University of Szeged
Department of Software Engineering

Modelling and Reverse Engineering C++ Source Code

Summary of the Ph.D. Thesis

of

Rudolf Ferenc

Supervisor:

Dr. Tibor Gyimóthy

Szeged
2004

Introduction

Software systems are rapidly growing and changing, so source code written today becomes legacy code in a very short period of time. This is mainly due to the swiftly changing market requirements and also to the ever-changing new technologies. The always tight deadlines often prevent the developers from properly releasing a product with up-to-date documentation (like design descriptions and source code comments). In such cases the only valid documentation is the source code itself. Some consequences of the above include lots of clones in the source code, dead code and fault-prone code, to mention only a few. *Reengineering* in general seeks to develop methods, techniques and tools to cure these problems and make program comprehension and maintenance easier. The first part of a reengineering process is called *reverse engineering*, which is defined as "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 or at a higher level of abstraction" [3].

The object-oriented paradigm has in recent years become the most popular one for designing and implementing large software systems. By now, many object-oriented systems have reached a state where they can be treated as legacy systems that need to be reengineered. In contrast with older programming languages like COBOL, for instance, reengineering methods for object-oriented programs are not yet fully elaborated. The really large and complex systems like telecom switching software and office suites are generally written in the C++ programming language. This object-oriented language is probably the most complex one and therefore, not surprisingly, it is least supported by reengineering methods and tools. This language offers us reengineers the most exciting challenges and opportunities for research.

To comprehend an unfamiliar software system we need to know many different things about it. We refer to this information as *fact*s 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 understand unknown source code is called a fact here. It is obvious that collecting facts by hand is only feasible when relatively small source codes are being investigated. Real-world systems that contain several million lines of source code can be only processed with the help of software tools.

In our approach tool-supported *fact extraction* is an automated process during which the subject system is analyzed on a file by file basis with analyzer tools to identify the source code's various characteristics and their interrelationships and to create some kind of representation of the extracted information. This information can afterwards be used by various reengineering tools like metrics calculators and software visualizers. The format of the outputs of the analyzer tools is unfortunately not standardized, almost every tool having its own format and this can lead to interoperability problems. Every application that would like to use the information gathered by other tools has to implement different convertors to gain access to the data.

This problematic situation was recognized by researchers and significant effort has been made to tackle this problem. One of the fruits of this grand effort is GXL [20] (Graph eXchange Language), which is a fine example of what can be achieved when we focus our energies. But using GXL itself is not enough. It offers a common medium for exchanging graphs (i.e. nodes and edges) with the help of XML, but does not describe how to represent various programming language-specific entities like C++ classes and functions. Researchers previously tackled this problem as well (e.g. [4–7; 16; 19; 22]), and different solutions were proposed but none of these is widely accepted and used. Without a common standard format (schema), smooth data exchange among different C++ reengineering tools is hard to achieve. 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* (in other words, a *model*) is an embodiment 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). In this work we present among other things a schema for the C++ language.

Plenty of work has been done in the field of re- and reverse engineering, which make use of the results of fact extraction. These include code measurements (different kinds of metrics), visualization, documentation and code comprehension. But relatively few papers deal with the actual process of fact extraction from C++ source code. We present methods with which automatic fact extraction can be achieved from real-world software systems. We developed a framework called *Columbus* [8–12; 14; 15] for supporting these methods and reverse engineering in general. The framework also takes care of fact representation, filtering and conversion to various formats to achieve tool interoperability. It is now used for research and education in many academic institutions around the world.

Going one step further in abstracting an analyzed software system we developed methods for recognizing design patters in our schema instances; and, this way, in the source code as well. Design patterns abstract practical solutions for frequently occurring design problems to an object-oriented format and are the most natural means when recovering the architectural design and the underlying design decisions from the software code.

To demonstrate that our methods could be used in practice we analyzed the source code of several versions of the popular internet suite *Mozilla* [27] and calculated various metrics from it. We used these metrics to predict the fault-proneness of the source code. We also compared the metrics of seven versions of Mozilla and showed how the predicted fault-proneness of the software changed during its development cycle.

