

Columbus – Reverse Engineering Tool and Schema for C++

Rudolf Ferenc¹, Árpád Beszédes¹, Mikko Tarkiainen² and Tibor Gyimóthy¹

¹University of Szeged, Research Group on Artificial Intelligence
{ferenc,beszedes,gyimi}@cc.u-szeged.hu

²Nokia Research Center
mikko.t.tarkiainen@nokia.com

Abstract

One of the most critical issues in large-scale software development and maintenance is the rapidly growing size and complexity of software systems. As a result of this rapid growth there is a need to better understand the relationships between the different parts of a large software system. In this paper we present a reverse engineering framework called Columbus that is able to analyze large C++ projects, and a schema for C++ that prescribes the form of the extracted data. The flexible architecture of the Columbus system with a powerful C++ analyzer and schema makes it a versatile and readily extendible toolset for reverse engineering. This tool is free for scientific and educational purposes and we fervently hope that it will assist academic persons in any research work related to C++ re- and reverse engineering.

Keywords

Tool interoperability, standard exchange format, C++ schema, front end, ASG, Columbus/CAN, GXL

1 Introduction

One of the most critical issues in large-scale software development and maintenance is the rapidly growing size and complexity of 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, 3]. The substantial amount of existing legacy code and/or high number of participants in code development also necessitates the use of tools for *reverse engineering* [22]. 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]. We introduce a tool that is able to analyze large C++ software systems and to present the extracted information in a common specification called the *Columbus Schema for C++* [8]. This schema

describes the way the various language elements should be represented and the relationships among these [11].

In recent years it has become increasingly apparent that, in order to successfully perform a real-life software maintenance or reengineering task, a suitably assembled *set of tools* is required (e.g. front ends and code-rewriters, metrics tools, documentation tools, visualization tools and even compilers). In order to achieve interoperability among these tools we need a *common schema* and a *front end* that generates data according to the schema. Based on previous research [11, 19] we present some requirements, which should be met when designing such a schema, front end and *framework* to assist in the integration of the RE tools.

Requirements for a framework:

- *Project handling.* The source code of real-life software systems is always divided into multiple files spread across several directories, which can have different build options, linking settings and interdependencies. The way this information is handled is of great importance for a reverse engineering tool.
- *Extendibility.* Extendibility is needed to connect the system with other tools. This can be done by furnishing a well defined interface (API), which allows access to the tool’s functionality.
- *Filtering.* Filtering is especially useful when extracting large projects. The reverse engineered code can produce huge amounts of extracted data, which is hard to present in a way that provides useful information to the user (the user is interested only in *parts* of the whole system at a given time).
- *Visualization.* After the data extraction from the subject system has been performed, information should be presented to the user in a usable form. The best way of doing this is to represent the data in visual form, e.g. in the form of diagrams, charts, or tables.

Requirements for a C++ front end:

- *Correctness.* The first and probably most important requirement for a front end is that it should generate correct information about the subject system.
- *Completeness.* The front end should be able to perform a complete analysis of the system. So called “fuzzy” parsers that extract only some higher level constructs (e.g. classes and functions) can be also useful, but only in a limited number of applications.
- *Fault tolerance.* It should have the ability to parse incomplete and syntactically incorrect source code.
- *Parsing speed.* The front end should be as fast as possible because of the generally large amount of source code.
- *Preprocessing.* The front end should be able to preprocess the C++ source files (e.g. to process include directives and to expand macros).
- *Dialects.* Unfortunately, C++ is a programming language that has several dialects (e.g. GNU, Microsoft, Borland). In order to be widely accepted, the front end should be able to handle as many dialects as possible.
- *Command line operation.* Because real-world C++ systems are usually built using makefiles and command line tools, the front end should be also able to operate in command line mode.

