Extracting Facts with Columbus from C++ Code

Rudolf Ferenc, Árpád Beszédes and Tibor Gyimóthy
University of Szeged, Department of Software Engineering
{ferenc|beszedes|gyimi}@inf.u-szeged.hu

Abstract

Fact extraction from software systems is the fundamental building block in the process of understanding the relationships among the system's elements. It is evident that in real life situations manual fact extraction must be supported by software tools which are able to analyze the subject system and provide useful information about it in various forms. These forms are most useful if they adhere to prescribed schemas and this way promote tool interoperability. In this work we outline our solution to tool supported fact extraction, which is built upon the reverse engineering framework Columbus and is supported by schemas for the C++ language. We describe the extraction process in detail and show how the extracted facts can be used in practice by processing the schema instances. We also introduce new features of the Columbus system not published previously, which among others include compiler wrapping and source code auditing.

Keywords

Reverse engineering, fact extraction, tool interoperability, standard exchange format, schema, C, C++, Columbus, CAN, CANPP

1 Introduction

Software systems are rapidly growing and changing, so source code written today gets out-of-date tomorrow. This is among others due to the quickly changing market requirements and also due to the continuously upcoming new technologies. The always tight deadlines often prevent the developers to release a product properly with up-to-date documentation (design descriptions, source code comments, etc.). As a result there is a great need to understand the relationships between the different parts of a large system.

To comprehend an unfamiliar software system we need to know many different things about it. We refer to this information as facts about the source code. A *fact* is for instance the size of the code. Another fact is whether a class has base classes. Actually any information that helps us to understand unknown source code is called a fact in this paper. It is obvious that collecting facts by hand is only feasible when relatively small source codes are investigated. Real-world systems that contain several million lines of source code can be only processed with the help of tools.

Tool supported *fact extraction* is in our approach an automatized process during which the subject system is analyzed file-by-file with analyzer tools to identify the source code's various characteristics and their interrelationships and to create some kinds of representations of the extracted information. The form of the output of these tools is usually prescribed by schemas.

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 incarnation of the schema which models a concrete software system. This concept is analogous to databases, which also have a schema (usually described by E-R diagrams) that is distinct from the concrete instance data (data records). Schemas have a very important role in the process of fact extraction. They define the central repository of the whole process where the facts are stored. We designed two schemas that prescribe the form for storing the facts: the *Columbus Schema for C++ Preprocessing* (for preprocessing related facts) [11] and the *Columbus Schema for C++* (for the C++ language itself) [3].

To make the results of fact extraction widely usable, we further process the schema instances to take various new formats. These can be very simple transformations, like for instance XML and HTML, but the processing can be much more sophisticated, like calculating metrics and recognizing design patterns. With the help of a so-called code audit processing we implemented a new tool called CPPAudit (see Section 3).

The main contributions of the paper can be summarized as follows. Most importantly, we define in detail a process for tool supported fact extraction. Taking the opportunity we also present new features in the Columbus system: the preprocessing schema, IDE integration, compiler wrapping and code auditing, to name only few.

In the next section we will introduce the process of how fact extraction can be done within the Columbus framework

and we also present the used tools. Furthermore, this section describes how the extracted facts can be used in practice. We then present our code auditor tool in Section 3. Afterwards, we describe our experiments in Section 4, which shows that our methods can be applied in real-world cases. Finally, in Section 5 we draw some conclusions and outline directions for future work.

2 Fact Extraction with Columbus

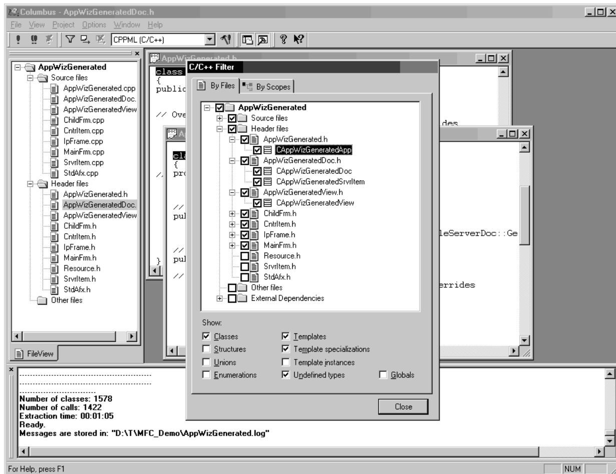

Figure 1. The user interface of Columbus REE

Columbus [4] is a reverse engineering framework, which has been developed in cooperation between the University of Szeged, the Nokia Research Center and FrontEndART [5]. 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 (see Figure 1). The Columbus REE is not limited for the C++ language; all C++ specific tasks are performed by different plug-in modules of it. 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. By doing this we have obtained a versatile and easily extendible environment for reverse engineering. The framework contains further tools as well (mostly command line tools), which actually do the C++-specific jobs, like analyzing the source code and further processing the results.

