

# Calculating Metrics from Large C++ Programs

István Siket, Rudolf Ferenc

Department of Software Engineering, University of Szeged, Hungary  
e-mail: {siket|ferenc}@inf.u-szeged.hu

## Abstract

In this work we present a new method called *compiler wrapping* for extracting information from the source code of large software systems written in the C++ language. This new method can be used without having to modify the analyzed source code in any way. With the extracted information we can calculate different object oriented metrics and characterize the analyzed system. For source code analysis and metrics calculation we employed the Columbus reverse engineering framework. To demonstrate the operability of our new approach we tested it on the open source internet suite Mozilla and found it very effective in obtaining the desired metrics.

**Categories and Subject Descriptors:** Applied Informatics

**Key Words and Phrases:** Fact extraction, reverse engineering, compiler wrapping, schema, C, C++, Columbus, CAN, CANPP

## 1 Introduction

A critical issue in large-scale software development and maintenance is the rapidly growing size and complexity of software systems. Due to this rapid growth it is difficult to measure the various parameters of the source code of large programs. Applying different program metrics can help us characterize object oriented programs more precisely, which means that we can express certain properties of the program's source code.

However, obtaining metrics in practice is not so easy because, if we want to calculate them for a large program, we first have to collect the necessary data from the source. We refer to this data as facts about the source code. By *fact* we mean any information that describes different properties of the subject system. One fact might be, for instance, the size of the code. Another fact might be whether a class has base classes. Actually any information that helps us describe source code in some way is called a fact here. Fact extraction is a process which defines different steps that describe the way how facts about the source code can be obtained. These steps include the acquisition of project/configuration information, the analysis of the source files with analyzer tools, the creation of some kind of representations of the extracted facts, the merging of these representations and different processing tasks performed on this merged representation to enable the actual use of the collected information (like calculating metrics).

Basically, a task similar to traditional compilation must be done. The source files must be analyzed each in turn and the desired properties of the code must be collected together. We used the Columbus framework [2] to analyze the source files and collect the facts. But real-world systems may contain several thousands of source files so doing this manually is not really feasible. Our aim was to develop a method which figures out which files have to be analyzed and how they relate to each other. In most cases this information is stored in so-called makefiles. Unfortunately, most makefiles are so complex that it is not feasible to analyze them (because they contain not only the source files and their parameters but a lot of other information needed for compilation). During compilation command line instructions are created according to these makefiles which coordinate the compilation steps. Our idea was to exploit this information by “wrapping” the compiler. This means that when the compiler is run, our program will start instead of it. This way we can get all necessary information about the system and the analysis can be done automatically.

In the next section we describe the Columbus framework in detail. In Section 3 we explain what “wrapping” means, how it works and we also mention some difficulties that we encountered during this development and show how we overcame them. Finally, in Section 4 we show how we applied this method to the Mozilla internet suite [4, 5, 6]. We present only several metrics (without any conclusion) which characterize Mozilla and help us to get a picture about its proportions.

## 2 The Columbus Framework

*Columbus* [2] is a reverse engineering framework that has been developed in cooperation between the University of Szeged, the Nokia Research Center and FrontEndART [3]. The main motivation behind developing the Columbus framework was to create a toolset which supports fact extraction and provides a common interface for other reverse engineering tasks as well.

The main tool is called *Columbus REE* (Reverse Engineering Environment), which is the graphical user interface shell of the framework. All C++ specific tasks are performed by different plug-in modules of the REE. Some of these plug-in modules are present as basic parts, but the REE can be extended to support other languages and reverse engineering tasks as well. The framework contains several command line tools, which actually do the C++-specific tasks like the analysis of the source code and the processing of the results. In the following we will briefly present what they are and do.

*CANPP* (C/C++ ANalyzer-PreProcessor) is a command line program for analyzing C/C++ preprocessing-related language constructs and for preprocessing the code. The input is a C/C++ source file with various settings (like include paths and macro definitions), and the outputs are the preprocessed file and the built-up instance of the Columbus Schema for C++ Preprocessing [7] of the source file. By *schema*, we mean a description of the form of the data, in terms of a set of entities with attributes and relationships. A *schema instance* is an embodiment of the schema which models a concrete software system (or part of it).

