A short introduction to Columbus/CAN

Rudolf Ferenc, Árpád Beszédes, Ferenc Magyar and Tibor Gyimóthy
{ferenc, beszedes, magyar, gyimi}@cc.u-szeged.hu
RGAI – University of Szeged
H-6720 Szeged, Aradi Vértanúk tere 1, Hungary

Abstract: In this paper we shortly present a reverse engineering framework called Columbus that is able to analyze large C/C++ projects. Columbus supports project handling, data extraction, -representation, -storage and -export. Efficient filtering methods can be used to produce comprehensible diagrams from the extracted information. The flexible architecture of the Columbus system (based on plug-ins) makes it a really versatile and an easily extendible tool for reverse engineering.
Keywords: reverse engineering, source code parsing, large-scale software systems, UML, Class Model, C/C++, templates, call graph

1. Introduction

One of the most critical issues in large-scale software development and maintenance is the rapidly growing size and complexity of the software systems. As a result of this rapid growth there is a need to understand the relationships between the different parts of a large system [1] [2]. The substantial amount of existing legacy code and/or high number of the participants in code development also necessitates the use of tools for *reverse engineering* [11]. Reverse engineering is "the process of analyzing a subject system to (a) identify the system's components and their interrelationships and (b) create representations of a system in another form at a higher level of abstraction" [4].

In this paper we present a reverse engineering framework called *Columbus* [5], which has been developed in a cooperation between the Research Group on Artificial Intelligence in Szeged and the Software Technology Laboratory of the Nokia Research Center. Columbus is able to analyze large C/C++ projects and to extract their UML Class Model [9] and call graph. It supports project handling, data extraction, data representation and data storage. Furthermore, efficient filtering methods can be used to produce comprehensible (clear-cut) diagrams from the extracted information.

2. Columbus

The main motivation for developing the Columbus system was to create such a tool, which implements a general framework for combining a number of reverse engineering tasks and to provide a common interface for them. Thus, Columbus is a framework, which supports project handling, data extraction, data representation, data storage, filtering and visualization. All these basic tasks of the reverse engineering process for the specific needs are accomplished by using the appropriate modules (*plug-ins*) of the system. Some of these plug-ins are present as basic parts of Columbus, while the system can be extended for other reverse engineering requirements as well. This way we get a really versatile and an easily extendible tool for reverse engineering.

2.1 Overview of the Columbus System

The basic operation of Columbus is performed by the use of three types of plug-ins (in form of MS Windows DLLs). These are the following:

- *Extractor plug-ins* (currently an extractor for C/C++) – The task of an extractor plug-in is to properly analyze a given input source file and to create a file, which contains the extracted information.
- *Linker plug-ins* – The task of a linker plug-in is to build up (in the memory) the complete merged internal representation of the project. This process is carried out based on the files created by the extractor plug-in. This plug-in is responsible also for filtering the merged data in order to produce a more clear-cut internal representation for exporting.
- *Exporter plug-ins* – The task of an exporter plug-in is to export the internal representation built up and filtered by the linker plug-in into a given output format. (The currently available exporters are for: TDE Mermaid 2.2, TED 1.0, Rational Rose, Microsoft Jet Database, HTML, XML and ASCII.)

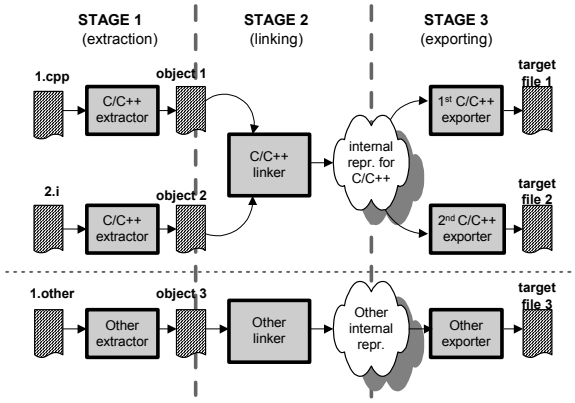
Beside the delivered plug-ins the user can easily write and add his/her own new plug-in DLLs to the Columbus system using the *plug-in API*.

2.2 Columbus Projects

The extraction process is based on a Columbus project. A project stores the input files (and their settings: precompiled header, preprocessing, output directories, message level, etc.) displayed in a tree-view, which represents a real software-system. The project can *simultaneously* contain source files of different programming languages. Non-source code files can be added to the project as well (e.g. documents, spreadsheets), which are displayed by Columbus using OLE technology.

2.3 The Extraction Process

The complete extraction process in Columbus can be seen in the Figure below:



The whole process is very similar to compiler systems. The first stage of the extraction process is the *data extraction*. Columbus takes the input files one by one and passes them to the appropriate extractor, which creates the corresponding internal representation files.

In the second stage the linker plug-in is automatically invoked in order to *link* (merge together) the internal representation files in the memory.

In the third stage after selecting the desired export format the *exporting* is performed. The exporting is usually based on a filtered internal representation. Filtering is discussed in detail in Section 4.

All stages of the extraction process can be influenced by setting various plug-in specific options. An important advantage of the Columbus system is that it can *incrementally* perform all of the above described steps, i.e. if the partial results of the certain stages are available and the input of the stage has not been changed, the partial results will not be recreated.

3. CAN

The parsing of the input source codes is performed by the C/C++ extractor plug-in of Columbus, which invokes a separate program called *CAN* (**C**++ **AN**alyzer). CAN is a command-line (console) application for analyzing C/C++ sources. This allows that it can be *integrated* into the user's makefiles and other configuration files by which it facilitates its automated execution in parallel with the software build process.