Requirements for a C++ schema:

- *Level of detail.* The schema should be able to represent C++ elements at different levels of abstraction. In [11] three levels of abstraction were defined: (1) Lexical Structure, (2) Syntax, and (3) Semantics.
- *Modularity/extendibility.* The schema should be modular and easily extendible. This means that logically different parts of the language should be described in different modules of the schema description (e.g. type representations, expressions) and that the interconnections between these packages should be as weak as possible.
- *Instance representation.* It should be relatively straightforward to create physical representations (e.g. files) from a schema instance – an abstract syntax graph – to facilitate data exchange (e.g. with GXL [14] or RSF [20]).
- *Dialects.* Similar to the *dialects* requirement set for front ends, when designing a schema special care needs to be taken with the different C++ dialects. In order to be widely applicable the schema should be sufficiently general to handle these dialects.

In this paper we present the Columbus tool and Schema for C++, which fulfill most of the above set requirements. Moreover, this tool is free for scientific and educational purposes and can be downloaded from FrontEndART’s homepage [12], and we hope that we will assist academic persons in any research work related to C++ re- and reverse engineering. The framework relieves researchers of the burden of having to write parsers for different purposes and allows them to concentrate on their own concrete research topic.

In the next section we will describe the Columbus framework in detail. In Section 3 we present the C++ front end of Columbus called *CAN*. We then present the Columbus Schema via an example in Section 4. Section 5 discusses some tools and schemas having similar objectives as ours. Finally, in Section 6 we draw some conclusions and outline directions for future work.

2 Columbus Framework

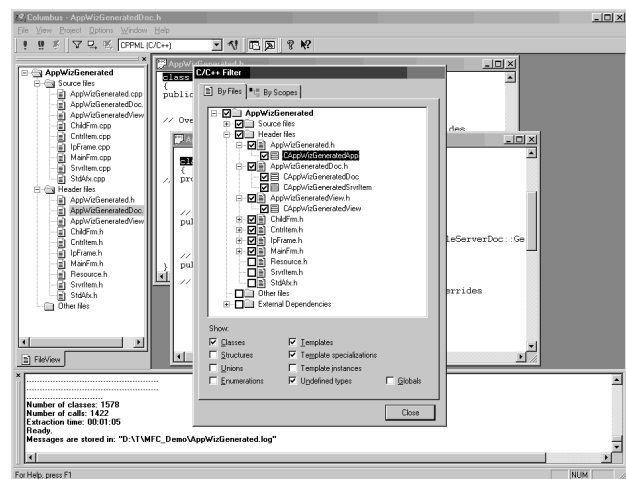


Figure 1. The user interface of Columbus

Columbus [10] is a reverse engineering framework, which has been developed in cooperation between the Research Group on Artificial Intelligence in Szeged, the Software Technology Laboratory of the Nokia Research Center and FrontEndART Ltd. Columbus is able to analyze large C/C++ projects and to extract data according to the Columbus Schema (see Section 4).

The main motivation behind developing the Columbus system was to create 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 and filtering. All these basic tasks of the reverse engineering process

are accomplished by using the appropriate modules (*plug-ins*) of the system. Some of these plug-ins are present as basic parts of Columbus, and the system can be extended to include other reverse engineering requirements as well. By doing this we have obtained a versatile and an easily extendible tool for reverse engineering.

Columbus is used in research projects in the Nokia Research Center (e.g. for reverse engineering UML Class Diagrams for Nokia's UML design environment *TDE* [26]) and in the *Maisa* project of the Helsinki University for recognizing standard Design Patterns in C++ programs [9].

Another example is the use of Columbus in the *FAMOOS* project with the *Crocodile* metrics tool [23]. An important application is the ongoing work on information exchange between Columbus and the *GUPRO* tool, which uses GXL as its input format [7]. A successful example of information exchange with the *rigi* graph visualizer tool [20] is also recently accomplished.

Figure 1 shows a typical snapshot of a Columbus session.