*CAN* (C++ ANalyzer) is a command line program for analyzing C++ code. The input of CAN is one complete compilation unit (a preprocessed source file) and the output is the built-up instance of the Columbus Schema for C++ [1] of the analyzed unit. Besides ANSI C++, CAN supports the Microsoft dialect used in Visual C++, the

Borland dialect used in C++ Builder and the GCC dialect used in g++.

*CANLink* (CAN Linker) is a schema instance linker tool. Similar to compiler linkers, it merges the instances of the Columbus schemas into a larger instance. So C++ entities that logically belong together (e.g. libraries and executables) are grouped into one instance. These merged instances have the same format as the original instances, so they can be further merged into one single schema instance to represent the software system as a whole.

*CANFilter* (CAN Filter) is a GUI program that makes the filtering of the (linked) schema instances possible visually. The filtered instances have the same format as the original instances.

*CAN2\**. With the help of these command line tools the (filtered) schema instances built from the extracted facts can be further processed. Some of these procedures apply transformations on the instances to convert them into other formats in order to promote tool interoperability, while others do different computations on the instances such as calculating metrics, recognizing design patterns and code auditing. In this paper we used the *CAN2Metrics* tool for calculating the necessary metrics.

*CANGccWrapper toolset*. The GCC compiler-wrapper toolset supports our novel compiler wrapping technique as described in the next section.

### 3 Compiler Wrapping

The source code of a software system is usually logically split into several files and these files are arranged into folders and subfolders. Furthermore, different preprocessing configurations can apply to them. In this section we deal with the case when the information on how these files are related to each other and what settings apply to them are stored in *makefiles* (used by the *make* tool). An important issue that we addressed was to not change anything in the subject system (not even the makefiles). The technique described in the following successfully deals with this issue. It was tested with the GCC compiler in Linux environment, but the idea is applicable as well to other compilers and operating systems.

The make tool and the makefiles represent a powerful pair for configuring and building software systems. Makefiles may contain not only the references to files to be compiled and their settings but may also contain various commands like invoking external tools. A typical example is when the source file to be compiled is generated on-the-fly from IDL descriptions by another tool. These powerful possibilities are a headache for reverse engineers because every action in the makefile must be somehow simulated in the reverse engineering tool. This may be extremely hard or even impossible in some circumstances.

We approached this problem from the other end and solved it by “wrapping” the compiler. This means that we temporarily hide the original compiler by a wrapper toolset. This toolset consists of several scripts and a program called *CANGccWrapper*. Among the scripts there is a key one called *CANGccWrap*, which is responsible for hiding the original compiler by changing the PATH environment variable. Actually, all the user has to do is to run this script. The script inserts the path of the folder in which the other scripts can be found to the beginning of the PATH environment variable. The names of the other scripts correspond to the different executable files of the compiler (for instance gcc, ar and ld). If we want to execute a program, the operating system searches for it in the folders given in the PATH variable in the same order. This means if the original

compiler should be invoked, our wrapper script will start instead of it because it appears first in the PATH variable. If we do not want to use the wrapper anymore it can be simply switched off. The other scripts are quite similar to each other, the only difference being that they “hide” different tools of the compiler.

The scripts first execute the original compiler tool (for instance g++) with the same parameters and in the same environment so the output remains the same. Hence, we do not notice that not the original compiler was called originally. The scripts also examine whether the original program terminated normally or not and they terminate with the same value. This is very important because complex systems usually run different kinds of tests before the compilation, which determine the compiler capabilities and environment settings. They usually do this by trying to compile small programs containing the issues needed to be tested and examine the termination of the compiler. If the scripts do not take into account the results of the compiler and always terminate normally even when the compilation failed, say, the compilation will be misled and this will probably cause problems later during compilation.