I state four main results in the thesis, which are listed below:

1. **Schema for the C++ programming language.**
2. **C++ fact extraction process and framework.**
3. **Design pattern recognition in C++ source code.**
4. **Analysis of the fault-proneness of open source software.**

In the following sections I will briefly present these results and emphasize my own contributions at the end of each section.

1 Schema for the C++ programming language

This work was motivated by the observation that successful data exchange among reengineering tools is of crucial importance. This requires a common format so that the various tools (like front ends, metrics tools and clone detectors) can "talk" to each other. We designed a schema, called the *Columbus Schema for C++* [7; 12; 16], which captures information about source code written in the C++ programming language. The schema is modular, so it offers further flexibility for its extension or modification. The schema is fine-grained, representing practically every relevant fact about the source code so that logically equivalent source code can be generated from its instances. This schema seeks to fill a gap that was present in the literature of reengineering science since the design of the C++ language. Nobody has published a C++ schema before in such detail as we have. The reason for this is probably the extreme complexity of the C++ language. This first result of the thesis includes, in addition to the design of the schema, its implementation (which is used by our C++ analyzer tool as well) and algorithms for name resolution, type-checking, serialization, class diagram generation and call graph creation, to name only a few.

The descriptions of the schema is given using standard UML class diagrams, which permits its simple implementation and easy physical representation (e.g. using GXL). Despite the fact that it is not suitable for formal descriptions, we chose UML because it is the de facto standard in object-oriented design so the schema can be relatively easily comprehended by the users.

The structure of the schema

Owing to the great complexity of the C++ language, we decided to modularize our schema in a similar way to that proposed in the discussion part of [16]. This also opens up the possibility for its extension and modification. We divided the schema into six packages. These are the following:

- *base*: the package contains base classes and data types for the remaining parts of the schema.
- *struc*: this package models the main program entities according to their scoping structure (e.g. objects, functions and classes).
- *type*: the classes in this package are used to represent the types of the entities.
- *templ*: the package covers the representation of template parameter and argument lists.
- *statm*: the package contains classes that model the statements.
- *expr*: the classes in this package represent every kind of expression.

In this summary we show only one of the diagrams, probably the most interesting one, that of the *struc* package (see Figure 3). The explanation of the full schema can be found in the thesis.

We illustrate the use of the schema through an example instance of it. We will utilize the example C++ source code given in Figure 1. The schema instance for the example is given in Figure 2. We use an 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. We have simplified the diagram for clarity by omitting attributes such as line numbers, which are not necessary for comprehension.

```
template <typename T, int Size>
class Array {
  T arr[Size];
public:
  virtual const T& get(int idx) const {
    return arr[idx];
  };
  virtual void set(int idx, const T& val) {
    arr[idx] = val;
  }
};
```

Figure 1: C++ source code example.