2.1 Overview of the Columbus System

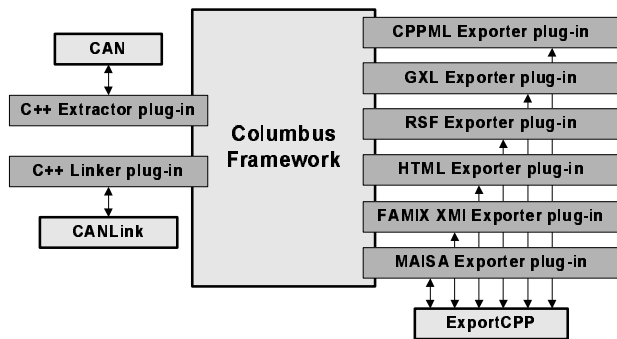


Figure 2. The structure of the framework

The basic operation of Columbus is performed via three types of plug-ins (see Figure 2). These are the following:

- *Extractor plug-ins* (currently only 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 also responsible 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, to a given output format. The currently available exporters are for three XML formats (CPPML, GXL and Famix XMI), RSF, HTML and Maisa. UML XMI and VCG exports will be available soon as well.

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

2.2 The Extraction Process

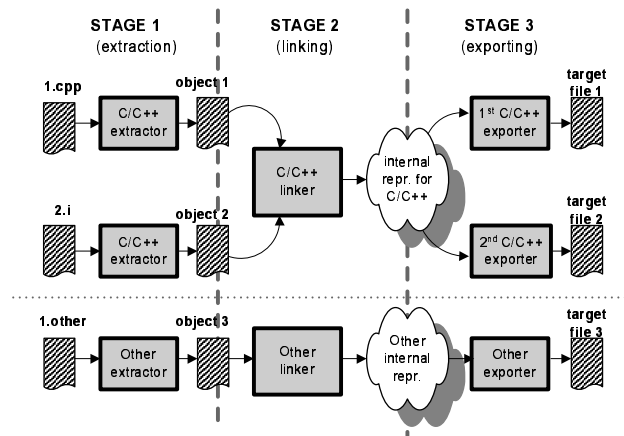


Figure 3. The extraction process

The extraction process is based on a Columbus *project*. A project stores the input files (and their settings: precompiled header, preprocessing, output directories, etc.) displayed in a tree-view, which represents a real software-system. The project can simultaneously contain source files of different programming languages.

The complete extraction process in Columbus can be seen in Figure 3. The process is very similar to a compiler system. The *first stage* of the extraction process is *data extraction*. Columbus takes the input files one by one and passes them to the appropriate extractor, which then creates the corresponding internal representation file.

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. Exporting is usually based on a filtered internal representation. Filtering is discussed in detail in Section 2.3.

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*

Famix XMI – CodeCrawler. With this exporter a *Famix* [6] XMI representation of the extracted information can be created. This format can be utilized with the *CodeCrawler* tool for visualization and metric calculations (an example is shown in Figure 6).

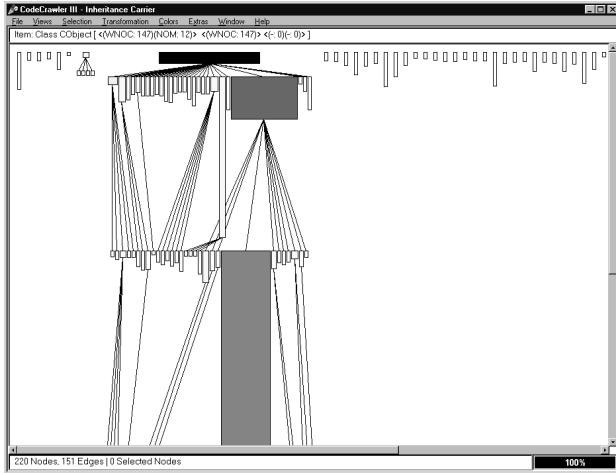


Figure 6. Visualization within CodeCrawler

Maisa. This exporter can be used to create an input file for the *Maisa* tool. *Maisa* [9, 21] is a metrics tool that assesses the quality of a software architecture. One of the functionalities of *Maisa* is the mining of design patterns from the input architecture.