After calling the original compiler the scripts also call the program CANGccWrapper, which will in turn call the corresponding analyzer tool (CANPP, CAN or CANLink). Since the parameters of the CANPP/CAN/CANLink tools are not the same as the compiler’s, they cannot be easily called directly from the scripts. Another problem is that we use different tools for different tasks (for instance CANPP can be used only for pre-processing related issues), while gcc can be used for different purposes like preprocessing, compiling and linking depending on with which parameters it is invoked. So we have to examine the parameters to choose the tool(s) which must be called (for instance “gcc -E ...” means “do not compile the file just preprocess it”). The CANGccWrapper program deals with such problems. The scripts call CANGccWrapper with the same parameters as the compiler and CANGccWrapper will call the required analyzer tools.

CANGccWrapper first examines which tool of the compiler was called because the parameters will be further examined according to this. In different cases it focuses on different parameters, we will only describe its working in the case of gcc because this is the most complex (it can be used for preprocessing, compiling and linking as well). For instance, if gcc is used for both compiling and linking it first preprocesses the files, compiles them and finally links them together, so CANGccWrapper has to simulate the same steps. Since the parameters can be grouped according to which tool uses them, CANGccWrapper has to group them similarly as well. We examine only the most frequently used parameters for all the three tools but it can be easily extended. CANGccWrapper collects mainly the “define” and “include” parameters for CANPP, “libraries” for CANLink and several others for CAN. Where necessary, CANGccWrapper modifies the relative directory paths to full paths to avoid potential ambiguities.

Now we will describe how our fact extraction process works in practice when using the CANGccWrapper toolset. It consists of five consecutive steps as can be seen in Figure 1.

### **Step 1: Acquiring project/configuration information**

In this paper we use our wrapper technology for fact extraction and acquiring project/configuration information is done from makefiles. This is implicitly handled by our wrapper toolset because it integrates itself into the usual build process as explained earlier.

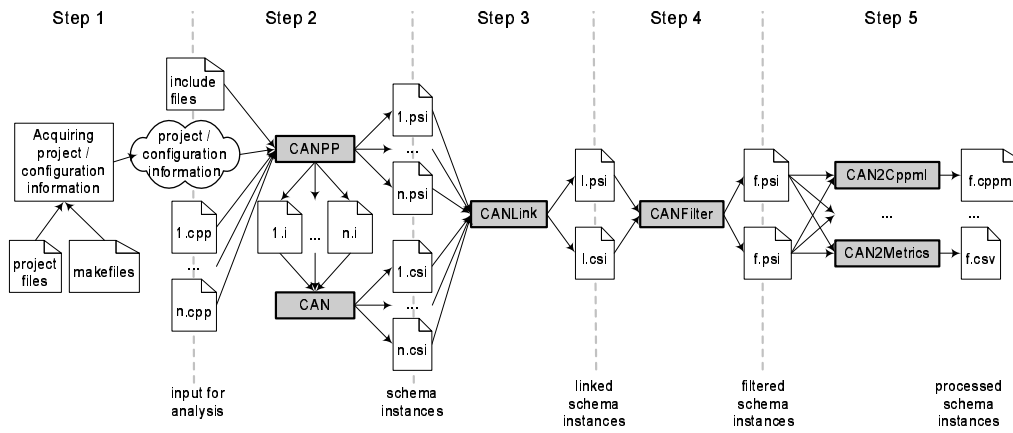


Figure 1: The Columbus fact extraction process

## Step 2: Analysis of the source code – creation of schema instances

In this step the input files are analyzed one at a time and the schema instances are created. First, CANPP preprocesses the input files and then CAN analyzes the preprocessed files and extracts C++ language related information. Both tools create the corresponding schema instances and save them to appropriate files. The CANPP/CAN tools are invoked through the CANGccWrapper program when the source files are compiled (as described previously). One of the issues which we had to deal in this step was that the outputs of CANPP (the preprocessed file and the corresponding schema instance) are not present in the makefiles of the real build process. The first one is needed immediately by CAN, so there is no problem in handing it over, but the schema instances are needed later in step 3 when they must be linked together. In that step only the compiled (object) and linked file names are available in the parameters and we had to find the files containing the associated schema instances. We overcame this problem by using the same file name for the outputs but with different extensions. These files are always stored next to the original output (in the same directory) so that they can be found in step 3.

