rather than hard-coding our solution into the *gcc* source code. This provides flexibility, and fits our theme of exchange among reverse engineering tools. In the upper right of the figure, we show the optional use of the g$^4$re command line flag `-fencode` to obtain, for each `tu` file, a GXL encoding of an instance of the GENERIC schema. At the bottom of the figure, we show the optional use of *gzip* to compress either a `tu` file, or a GXL instance,

In Figure 5.4 we illustrate the process of using g$^4$re to create a set of GXL files that contain instances of the CppInfo schema. We show the input, the set of files to transform

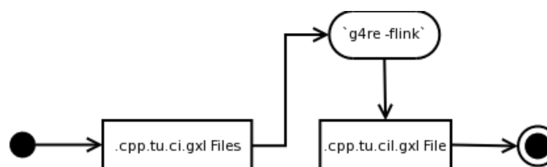


Figure 5.5: UML Activity Diagram for API Input. *The process of creating a file for use with the API.*

(obtained as shown in Figure 5.3), at the left of the figure. We show the output, a set of files to link, at the right of the figure (see Subsection 5.1.4 for details).

We show the use of the g$^4$re command line flag `-ftransform` to obtain, for each GENERIC instance, a GXL encoding of a temporary instance of the CppInfo schema. We show the creation of these temporary instances, which use string encodings of the unique names for the contained instances of CppInfo classes, in the center of Figure 5.4. We omit showing the optional use of *gzip* in this figure.

In Figure 5.5 we illustrate the process of using g$^4$re to create a GXL file that contains a linked instance of the CppInfo schema. We show the input, the set of files to link (obtained as shown in Figure 5.4), at the left of the figure. We show the output, a GXL encoding of the linked instance of the CppInfo schema, at the right of the figure (see Subsection 5.1.5 for details).

We show the use of the g$^4$re command line flag `-flink` to obtain, for a set of temporary instances of the CppInfo schema, a GXL encoding of the linked instance of the CppInfo schema. We show the creation of the linked instance, which uses unique integers to identify the contained instances of CppInfo classes, at the right of Figure 5.5. We omit showing the optional use of *gzip* in this figure.

### 5.2.2 Usage

In Source Listing 5.1, we list a small C++ program that consists of ten classes. We list two *root classes*, Shape and Visitor, on lines 1 and 7, respectively. Root classes do not have base classes. We list three *interior classes*, Rectangle, ComputationVisitor, and SerializationVisitor,

```
 1  class Shape { };
 2  class Circle    : public Shape { };
 3  class Rectangle : public Shape { };
 4
 5  class Square : public Rectangle { };
 6
 7  class Visitor { };
 8  class ComputationVisitor   : public Visitor { };
 9  class SerializationVisitor : public Visitor { };
10
11  class AreaComputationVisitor      : public ComputationVisitor { };
12  class PerimeterComputationVisitor : public ComputationVisitor { };
13
14  class XmlSerializationVisitor : public SerializationVisitor { };
```

**Source Listing 5.1:** Sample C++ program. *Two disjoint inheritance hierarchies that consist of ten classes.*

on lines 3, 8, and 9, respectively. Interior classes have one or more base classes, and one or more derived classes. Finally, we list five *leaf classes*, Circle, Square, AreaComputationVisitor, PerimeterComputationVisitor, and XmlSerializationVisitor, on lines 2, 5, 11, 12, and 14, respectively. Leaf classes have one or more base classes, but no derived classes.

```
 1  void countClasses ( const cppinfo::api::Filename_t& filename ) {
 2    using cppinfo::api::Interface;
 3    using cppinfo::api::LinkedInterface;
 4     Interface* interface = new LinkedInterface( filename );
 5     unsigned root = 0, interior = 0, leaf = 0;
 6     cppinfo::ConstClassPtrListIterator_t i = interface->getClasses().createIterator();
 7     while ( true == i->isValid() ) {
 8        const cppinfo::ConstClassPtr_t c = i->getCurrent();
 9        unsigned baseCount = c->getBaseClasses().size();
10        unsigned derivedCount = c->getDerivedClasses().size();
11        if ( 0 == baseCount ) {
12           ++root;
13        }
14        else {
15           if ( 0 < derivedCount )
16              ++interior;
17           else
18              ++leaf;
19        }
20        i->moveNext();
21     }
22     delete i;
23  }
```

**Source Listing 5.2:** Sample user program. *A simple program analysis that counts the number of root, interior, and leaf classes.*

In Source Listing 5.2, we list a C++ function that instantiates and queries an API instance to compute the number of root, interior, and leaf classes in a C++ program. We

list the function declaration on line 1, where the parameter filename denotes the input program (see Subsection 5.2.1 for details). We list the API instantiation on line 3, where the filename is passed to the constructor of class cppinfo::api::LinkedInterface. On line 5, we use the list point of access provided by the API to obtain an iterator that accesses each class in the input program[4]. Finally, we list a *while* loop on lines 6–20 that computes the number of root, interior, and leaf classes.

[4]The list contains all `ClassNonTemplate`, `ClassTemplate`, and `ClassTemplateInstantiation` instances. A trivial addition to the loop is required to exclude instances of one or two of these classes.