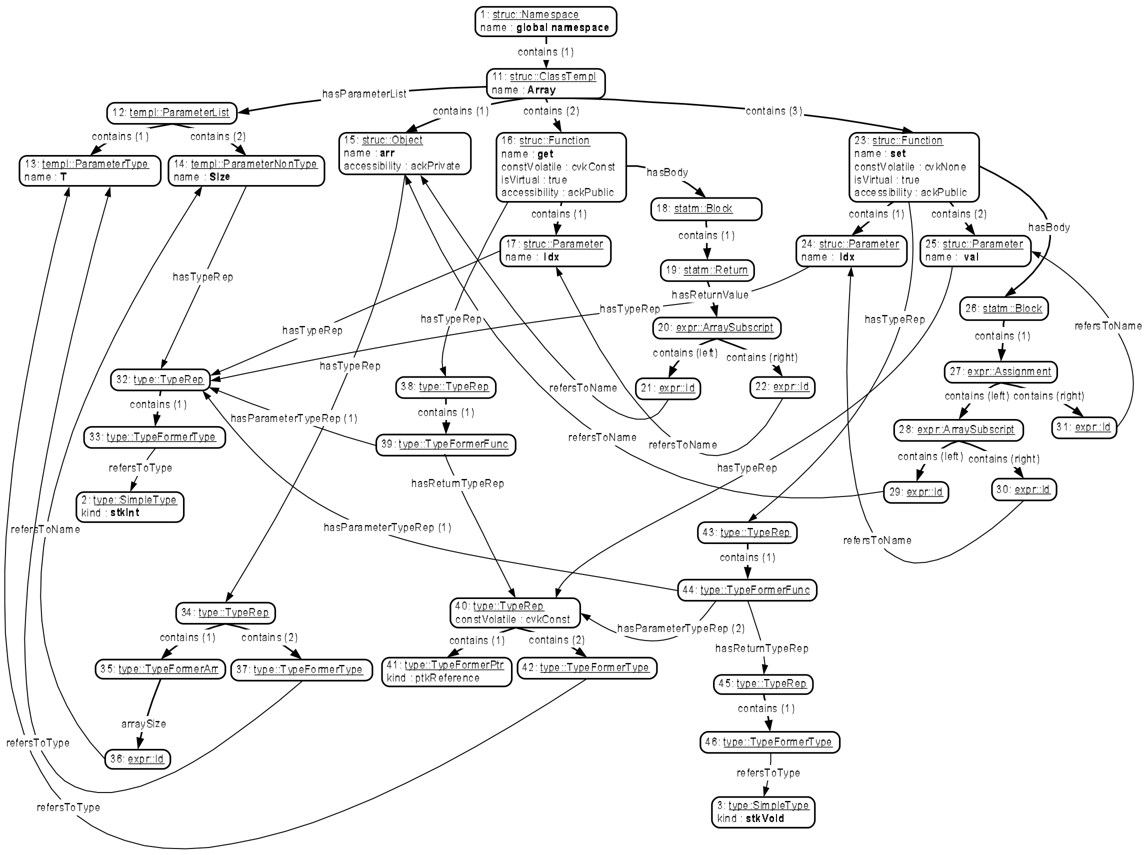


Figure 2: Schema instance (model) for the example.

Data exchange with other tools

Successful interchange of the data created by Columbus according to the Columbus Schema for C++ was achieved in several studies. The first application was created in cooperation with the Nokia Research Center for Nokia's proprietary UML design environment *TDE* [31]. We used the Columbus framework as the C++ analyzer front end of TDE. The built-up schema instances were converted to UML class diagrams and transferred to TDE though a COM interface. The *Maisa* [25; 26] project