CANGccWrapper first calls CANPP with the appropriate parameters as we mentioned earlier. The input of CANPP is a source file and the needed header files. CANPP analyzes and preprocesses it and saves the extracted schema instance to a file with the extension “.psi” and the preprocessed file to a file with the extension “.i”. Afterwards, CANGccWrapper invokes CAN with the appropriate parameters and input/output files. CAN uses the preprocessed file created by CANPP, analyzes it and saves the extracted schema instance to a file with the extension “.csi”.

## Step 3: Linking of schema instances

Similar to a real compiler linker that links object files, CANLink links schema instances (which logically belong together) into a merged schema instance that represents a logical unit (subproject) of the system. These merged schema instances can of course be further merged into a single one to represent the whole software system at the same time.

We have two kinds of schema instances as inputs. These are linked in parallel with the linking of the regular object files. CANGccWrapper invokes CANLink to link these schema instances which correspond to the object files linked at this step. As a result there will be only several files which contain all the extracted information that belong to the different subsystems of the subject system. The function of the wrapper ends here in step 3, where all the extracted information is available for further processing.

The following two steps have no equivalents in the traditional compilation process but they help us in handling the extracted facts (the number of facts can be very large as we shall see later) and putting them in a form which is the most suitable for us.

#### Step 4: Filtering the schema instances

Since the schema instances can be quite large and can contain an unmanageable amount of information or we may be simply only interested in certain properties of the system, there is a possibility of filtering the extracted schema instances. Then we can obtain the desired information. This can be done in the step here by using the CANFilter tool. This step is not compulsory; we may use the linked schema instances as they are without any filtering as well.

#### Step 5: Processing the schema instances

The extracted information can be used for many purposes and to promote tool interoperability we can transform the linked/filtered schema instances into any supported format (like GXL and Famix). There are several tools which can do this like CAN2Gxl and CAN2Famix. Apart from simply converting the facts into other formats, more sophisticated processing can also be done. In this work we used the CAN2Metrics tool to obtain different metrics.

### 3.1 Encountered Difficulties

Our approach is still not perfect because we found some configuring examples in makefiles that seem quite “sneaky”. For instance, objects may be moved to another directory before linking, so our wrapper could not find the files containing the associated schema instances in step 3 (it searches them next to the original location of the object files). Another example (in Mozilla) was that the build process created file links to the object files and in the linking step it used these links instead of the original file names.

One solution could also be to wrap the *cp* (file copy), *mv* (file move) and *ln* (file link) commands in a similar way as the compiler, but sometimes this is insufficient. For instance, Mozilla created the file links with its own program (compiled on-the-fly) and wrapping the *ln* tool does not help. In this case we examined all the files and if a file link is found instead of a real file we know that the file containing the corresponding schema instance is next to the file the link points to. It is just one solution and it is obvious that it cannot be used in the case of *cp* and *mv*.

Using special characters (like spaces, quotes and backslashes) in the parameter list can cause problems as well. The Linux shell resolves these special characters and the wrapper scripts and the CANGccWrapper tool get the “modified” parameters which are of course correct but sometimes they cannot be simply sent to the wrapped compiler without modification. CANGccWrapper has to deal with these special characters as well.

## 4 Experiment

We tested our wrapping approach with the open source real-world system *Mozilla* [4, 5, 6] successfully.

First of all, let us take a look at the compilation time (with and without wrapping). This can be very important in the case of large projects such as Mozilla. We measured the compilation time of Mozilla version 1.6 on a Linux platform (kernel 2.4.22) on a computer with a 3GHz Pentium-4 processor with 1 GB RAM. The original compilation took 27 minutes while with wrapping enabled it took 3 hours and 10 minutes (original compilation included). This consisted of three parts: the compilation itself with wrapper enabled took 2 hours and 22 minutes (without filtering), linking all schema instances into a single one took 11 minutes and calculating the metrics from the final schema instance took about 37 minutes (without filtering). Of course, the last two steps are not obligatory. We treated all the subsystems of Mozilla as a whole and calculated several metrics. We provide two examples in the following.