Other. The UML XMI and VCG exporters are partially implemented and will be available soon. Besides these formats, arbitrary third-party plug-ins can be easily written and added to the Columbus system.

2.5 Evaluation of the Columbus framework

In this section we will evaluate the Columbus tool according to the requirements stated in the Introduction:

- **Project handling.** As already described in Section 2.2, Columbus has powerful project handling features. It stores the input source files and their settings, such as the precompiled header, preprocessing options and output directories. Visually, it is displayed in a tree-view. In addition Columbus offers a limited capability for importing existing Microsoft Visual C++ project files. The project handling also has a deficiency, namely supporting dependencies among multiple projects. It handles only one project at a time.
- **Extendibility.** Columbus is highly extendible due to its plug-in architecture (see Section 2.1). The user can easily write and add his/her own new plug-in to the Columbus system using an easy-to-use *plug-in API*.

- **Filtering.** Columbus offers powerful filtering (see Section 2.3). The default filtering already removes those C++ elements that come from the standard libraries and leaves only the user’s own classes in the filter.
- **Visualization.** Currently Columbus does no real visualization (just a simple tree-view in the filter window that lists the classes and namespaces in their scoping structure). Instead it passes this job over to other tools that have been integrated with Columbus (e.g. *rigi* [20] and *CodeCrawler* [6]).

3 CAN – The C/C++ Analyzer

The parsing of the input source code is performed via the C/C++ extractor plug-in of Columbus, which invokes a separate program called *CAN* (C++ ANalyzer). *CAN* is a command line application for analyzing C/C++ code. This allows its integration into the user’s makefiles and other configuration files, thus facilitating automated execution in parallel with the regular software build process.

In essence, *CAN* accepts one complete translation unit at a time (a preprocessed source file). 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. These files will be *linked* (merged together) by *CANLink*, the command line linker for *CAN*.

CAN is able to instantiate templates at source level, which is accomplished using a *two-pass technique* in program analysis. The first pass only recognizes the language constructs in connection with the templates (like a “fuzzy” parser) and instantiates them. The second pass then performs the complete analysis of the source code and creates its internal representation.

The C++ language processed by the analyzer meets the ISO/IEC standard of 1998 [15]. Moreover, this grammar has been extended with the Microsoft extensions used in Visual C++ and Borland extensions used in C++ Builder.

The information collected by *CAN* corresponds to the Columbus Schema (see Section 4). *CAN* supports the *precompiled headers* technique as well, which 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 carry on with the analysis from the next parsable statement after the error.

3.1 Experiments

In this section we demonstrate *CAN*’s extraction capabilities. The experiments were performed on different C++ projects listed below:

- *IBM Jikes compiler* [16]. Using this project we investigated CAN's capabilities for handling sophisticated class hierarchies.
- *Leda graph library* [18]. This project was used to demonstrate the front end's capabilities for handling sophisticated templates.
- *StarOffice Writer* [25]. This is a large C++ project that consist of 9,449 source files (more than 1.7 million lines of non-preprocessed code!). Using this project we investigated CAN's capabilities for handling real-size, huge projects.

The following table contains the size information of the projects:

Project infos	No. of files	Size	LOC ¹
Jikes	77	3.5MB	94 611
Leda	508	2.9MB	116 752
Writer	9 449	66.5MB	1 764 574

The next table shows the measurement results of the extraction (all tests were performed on an Intel PIII-800 machine with 384MB RAM running Windows 2000). It contains the extraction time and the memory consumption.

Extraction	Time ²	Memory
Jikes	00:01:02	19MB
Leda	00:05:50	49MB
Writer	01:55:09	139MB

The table below shows the number of extracted items from the test projects.