The outline of the extraction process within the Columbus framework can be seen in Figure 2. The process consists of five consecutive steps where each step uses the results of the previous one. These steps may be performed fundamentally in two different ways: using the visual user interface of Columbus REE, or using only the command-line programs.

The steps of the fact extraction process will be described in the following. An important advantage of the presented steps is that they can be performed *incrementally*, that is, if the partial results of certain steps are available and the input of the step has not been altered, these results must not be regenerated.

Step 1: Acquiring project/configuration information

Acquiring project/configuration information is indispensable to carry out the extraction process. The source code of a software system is usually logically split into a number of files and these files are arranged into folders and subfolders. Furthermore, different preprocessing configurations can apply to them. The information on how these files are related to each other and what setting apply to them are usually stored either in *makefile*s (in the case of building software with the *make* tool), or in different *project files* (in the case of using different *IDE*-s – Integrated Development Environments).

We introduce a so-called *compiler wrapping* technique for using makefile information and two different approaches for handling IDE project files: IDE integration and project file import.

Compiler wrapping. Makefiles can contain not only the references to files to be compiled and their settings but can also contain various commands, like invoking external tools. These powerful possibilities are bad news for reverse engineers, because every action in the makefile must be somehow simulated in the reverse engineering tool. This can be extremely hard or even impossible in some cases. We approached this problem from the other end and solved it by "wrapping" the compiler. This means that we temporarily hide the original compiler, and this way if the original compiler should be invoked our wrapper program will start instead of it, which executes first the original compiler, and second, it invokes our analyzer tools as well. These are invoked with the appropriate parameters in the same environment to build up the required schema instances. This way all we have to do is to build a software system as usual (but with the wrapper switched on).

IDE integration. In this case our tool appears as a new toolbar within the IDE and its operation is very similar to the usual build process. The active project is analyzed and the output can be transformed into any format supported by the Columbus framework. (Currently Microsoft Visual Studio 6.0 and .NET are supported.)

Project file import. The Columbus REE is able to parse Microsoft Visual C++ 6.0 and .NET project files and to im-

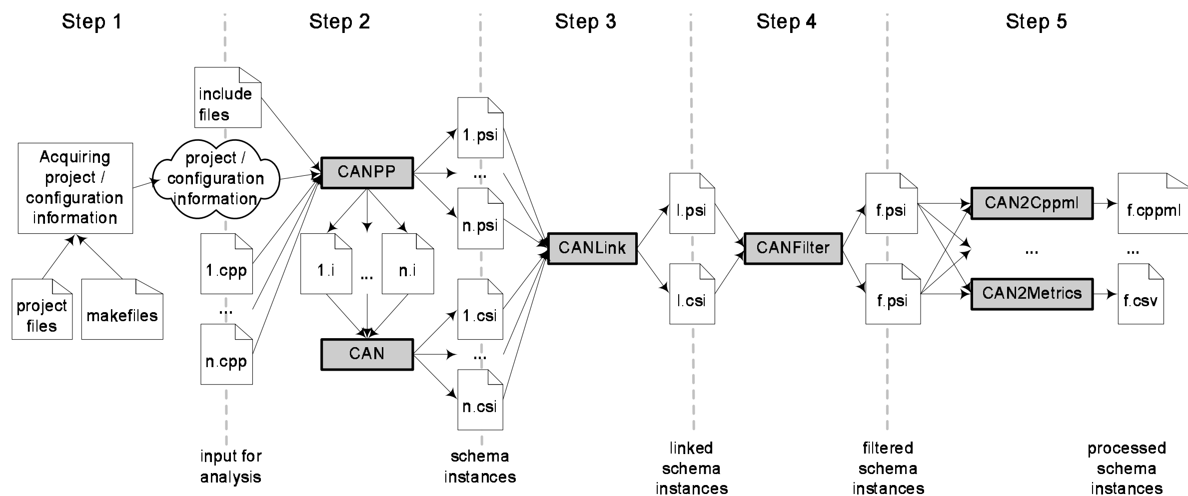

Figure 2. The fact extraction process

port all relevant information from them to be able to analyze the project. All Columbus REE features can be used in this case.

Manual setup. Besides these possibilities the project can be built up by hand also in the Columbus REE (for instance if no project information is available at all). A so-called *Project Setup Wizard* is available to help in this task.