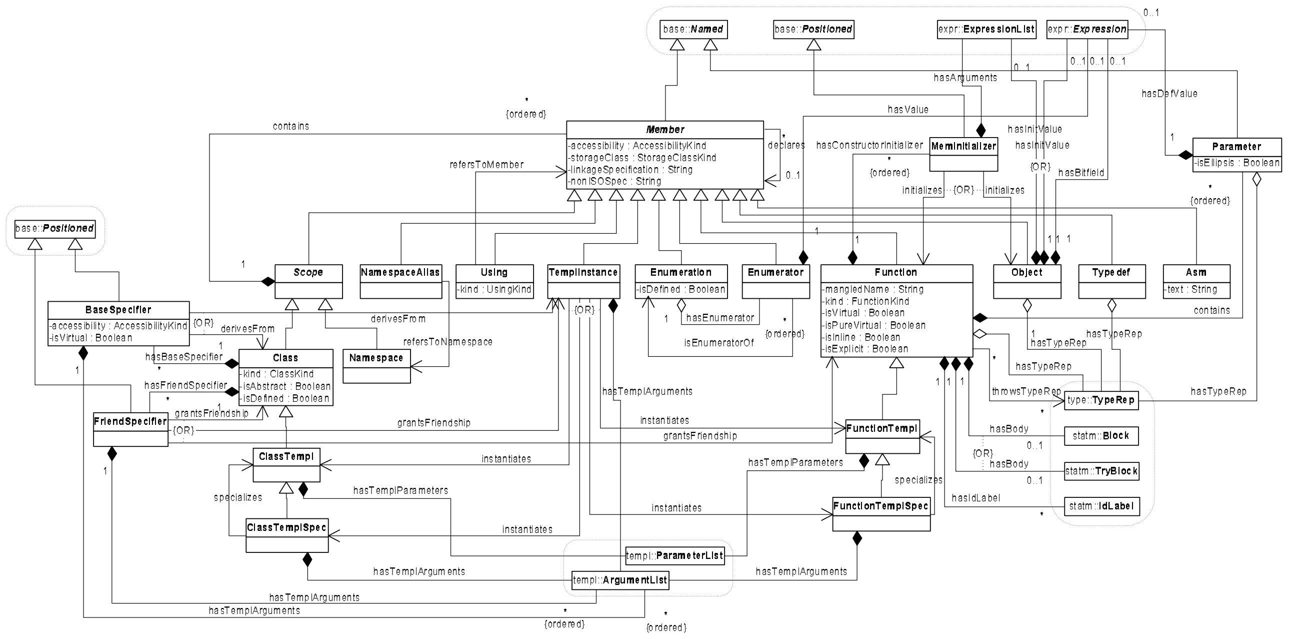


Figure 3: Class diagram of the **struc** package, one of the six packages of the schema.

of the University of Helsinki for recognizing standard Design Patterns [18] in C++ programs also successfully utilized the output created by Columbus. Another example of the schema's use was in a FAMOOS project with the Crocodile metrics tool [29]. An important application is the currently ongoing work on the exchange of data between Columbus and the GUPRO tool [6], which uses GXL as its input format. We also achieved a successful interchange with the *rigi* graph visualizer tool [24; 28]. Recently, we started a joint study with the University of Waterloo in Canada to visualize the Columbus schema instances in PBS, the Portable Bookshelf [17].

Own contribution

This work was motivated by the observation that successful data exchange is crucial for re- and reverse engineering tools. This requires a common format, one that can be used in various tools like front ends and metrics tools. A standard schema still has to be found. In this work I propose an exchange schema for the C++ language called the Columbus Schema for C++ for this purpose.

I designed the Columbus Schema for C++ and also implemented the schema which is part of the reverse engineering framework called Columbus. The implementation also contains algorithms for name resolution, type-checking, serialization, class diagram generation and call graph creation, to name only a few.

2 C++ fact extraction process and framework

Extracting facts from small programs is simple enough and can be done even by hand. The real challenge is to analyze real-world software systems that consist of several million lines of code. The literature lacked methods and the community did not have tools with which such a complex task could be efficiently performed. We present a process [15] that lists five key steps which have to be done to successfully carry out a fact extraction task from C++ source code. The process deals with important points such as handling configurations, linking the schema instances, filtering the data obtained and converting it to other formats to facilitate data exchange. We also present the Columbus reverse engineering framework [12] in detail, which readily supports the process. The framework is widely used at universities across the world, and so far over 600 downloads of the framework have been registered.

The fact extraction process

An outline of the process of fact extraction and presentation can be seen in Figure 4. The process consists of five consecutive steps where each step uses the results of the previous one. In the following, these steps will be specified in detail.

An important advantage of this approach is that the steps of the process can be performed *incrementally*, that is, if the partial results of the certain steps are available and the input of the step has not been altered, these results do not have to be regenerated.

Step 1: Acquiring project/configuration information

The source code of a software system is usually split into several files and these files are arranged into folders and subfolders. In addition, different preprocessing configurations can apply to them. The information on how these files are related to each other and what settings apply to them are usually stored in *makefile*s (when building software with the *make* tool) or in different *project files* (when using different *IDEs – Integrated Development Environments*). Acquiring this information from these files is a non-trivial task as different IDEs use different (and in most cases undocumented) file formats. Makefiles present additional difficulties: getting information out of them is extremely hard because they are not just suitable for code compilation; they can perform any arbitrary task. We introduce a so-called *compiler wrapping* method for using makefile information and two different approaches for handling IDE project files: *IDE integration* and *project file import*.