Statistics	Classes	Namespaces	Funcs	Attribs
Jikes	275	1	3 471	1 643
Leda	1 563	1	10 802	8 287
Writer	4 988	99	61 553	23 862

It can be seen that the memory consumption is approximately linear with the complexity of the input (i.e. the number of different C++ language elements like classes and functions). Due to this fact and the successful extraction of StarOffice Writer, we can say that the front end is able to handle real-size, large projects.

3.2 Evaluation of the CAN front end

In this section we will evaluate the CAN front end based on the requirements stated in the Introduction:

¹Lines of non-preprocessed code.

²Time format: hh:mm:ss.

- *Correctness.* When evaluating a tool proving its correctness is probably the most difficult task of all. In the case of a front end, correctness means that the extracted information entirely corresponds to the input. In the case of a real-size, huge C++ project such a comparison is practically unfeasible. The only way of doing this is to take random samples from the output, investigate the corresponding input and check if it is in accordance with the samples taken. This method found that the extracted information is mostly correct (only some very complex templates caused problems for the front end).
- *Completeness.* The front end performs a complete analysis of the system, but is does not yet build up the parts of the ASG for statements and expressions (it is partially implemented and the next release will contain it). However, the call graph of the system is already present.
- *Fault tolerance.* CAN is able to parse incomplete and syntactically incorrect source code (see Section 3).
- *Parsing speed.* According to the results shown in the second table in Section 3.1, we can say that the analyzer is rather fast. It extracted StarOffice Writer in less than two hours.
- *Preprocessing.* CAN does not yet perform any preprocessing. However, for files that are not preprocessed, it invokes an external preprocessor. The implementation of a built-in preprocessor is in progress.
- *Dialects.* Apart from standard C++, CAN can handle the Microsoft and the Borland dialects of C++.
- *Command line operation.* As already stated previously, CAN and CANLink are command line tools, which can also be used without the framework, e.g. in makefiles.

4 Columbus Schema for C++

Successful data exchange is crucial among reverse engineering tools. This requires a common *format*, which is applicable in various reverse engineering tools such as front ends and metrics tools. A standard schema must be found. In this paper we propose an exchange schema for the C++ language called the *Columbus Schema for C++* [8].

The Columbus Schema captures the (preprocessed) C++ language at low detail (AST) and also contains higher-level elements (e.g. semantics of types). The description of the schema is given using UML Class Diagrams, which permits its simple implementation and easy physical representation (e.g. using GXL). It is modular, thus it provides addi-

tional flexibility for any future extension/modification. The schema is divided into six packages:

- *base*: The base package that contains the base classes and data types for the remaining parts of the schema.
- *struc*: This package models the main program elements based on their scoping structure, such as objects, functions and classes.
- *type*: The classes in this package are used to represent the types of the elements in the *struc*, *templ* and *expr* packages.
- *templ*: The package covers the representation of template parameter and argument lists, and is used by the *struc* and *type* packages.
- *statm*: The package contains classes modelling the statements.
- *expr*: The classes in this package represent all kinds of expressions.

A detailed description of the schema is given in our previous paper [8]. We will present it here through an example.

4.1 Example schema instance

We illustrate the use of our schema through an example instance of it. We will employ an extended version of the example in [11] (see Figure 7). The ASG for the example is given in Figure 8. We use an UML Object Diagram-like notation, where the object instances of the schema's classes are represented and the links that connect them clearly show the instances of various association and aggregation relations. The ordered associations are represented by numbering the links. For the sake of clarity, we have simplified the diagram by omitting some attributes, such as line numbers, which are not necessary for the description.

The schema uses integers as unique identifiers for nodes, so the key of the topmost node is "1." The class of each node is given to the right of the key number; for example, the class of node 1 is *struc::Namespace*. The schema uses a *name* attribute in some nodes to give the name of the source item being represented; for example, the name attribute of node 1 is "*global namespace*."

We will use the example to explain how the template class, the types, the object and functions are represented in the ASG based on the Columbus schema.