Basically, CAN accepts one complete translation unit at a time (a preprocessed source file). However, for files that are not preprocessed a preprocessor will be invoked. The actual results of CAN are the internal representation files, which are the binary saves of the internal representations built up by CAN during extraction.

One of the greatest asset of CAN is probably the *handling of templates* and their *instantiation at source level*, which is accomplished using a *two-pass technique* for analysis. This way a separate analyzer belongs to both passes, which recognize different things from their inputs. As the task of the first pass is only to recognize the language constructs in connection with the templates, the analyzer of this pass ignores everything else (like a *"fuzzy" parser*). The second pass performs the complete analysis of the source code and creates its internal representation. So the analyzer of this pass is a complete C++ analyzer. The language description of C++ implemented in the analyzer covers the ISO/IEC C++ standard of 1998 [10]. Furthermore, this grammar is extended by the Microsoft extensions used in Microsoft Visual C++ 6.0.

The information collected by CAN comprises the UML Class Model including C++ templates (definitions, specializations and instantiations) and the call graph. CAN supports the *precompiled headers* technique as well that is widely used by compiler systems in order to decrease compilation time. This technique is efficient especially in case of *large projects*. The parser is *fault-tolerant* (it has the ability to parse incomplete, syntactically incorrect source code), which means that it can continue the analysis from the next parsable statement after the error.

4. Producing Comprehensible Diagrams

The reverse engineered code can produce huge amount of extracted data, which is hard to visualize in a way that offers useful information for the user (the user is interested only in parts of the whole system at a time). Different filtering methods in Columbus can help solving this problem.

There are four options for filtering:

- Filtering *by input source files*: only classes that come from the given input files can be selected.
- *Filtering according to scopes*. Classes or namespaces can be selected individually in a tree-view browser.
- *Filtering using class dependencies* (e.g. aggregation, inheritance), with which the given relations can be selected. An interesting and useful feature is *Diagram Completing*, with which we can control the possible elements brought in by the relations transitively controlling this way the completeness of the class diagram (e.g. using this option we can select all derived classes of a given class).
- *Filtering "by hand"*: The classes can be individually selected/deselected on the displayed class diagram customizing it this way.

5. Conclusion

In this paper we briefly presented the functionalities of the Columbus toolset with respect to its reverse engineering capabilities.

Columbus supports several reverse engineering tasks (e.g. project handling, data extraction and data representation/visualization with filtering and exporting options). The current version is able to analyze C/C++ projects but due to its flexible architecture it is easy to extend it with other languages as well.

The main features of Columbus can be summarized as follows:

- Effective project handling (capability for importing MS Visual C++ projects, integration into the user's project).
- Powerful C/C++ extraction (fast, fault-tolerant parsing, handling of complex templates, visualizing "hidden" template instances).
- Direct access to the extracted information (*via* its API).
- Creation of comprehensible diagrams (filters, layout).
- Easy-to-use user interface (very similar to IDE-s).
- Extensibility (plug-in architecture, user plug-ins *via* its plug-in API).
- Various output formats (Mermaid, TED, Rose, MS Jet, html, XML, ASCII).

In the future we will extend the system for other source languages (e.g. Java) and more output (export-) formats. Further improvements are under development as well, which may be useful for better code understanding (e.g. dependency-graph [6][3][7]). In the future we plan to enhance Columbus so that it supports *architectural reconstruction* of software systems [1] (recognizing design-patterns [8], component interaction, structural information).

References

1. Armstrong, M. N., Trudeau, C. *Evaluating Architectural Extractors*. In Fifth Working Conference on Reverse Engineering. Oct. 12-14, 1998. Honolulu, Hawaii, USA. 30-39.
2. Bellay, B. and Gall, H. *An Evaluation of Reverse Engineering Tool Capabilities*. In Software Maintenance: Research and Practice. 10. 1998, 305-331.
3. Beszédes, Á., Gergely, T., Szabó, Zs. M., Csirik, J. and Gyimóthy, T. *Dynamic Slicing Method for Maintenance of Large C Programs*. In Proc. 5th European Conference on Software Maintenance and Reengineering (CSMR 2001). Lisbon, Portugal, March 14-16, 2001. 105-113.
4. Chikofsky, E.~J. and Cross II, J.~H. *Reverse engineering and design recovery: A taxonomy*. IEEE Software 7, 1. Jan. 1990. 13-17.
5. *Columbus Setup and User's Guide*. Version 2.5, © 1998-2000 Nokia Research Center.
6. Gyimóthy, T., Beszédes, Á., and Forgács, I. *An Efficient Relevant Slicing Method for Debugging*. In Proc. 7th European Software Engineering Conference (ESEC). Toulouse, France. Sept. 1999. LNCS 1687. 303-321.
7. Jackson, D. and Rollins, E. J. *A new model of program dependences for reverse engineering*. In Proceedings of the second ACM SIGSOFT symposium on Foundations of software engineering. 1994. 2-10.
8. Keller, R. K., Schauer, R., Robitaille, S. and Pagé, P. *Pattern-Based Reverse-Engineering of Design Components*. 1999. ICSE '99, Los Angeles CA, USA. 226-235.
9. *OMG Unified Modeling Language Specification*. Version 1.3, © 1999 Object Management Group, Inc.
10. *Programming languages – C++*. ISO/IEC 14882:1998(E).
11. Quilici, A. *Reverse engineering of legacy systems: a path toward success*. Proceedings of the 17th international conference on Software engineering. 1995. 333-336.