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

In this step the input files are processed one by one using the project/configuration information acquired in the first step. First, the preprocessing and extraction of preprocessing-related information are carried out by the preprocessor. Second, the preprocessed file is handed over to the C++ analyzer, which then analyzes the file and extracts C++ language-related information from it. Both tools create the corresponding schema instance files.

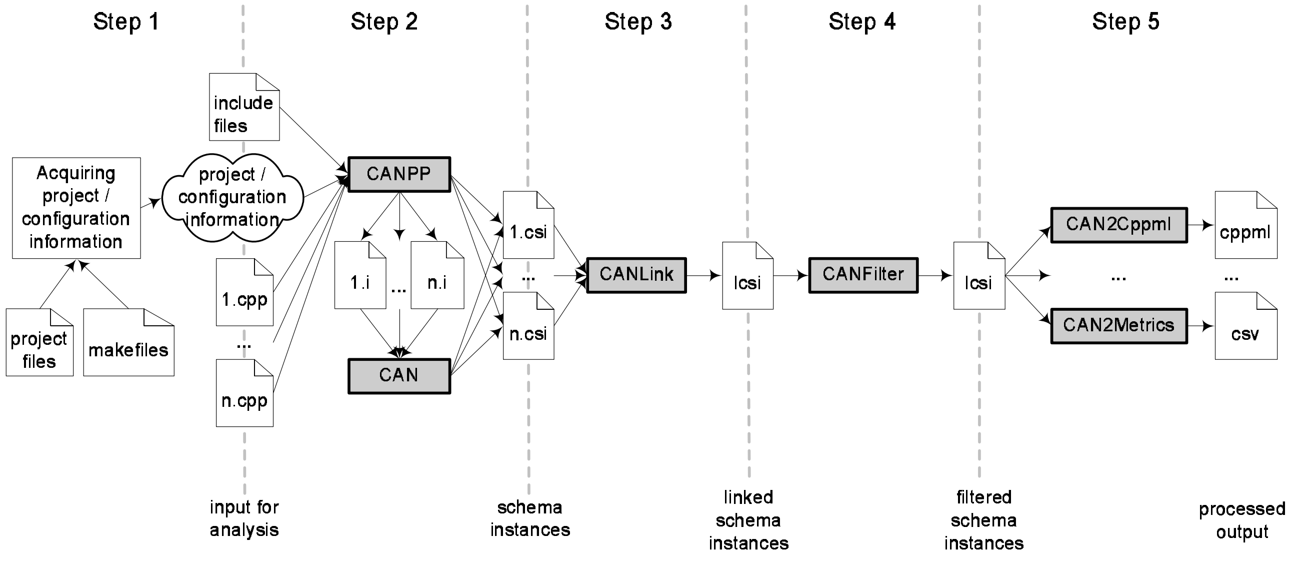


Figure 4: The fact extraction process.

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. This way, similar to real compiler systems that create different files which contain C++ entities that logically belong together (like libraries and executables), the related entities are grouped together.

Step 4: Filtering the schema instances

In the case of large projects the previous steps can produce large schema instances which contain huge amounts of extracted data. This is difficult to present in a useful way to the user. Different methods can help in solving this problem, like selecting only a particular module for further processing.

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 converted to different formats to be widely usable.

The Columbus framework

The fact extraction process is supported by our reverse engineering framework, which we present here. We developed the *Columbus Reverse Engineering Framework* [8–12; 14; 15] in an R&D project with the Nokia Research Center. The main motivation behind developing the 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. The Columbus REE is not limited to the C++ language; all C++ specific tasks are performed by using 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

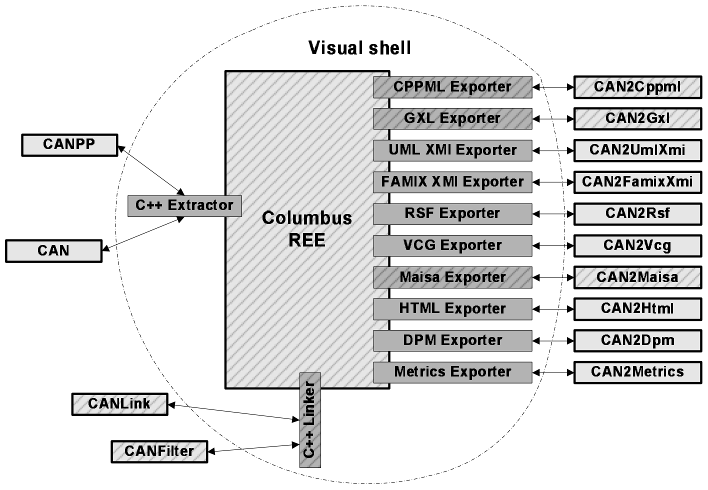


Figure 5: The C++ specific configuration of Columbus REE.