The template class *Array* is represented by node 11. It contains a template parameter list (node 12), which has two children (the order is shown on the connecting edges) that represent the two template parameters. The first one is a type name *T* that is referred from two other nodes. The second one is the non-type parameter (value) *Size*, which has a

type. This type is represented by *TypeRep* node 32. As it can be clearly seen, the template representation is completely separated from the basic scoping structure of the ASG. By doing this, it can be easily replaced with another kind of template description if required.

The template class *Array* has three children (ordered): object *arr* (node 15); and functions *get* (node 16) and *set* (node 23). The type of object *arr* is represented by *TypeRep* node 34. The facts that function *get* is virtual and constant are stored as attribute fields in node 16. The function has one parameter with the name *idx* (node 17). The type of this node is the same as the type of template parameter *Size*, so it refers to the same *TypeRep* 32. In this way there will be only one instance of each unique type representation and it is straightforward to compare the types of two different C++ elements. We will present the type representation of function *get* in detail (node 38).

TypeRep 38 contains a *TypeFormerFunc* node (39), which refers to the *TypeRep* of the function's return type (node 40) and the *TypeRep* of the parameter *idx* (node 32). This is needed because the type of a function includes the whole signature (return type and parameter types). The return type representation (node 40) stores the constant nature of the return type and contains two type formers for the type itself. The first one is a pointer (reference) former (*TypeFormerPtr* 41), which means that the return type is a reference to a type. The second type former is a *TypeFormerType* (node 42), which refers to template parameter 13. So the overall meaning of the type representation is a reference to template parameter *T*. It can be seen that, similar to the template representation, the type representation is also completely separated from the rest of the ASG so it can be easily modified or replaced with another approach of describing types if needs be.

The body of function *get* is represented by node 18, which is a block statement (*statm::Block*). This block is rather simple, containing just one return statement (*statm::Return* node 19) that represents its return value with node 20. This return value is an array subscript expression (*expr::ArraySubscript*) whose both left and right operands are of the kind *expr::Id*, which simply refer to nodes *arr* (15) and *idx* (17), respectively.

The representation of function *set* is similar to that mentioned earlier, so we will not elaborate on it here.

4.2 Evaluation of the Columbus Schema

In this section we will evaluate the Columbus Schema according to the requirements stated in the Introduction:

- *Level of detail*. The schema does not fully represent the lexical structure, but this is not a major drawback since there are only few application types that make use of it (e.g. pretty printing). The syntax and the semantics

of the subject system are represented in fine-grained detail. However, higher level information such as data dependencies are not provided.

- *Modularity/extendibility.* The schema is modular, it is divided into six packages, and the connection between them is very weak. This means that it is easily extendible (e.g. the type representation package can be freely replaced if required).
- *Instance representation.* A schema instance (an abstract syntax graph) of the Columbus Schema is a simple graph, so it can be easily represented in any desired physical form. Some of these have already been implemented (e.g. GXL [14] or RSF [20]).
- *Dialects.* The Columbus Schema contains an attribute for storing non-standard C++ specifiers. Apart from this, the modular structure of the schema allows its easy extension to support arbitrary language dialects and extensions.

5 Related Work

In this Section we will discuss three tools and two schemas which have similar objectives to the Columbus tool and Schema. We will consider these tools because similar to Columbus, they are freely available for research and academic purposes.

5.1 Datrix and CPPX

Datrix. The Datrix team, a part of Bell Canada's Quality Engineering and Research group, implemented source code assessment tools with the goal of evaluating the maintainability and the evolvability of software products [17]. *Datrix* is an analyzer that extracts information from C/C++/Java source code files according to the Datrix ASG Model [2]. Similar to CAN, it creates separate output files for individual compilation units. These output files can be in form of TA (Tuple-Attribute Language) or VCG. For handling C/C++ sources Datrix has also a preprocessor utility. Unfortunately the project at Bell Canada come to an end in year 2000 before all planned tools could be implemented. For example, the Datrix toolset lacks a linker for merging the extracted ASG-s, so viewing the whole system at the same time is not possible.