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

In this step the input files are processed one by one using the project information acquired in the first step (for instance macro definitions to be used and paths to different header files to be included). First, the preprocessing and the extraction of preprocessing related information is done with the *CANPP* tool. Second, the preprocessed file is handed over to the *CAN* tool, which then analyzes the file and extracts C++ language related information. Both tools create the corresponding schema instances and save them into appropriate files. This step is performed for every single input file.

In the Columbus REE, the analysis of the input source file is performed with the C++ extractor plug-in module, which invokes the CANPP and CAN tools for carrying out the real analysis.

CANPP is a command line program for analyzing C/C++ preprocessor related language constructs and for preprocessing the code. The input is a C++ source file with various settings (like include path and macro definitions), and the outputs are the preprocessed file and the built up instance of the Columbus Schema for C++ Preprocessing of the source file.

CAN 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++ of the analyzed unit. The C++ language processed by the analyzer meets the ISO/IEC standard of 1998 [8]. Moreover, this grammar has been extended with the Microsoft extensions used in Visual C++, the Borland extensions used in C++ Builder and the GCC extensions used in g++. The parser is *fault-tolerant* (it has the ability to parse incomplete, syntactically incorrect source code), which means that it can carry on with the analysis from the next parsable statement after the error.

Step 3: Linking of schema instances

After all the schema instance files have been created the *linking* (merging) of the related schema instances is done with the *CANLink* tool. This way, similarily to real compiler systems that create files which contain C++ entities that logically belong together (for example libraries and executables), the related entities are grouped accordingly. The outputs of this step are the merged schema instances for each logical unit (subproject) of the analyzed software system. These merged instances can be of course further merged into one single schema instance to represent the whole software system at the same time.

In the Columbus REE, the linking of the schema instances created by CANPP and CAN is performed by the C++ linker plug-in module, which invokes the CANLink tool for carrying out the real linking.

Step 4: Filtering the schema instances

In the case of really large projects the previous steps can produce large schema instances that contain huge amounts

of extracted data. This is hard to present in a useful way to the user (he/she is usually interested only in parts of the whole system at a given time). Different filtering methods in the Columbus REE can help in solving this problem. (In command-line based processes no such filtering methods are available yet.)

There are three options for filtering:

- *Filtering using C++ element categories*, for instance classes, templates and enumerations. With this option all elements that do not belong to the selected categories will be filtered out.
- *Filtering by input source files*. All C++ elements that come from the input files which are not selected are filtered out. In this way all elements that come from system libraries, for instance, can be easily filtered out (these header files are not strictly part of the user's project).
- *Filtering according to scopes*. Different C++ elements like classes or namespaces can be selected/deselected individually in a tree-view browser that represents the scoping structure of the project.

Step 5: Processing the schema instances

Because different C++ re- and reverse engineering tools use different schemas for representing their data, the (filtered) schema instances must be further processed. The processing may consist of transforming the schema instance into another format and/or applying further computations on it. The following formats/applications are included in the Columbus REE currently:

PPML and CPPML. This transformation permits the creation of XML documents (called *PPML* – Preprocessor Markup Language and *CPPML* – C++ Markup Language) that have structures based on the corresponding Columbus schemas. The exported documents conform to their Document Type Definitions, as described on FrontEndART's homepage [5].

GXL. With this transformation GXL representations can be created from the extracted information. *GXL* (Graph eXchange Language) [7] is an XML-based graph-description format. Since the Columbus schemas basically define graphs, this format is suitable for representing them in a convenient way. The call graph of the analyzed system can be also created in GXL form.

UML XMI. This processing allows the creation of standard UML XMI documents from the Columbus Schema for C++. The XMI document contains the class diagram of the analyzed project which can be further processed with XMI enabled tools (like Rational Rose, Borland Together ControlCenter, etc.).

Famix XMI. With this processing a *Famix* [2] XMI representation of the extracted information can be created. This format can be utilized in the *CodeCrawler* tool for visualization and metric calculations.

RSF. Three transformations are available for creating *rigi* RSF [9] documents: (1) a graph based on the Columbus Schema for C++, (2) a call-graph and (3) a UML class diagram-like graph. All of these use different rigi domains which can be created with Columbus as well.

HTML documentation. This processing can be used to create a hypertext documentation of the extracted project in *HTML* form. The generated documentation presents the project in a browsable and user-friendly fashion. All the necessary information is presented about the classes and other elements in a structured way. Three types of browser frames are also supplied, with which a project can be easily navigated. These present the classes using (1) their names in alphabetical order, (2) the scoping structure and (3) the inheritance relationship.