and reverse engineering tasks as well. The actual analysis and fact processing is done by different command line tools invoked by Columbus REE (these tools run on both Windows and GNU/Linux operating systems). The Columbus framework currently includes the following tools, which are listed and presented in the following:

- *Columbus REE*. The graphical user interface shell of the framework.
- *Columbus IDE Add-ins*. Graphical user interface shell functionality of the framework in IDEs.
- *CANGccWrapper toolset*. GCC compiler-wrapper tool with various helper scripts.
- *CANPP*. C/C++ preprocessor and preprocessing schema instance builder tool.
- *CAN*. C++ analyzer and schema instance builder tool.
- *CANLink*. C++ schema instance linker tool.
- *CANFilter*. C++ schema instance filter tool.
- *CAN2**. C++ schema instance converter and processor tools.

Columbus REE (Reverse Engineering Environment)

The Columbus REE is a general reverse engineering environment. All C++ specific tasks are performed by its plug-in modules which are categorized as *extractor-*, *linker-* and *exporter plug-ins*. The reader should look at Figure 5 for the actual C++ specific configuration.

Columbus IDE Add-ins

A significant part of the Columbus REE is engaged in managing the project (configuration) of the files to be analyzed. This work is also done by the popular IDEs, so it was logical to package the remaining part of the Columbus REE – the part that deals with the extraction process – into a separate component called *Columbus DLL*, which communicates with different so-called *Columbus Add-ins* for IDEs. These add-ins are basically plug-ins that extend the functionality of the IDEs.

Compiler wrapping

The make tool and the makefiles represent a powerful pair for configuring and building software systems. Makefiles may contain not only references to files to be compiled and their settings, but also various commands like those invoking external tools. These possibilities are a headache for reverse engineers because every action in the makefile must be simulated in the reverse engineering tool.

We approached this problem from the other end and solved it by "wrapping" the compiler. This means that we hide the original compiler by using a wrapper toolset. This toolset hides the original compiler by changing the PATH environment variable to point to our scripts carrying the names of the executable files of the compiler. If the original compiler should be invoked, one of our wrapper scripts will start instead of it, which, after running the original compiler, also starts our analyzer tools (through the program *CANGccWrapper*). We successfully used this approach with GCC for extracting information from the open source real-world software system called *Mozilla*. This proves the operability of this method.

Filtering

We offer tree methods in the *CANFilter* tool which help in filtering the schema instances: *filtering using C++ entity categories*, *by input source files* and *according to scopes*.

Schema instance conversions

Because different C++ re- and reverse engineering tools use different schemas for representing their data, the schema instances can be converted to other formats to achieve tool interoperability. We can convert our schema instances to several formats. These formats are: *CPPML*, *GXL*, *UML XMI*, *FAMIX XMI*, *RSF*, *VCG*, *Maisa* and *HTML*.

Derived outputs

We also use our schema instances to prepare different derived outputs. This means applying further computations on the instances. The following outputs are available: *metrics, design pattern mining* and *SourceAudit*.

Own contribution

I created an approach that lists five steps which have to be done to successfully carry out a fact extraction task. I developed some significant parts of the Columbus reverse engineering framework, which supports the process (among others the parts which are filled with striped lines in Figure 5). The framework contains various tools and extension mechanisms so it relieves researchers of the burden of having to write parsers for different purposes and allows them to focus on their own particular research topic.