CPPX. *CPPX* [5] is a free, open source, general purpose parser and fact extractor for C++ developed at the University of Waterloo. It relies on the preprocessing, parsing, and semantic analysis of the GNU g++ compiler, and produces a graph based on the Datrix fact model, in either GXL, TA, or VCG format. Because it relies on the output of a real C++ compiler, it produces very precise data about the analyzed

system, but it has the drawback that it cannot handle other C++ language dialects.

As we have already mentioned, both Datrix and CPPX extract information according to the Datrix ASG Model. A detailed description of the model is available in [2]. This is a well written documentation, but – apart from a short initiative [13] – it lacks a real schema description (e.g. using UML Class Diagrams). Another deficiency of this model is that it tries to be language independent, thus it represents C++ artifacts at a higher level than the Columbus Schema, e.g. it has no distinct nodes for templates and template specializations – these are all simply generic types in Datrix.

Both the Datrix and CPPX toolsets consist of command line utilities only, and lack a user interface for easy manageability.

5.2 Source-Navigator

Red Hat Source-Navigator [24] is a code analysis and comprehension tool that provides a graphic framework for understanding and reengineering large software projects. Source-Navigator parsers scan through source code, extracting information from existing C, C++, Java, COBOL and other programs and then use this information to build a project database. Source-Navigator graphical browsing tools use this database to query symbols (such as functions and global variables) and the relationships between them.

Similar to Columbus, in addition to the languages supported in the standard distribution, new parsers can be added using the Software Development Kit (SDK). Source-Navigator, like Columbus, supports project handling, data extraction, data representation, data storage and filtering. The project files are displayed in a simple file list, or according to the physical tree structure on the disk. There is no way of defining logical folders as one can in Columbus. The user interface of Source-Navigator is more powerful than that of Columbus. However, Source-Navigator has only simple textual outputs (lists); more complex outputs like XML are not supported.

The C/C++ parser of Source-Navigator is a fuzzy parser, which means that it extracts only information that it needs for displaying various browser windows like class hierarchies or include graphs. It does not do preprocessing either. Consequently, the parser is very fast, but it does not provide low-level, detailed information about the subject system (i.e. it can be used only for architectural analyses).

Its database has a general format that is suitable for storing high-level information about projects written in different programming languages. For this reason, it provides no schema explicitly for C++ as Columbus does. However, it provides an API for accessing the extracted information that is stored in binary files that are similar to a relational database.

Command line operation is absent in Source-Navigator, so the front end can only be used within the user interface.

6 Conclusion and Future Work

In this paper we presented a schema for C++ and a reverse engineering tool that produces data based on it. It is free for scientific and educational purposes, our intention is to support academic persons in their research work. The framework relieves researchers of the burden of having to write parsers for different purposes and allows them to concentrate on their own concrete research topic.

The main advantage that this work offers is that it provides a modular *common schema*, a powerful *front end* that produces data according to the schema and an extendible *framework* which holds everything together. The framework already supports several popular tools like rigi and CodeCrawler, but it can be easily extended to support any arbitrary tool as well.

In the future we plan to extend the framework with additional export formats to support even more RE tools. We also plan to add a new extractor for supporting the Java programming language. This of course requires a schema for Java as well. On the front end side we are seeking to finish the implementation of adding expressions to the ASG. We also plan to support further C++ dialects and to develop an own preprocessor. Finally, we intend to publish the statement and expression parts of our C++ schema in a forthcoming paper.