Metrics. With this processing 88 different object oriented metrics are calculated from the schema instances.

Design Patterns. This processing checks the schema instances for occurrences of design patterns [6] (see [1] for details).

CPPAudit. This processing checks the schema instances for different coding rule violations (see Section 3 for a detailed description).

3 Code Auditing

A special-purpose tool was developed on top of the Columbus fact extraction technology. This tool, called *CPPAudit*, is a code auditor that is able to investigate source code and check it against a set of rules (organized into rule packages) that describe the preferred properties of the code. These rules mostly involve issues related to coding style, but in some cases they extend the warning reporting capabilities of the compiler. Code auditing is performed according to the available rule packages, of which there is a built-in one called *general*. Further rule packages will be available separately, so the program can be easily extended at any time with new packages.

From the user's point of view the basic operation of the tool is very simple: a subject system is analyzed and the eventual rule violations found are supplied to the user. Internally however, complex machinery is involved. Namely, the same extraction process is performed as described in Section 2: the source code is preprocessed and analyzed for preprocessor- and language related facts, and the adequate schema instances are produced. Then linking and default filtering is performed (to select only facts strictly related to the project). In the final step the filtered schema instances are analyzed according to the chosen set of rules and finally the warnings are issued on the output.

CPPAudit can be used in two ways. First, it can be used in user makefiles with the same compiler wrapping technology presented in Step 1 in the previous section. Second, the tool can be integrated into the Microsoft Visual Studio 6.0 and .NET environments (it can be used to check Visual C++ projects) using the IDE integration capability of the Columbus framework (see Section 2). In this case the tool appears as a toolbar within the Visual C++ environment and its operation is fully automatic: the project is analyzed and the reported warnings are provided in the output window of the environment.

4 Experiments

We executed the steps of the fact extraction process described in this paper on an older version of StarOffice Writer [10], which is a large open source C++ project. The following table contains the size information of its source code:

Number of files	Size	LOC[1]
9 449	66.5MB	1 764 574

The project was set up in the Columbus REE, on an Intel P4-1900 machine with 512MB RAM running Windows 2000. The extraction time was 4 hours, 28 minutes and 25 seconds, and the memory consumed was 263 MBytes (full analysis was performed – statements and expressions were extracted as well).

The table below shows the number of some of the extracted items of the test project.

Classes	Namespaces	Functions
4 988	99	61 553
Objects	Statements	Expressions
23 862	308 774	3 062 519

As an experiment we processed the extracted facts and successfully produced all supported formats. For example, the CPPML file was 507MB large.

5 Conclusion and Future Work

In this paper we presented a reverse engineering framework using which different facts about C++ source code can be extracted. It is free for scientific and educational purposes; our intention is to support academic persons in their research work [5]. The framework relieves researchers of the burden of having to write extractors for different purposes and allows them to concentrate on their own concrete research topic.

The main advantage of this work is that it offers an extendible framework for fact extraction and transformation. The framework already supports several popular tools like rigi and CodeCrawler, but it can be easily extended to support any other reverse engineering tool as well. We described how a complete fact extraction process can be performed with our framework, and also outlined some new features not published previously.

In the future we plan to extend the framework with additional processings to support even more RE tools and to add new extractors for supporting additional programming languages. The development of a special-purpose tool similar to CPPAudit is also in our plans for metrics calculation. We already started a very challenging research about designing a compact link between the Columbus Schema for C++ and the Columbus Schema for C/C++ Preprocessing.

[1] Lines of non-preprocessed code.

References

[1] Z. Balanyi and R. Ferenc. Mining Design Patterns from C++ Source Code. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM 2003)*, pages 305–314. IEEE Computer Society, Sept. 2003.

[2] S. Demeyer, S. Ducasse, and M. Lanza. A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization. In *Proceedings of WCRE'99*, 1999.

[3] 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.

[4] 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.

[5] Homepage of FrontEndART Software Ltd. `http://www.frontendart.com`.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Pub Co, 1995.

[7] R. Holt, A. Winter, and A. Schürr. GXL: Towards a Standard Exchange Format. In *Proceedings of WCRE'00*, pages 162–171, Nov. 2000.

[8] International Standards Organization. *Programming languages – C++*, ISO/IEC 14882:1998(E) edition, 1998.

[9] H. A. Müller, K. Wong, and S. R. Tilley. Understanding Software Systems Using Reverse Engineering Technology. In *Proceedings of ACFAS*, 1994.

[10] The StarOffice Homepage. `http://www.sun.com/software/star/staroffice`.

[11] 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.