I designed and implemented the following parts of the framework: *Columbus REE*, *C++ linker plug-in*, *CPPML/GXL/Maisa exporter plug-ins*, *Columbus IDE Add-ins*, *CANLink*, *CANFilter* and the *CANGccWrapper* toolset, not to mention the following conversion algorithms: *CPPML – C++ Markup Language* (including the design of the language), *GXL – Graph eXchange Language* and *Maisa* (the algorithms were implemented within the *CAN2Cppml*, *CAN2Gxl* and *CAN2Maisa* tools). I also participated in the design of the *Design Pattern Miner* module.

3 Design pattern recognition in C++ source code

Existing reverse engineering tools and methods produce a wide variety of abstract software representations. A natural strategy of abstracting object-oriented programs is to represent them as a set of UML diagrams. While the automatic generation of UML diagrams from software code is supported by a number of reverse engineering tools, recognizing *design patterns* [18] is, currently, almost totally without advanced tool support. Design patterns are the most natural and useful assets when recovering the architectural design and the underlying design decisions from the software code. The third main result of the thesis offers two methods for discovering design pattern instances in C++ source code. First, we present a method [13] and toolset for recognizing design patterns with the integration of Columbus and *Maisa* [25; 26]. The method combines the fact extraction capabilities of the Columbus framework with the pattern mining ability of Maisa. Second, we present a new solution to the problem of pattern detection with a sophisticated, parameterizable, fast graph matching algorithm that recognizes design patterns in our schema instances [1]. It includes the detection of call delegations, object creations and operation redefinitions. These are the elements that identify pattern occurrences more precisely. The pattern descriptions are stored in our new XML-based format, the *Design Pattern Markup Language* (*DPML*). This gives the user the freedom to modify the patterns, adapt them to his or her own needs, or create new pattern descriptions.

Integration of Columbus and Maisa

Maisa is a software tool for the analysis of software architectures, developed in a research project at the University of Helsinki. The real purpose of Maisa is to analyze design level UML diagrams and compute architectural metrics for early quality prediction of the software system. In addition, Maisa looks for instances of design patterns in UML diagrams. The level of abstraction is crucial for the success of the analysis: the more detailed the diagrams are, the more accurate the results will be. Therefore design pattern mining at the detailed level of source code is a promising way of improving the practical usability of Maisa.

Because Maisa is implemented entirely in Java, it cannot access the schema instance directly in memory, so we have instead chosen a trivial way of connecting the two tools: an exporter plug-in in Columbus creates a file in Maisa's input file format, which can then be loaded and further processed by Maisa. The file created by Columbus contains the necessary reverse engineered information as PROLOG facts about the main program entities (classes, attributes, etc.) and their relationships (subclassing, composition, etc.).

Experiments

This design pattern recognition approach was tested with simple experiments. We implemented some of the standard design patterns in C++ (Singleton, Visitor, Builder, Factory Method, Prototype, Proxy and Memento) and afterwards used Columbus to analyze the code and to create the input files for Maisa. Finally, Maisa was applied to recognize the design patterns, which succeeded in all these cases. While our initial experiments show the potential capability of the pattern recognition approach, more extensive experiments with real-world software must be carried out to verify the real power of the method.

Pattern miner algorithm in Columbus

Most of the existing approaches of recognizing design patterns in source code only search for *basic pattern structures*. We developed a new method which overcomes this problem by using as much useful information from the source as possible. First, we analyze the C++ source code with the Columbus framework, which then builds the appropriate schema instance. Next, we load our pattern descriptions which are stored in DPML files. Finally, our algorithm binds classes found in the source code to pattern classes that are part of the pattern description and checks whether they are related in the way that is described in the pattern. Here we use composition, aggregation, association and inheritance relationships for classes, and call delegation, object creation and operation redefinition (overriding) relationships for operations. The results of the function-body analysis is what gives us more precision compared to other approaches in detecting design pattern occurrences in the source code.