References

- [1] M. Armstrong and C. Trudeau. Evaluating Architectural Extractors. In *Proceedings of WCRE'98*, pages 30–39, Oct. 1998.
- [2] Bell Canada Inc., Montréal, Canada. *DATRIX – Abstract semantic graph reference manual*, version 1.4 edition, May 2000.
- [3] B. Bellay and H. Gall. An Evaluation of Reverse Engineering Tool Capabilities. In *Software Maintenance: Research and Practice. 10.*, pages 305–331, Oct. 1998.
- [4] E. J. Chikofsky and J. H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. In *IEEE Software* 7, pages 13–17, Jan. 1990.
- [5] T. R. Dean, A. J. Malton, and R. Holt. Union Schemas as a Basis for a C++ Extractor. In *Proceedings of WCRE'01*, pages 59–67, Oct. 2001.
- [6] S. Demeyer, S. Ducasse, and M. Lanza. A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization. In *Proceedings of WCRE'99*, 1999.
- [7] J. Ebert, R. Gimnich, H. H. Stasch, and A. Winter. GUPRO – Generische Umgebung zum Programmverstehen, 1998.
- [8] R. Ferenc and Á. Beszédes. Data Exchange with the Columbus Schema for C++. In *Proceedings of CSMR 2002*, pages 59–66, Mar. 2002.
- [9] R. Ferenc, J. Gustafsson, L. Müller, and J. Paakki. Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa. In *Proceedings of SPLST 2001*, pages 58–70. University of Szeged, June 2001.
- [10] R. Ferenc, F. Magyar, Á. Beszédes, A. Kiss, and M. Tarkainen. Columbus – Tool for Reverse Engineering Large Object Oriented Software Systems. In *Proceedings of SPLST 2001*, pages 16–27. University of Szeged, June 2001.
- [11] R. Ferenc, S. E. Sim, R. C. Holt, R. Koschke, and T. Gyimóthy. Towards a Standard Schema for C/C++. In *Proceedings of WCRE'01*, pages 49–58. IEEE Computer Society, Oct. 2001.
- [12] Homepage of FrontEndART Software Ltd. <http://www.frontendart.com>.
- [13] R. Holt, A. E. Hassan, B. Lagu, S. Lapierre, and C. Leduc. E/R Schema for the Datrix C/C++/Java Exchange Format. In *Proceedings of WCRE'00*, Nov. 2000.
- [14] R. Holt, A. Winter, and A. Schürr. GXL: Towards a Standard Exchange Format. In *Proceedings of WCRE'00*, pages 162–171, Nov. 2000.
- [15] International Standards Organization. *Programming languages – C++*, ISO/IEC 14882:1998(E) edition, 1998.
- [16] IBM Jikes Project. <http://oss.software.ibm.com/developerworks/opensource/jikes>.
- [17] J. Mayrand and F. Coallier. System acquisition based on product assessment. In *Proceedings of ICSE'96*, 1996.
- [18] K. Mehlhorn and S. Naeher. Leda: A platform for combinatorial and geometric computing. In *Cambridge University Press*, 1997.
- [19] H. A. Müller. Criteria for Success of an Exchange Format. In *Workshop meeting, CASCON'98*, Nov. 1998.
- [20] H. A. Müller, K. Wong, and S. R. Tilley. Understanding Software Systems Using Reverse Engineering Technology. In *Proceedings of ACFAS*, 1994.
- [21] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, and A. Verkamo. Software metrics by architectural pattern mining. In *Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress)*, pages 325–332, 2000.
- [22] A. Quilici. Reverse Engineering of Legacy Systems: A Path Toward Success. In *Proceedings of ICSE'95*, pages 333–336, 1995.
- [23] C. Riva, M. Przybilski, and K. Koskimies. Environment for Software Assessment. In *Proceedings of ECOOP'99*, 1999.
- [24] The Source-Navigator IDE Homepage. <http://sources.redhat.com/sourcenav>.
- [25] The StarOffice Homepage. <http://www.sun.com/software/star/staroffice>.
- [26] A. Taivalsaari and S. Vaaraniemi. TDE: Supporting Geographically Distributed Software Design with Shared, Collaborative Workspaces. In *Proceedings of CAiSE'97*, LNCS 1250, pages 389–408. Springer Verlag, 1997.