**Example 1: System level metrics.** In Figure 2 we present some system level metrics that describe some basic properties of the whole system. The metrics and their definitions can be seen below the table.

AHF	NCL	TLOC	TNM	TNA	WMC*
0.42	9701	1486986	107953	79836	24.6

**AHF** (Attribute Hiding Factor). The average of the invisibilities of each attribute defined in each class. The invisibility of an attribute is the percentage of the total classes from which this attribute is not visible [8].

**NCL**. Number of classes [9].

**TLOC**. Total number of (non-empty) lines of code in the system [9].

**TNM**. Total number of methods in the system [9].

**TNA**. Total number of attributes in the system [9].

**WMC** (Weighted Methods per Class). Weight: McCabe cyclomatic complexity [8, 10].

**WMC\***. We calculated the average of WMCs for all the classes.

Figure 2: Some system level metrics that were calculated

**Example 2: Class level metrics.** With these metrics the classes can be examined one at a time and they can be also used to calculate different statistics that describe the whole system. In our example (see Figure 3) we show a statistic using the metric DIT. We examined the classes one by one and classified them according to its DIT value.

Depth	0	1	2	3	4	5	6	7	8	9	10
Number of classes	1705	3562	2217	1186	481	190	139	133	75	11	2

**DIT** (Depth of Inheritance Tree). The length of the longest path from the class to the root in the inheritance hierarchy [10].

Figure 3: Some system level metrics

## 5 Summary

In this paper we calculated various metrics from the source code of the well-known open source web and e-mail suite called Mozilla. To obtain the required metrics we used our Columbus framework which has been further developed recently with a novel, so-called *compiler wrapping* technology (see Section 3) giving us the possibility of automatically analyzing and extracting information from practically any software system that compiles with GCC on the Linux platform. Moreover, we can do this *without modifying* any of the source code or makefiles. We also briefly introduced our fact extraction process briefly to demonstrate what logic drives the different tools of the Columbus framework and what steps need to be taken to get the necessary facts for calculating the metrics. Using the extracted facts we calculated different metrics to characterize the Mozilla system.

In the future we plan to gather metrics also from the previous versions of Mozilla to study how the software evolved and how their representative parameters changed. We also intend to build and fill a large database containing metrics about different open source projects (like OpenOffice, for instance) and to make this publicly available for researchers.

## References

- [1] R. Ferenc and Á. Beszédes. Data Exchange with the Columbus Schema for C++. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR 2002)*, pages 59–66. IEEE Computer Society, Mar. 2002.
- [2] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, Oct. 2002.
- [3] Homepage of FrontEndART Software Ltd. <http://www.frontendart.com>.
- [4] M. W. Godfrey and E. H. S. Lee. Secrets from the monster: Extracting mozilla’s software architecture. In *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.
- [5] The Mozilla Homepage. <http://www.mozilla.org>.
- [6] C. R. Reis and R. P. de Mattos Fortes. An overview of the software engineering process and tools in the mozilla project. In *Proceedings of the Workshop on Open Source Software Development*, pages 155–175, Feb. 2002.
- [7] L. Vidács, Á. Beszédes, and F. Rudolf. Columbus Schema for C/C++ Preprocessing. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, to appear. IEEE Computer Society, Mar. 2004.
- [8] J. R. Abounader, D. A. Lamb. Data Model for Object-Oriented Design Metrics. Queen’s University, Kingston, ON. 1997.
- [9] F. Fioravanti, P. Nesi. A method and tool for assessing object-oriented projects and metrics management. In *The Journal of Systems and Software*, North-Holland, Elsevier Science Inc. Press, New York, USA, Vol.45, 2001.
- [10] V. Laing, C. Coleman. Principal Components of Orthogonal Object-Oriented Metrics. White Paper Analyzing Results of NASA Object-Oriented Data (323-08-14), October 2001.

## Postal addresses

**István Siket**

*Department of Software Engineering  
University of Szeged  
H-6720 Szeged, Árpád tér 2, Hungary*

**Rudolf Ferenc**

*Department of Software Engineering  
University of Szeged  
H-6720 Szeged, Árpád tér 2, Hungary*