Experiments

We performed experiments on four real-word, publicly available C++ projects listed below:

- *Jikes* [21]. Open source Java compiler system from IBM.
- *LEDA* [23]. Library of efficient data types and algorithms.
- *StarOffice Calc* [30]. The spreadsheet application of StarOffice, a large C++ project that consist of 6,307 source files (more than 1.2 million non-preprocessed non-empty lines of code).
- *StarOffice Writer* [30]. The word processing application of StarOffice, a large C++ project that consist of 6,794 source files (more than 1.5 million non-preprocessed non-empty lines of code).

Table 1 shows the number of different design pattern instances found in the test projects. For most design patterns we also wrote their "soft" versions in which we slightly relaxed the original specifications in [18] (for instance, we did not demand that some classes be abstract). Except for LEDA, the other three are more recent projects, and it was noticed that there were many more patterns in these projects than in LEDA.

Own contribution

I introduced two methods for discovering design pattern instances in C++ source code.

First, I presented a method and tool set for recognizing design patterns with the integration of Columbus and Maisa. The method combines the fact extraction capabilities of the Columbus reverse engineering framework with the clause-based pattern mining ability of Maisa. I analyzed the C++ code with Columbus, and I wrote the schema instance converter algorithm to export the collected facts to a form understandable to Maisa. This form is a clause-based design notation in PROLOG.

Second, I gave a new solution to the problem of pattern detection which includes the detection of call delegations, object creations and operation redefinitions. These are the elements that identify pattern occurrences more precisely. The pattern descriptions are stored in an XML-based format designed by me, the Design Pattern Markup Language (DPML). This gives the user the freedom to

Statistics	Jikes	LEDA	Calc	Writer
Abstract Factory	-	-	-	-
Builder	-	-	2	7
Builder soft	-	-	17	9
Factory Method	-	-	-	-
Factory Method soft	-	-	1	9
Prototype	1	-	-	1
Prototype soft	1	-	-	1
Singleton	-	-	-	-
Adapter Class	-	-	-	16
Adapter Class soft	-	-	13	16
Adapter Object	54	-	27	62
Adapter Object soft	62	-	153	135
Bridge	-	-	-	-
Bridge soft	-	-	73	80
Decorator	-	-	-	-
Decorator soft	-	-	-	-
Proxy	36	-	-	4
Proxy soft	44	-	-	5
Chain of Responsibility	-	-	-	-
Iterator	-	-	-	-
Iterator soft	-	-	1	-
Strategy	4	1	10	5
Strategy soft	12	2	20	32
Template Method	5	-	94	101
Visitor	-	-	-	-
Visitor soft	-	-	-	5
Sum total	235	6	442	525

Table 1: Number of design pattern instances found.

modify the patterns, adapt them to his or her own needs, or create new pattern descriptions. Then the method was tested on four public-domain projects.

4 Analysis of the fault-proneness of open source software

Open source software systems are becoming evermore important these days. Many large companies are investing in open source projects and lots of them are also using this kind of software in their own work. As a consequence, many of these projects are being developed rapidly and quickly become very large. But because open source software is often developed by volunteers in their spare time, the quality and reliability of the code may be uncertain. Various kinds of code measurements can be quite helpful in obtaining information about the quality and fault-proneness of the code. We calculated the object-oriented metrics validated in [2] for fault-proneness detection from the source code of the open source internet suite *Mozilla* [27] with the help of our compiler wrapper toolset. We then compared our results with those presented in [2]. One of our aims was to supplement their work with metrics obtained from a real-world software system. We also compared the metrics of seven versions of Mozilla (see Table 2 for some size metrics of the analyzed versions) to see how the predicted fault-proneness of the software changed during its development cycle.

ver.	NCL	TLOC	TNM	TNA	Definition of the metrics
1.0	4,770	1,127,391	69,474	47,428	
1.1	4,823	1,145,470	70,247	48,070	NCL: Number of Classes.
1.2	4,686	1,154,685	70,803	46,695	TLOC: Total number of non-empty lines of code.
1.3	4,730	1,151,525	70,805	47,012	TNM: Total Number of Methods in the system.
1.4	4,967	1,171,503	72,096	48,389	TNA: Total Number of Attributes in the system.
1.5	5,007	1,169,537	72,458	47,436	
1.6	4,991	1,165,768	72,314	47,608	

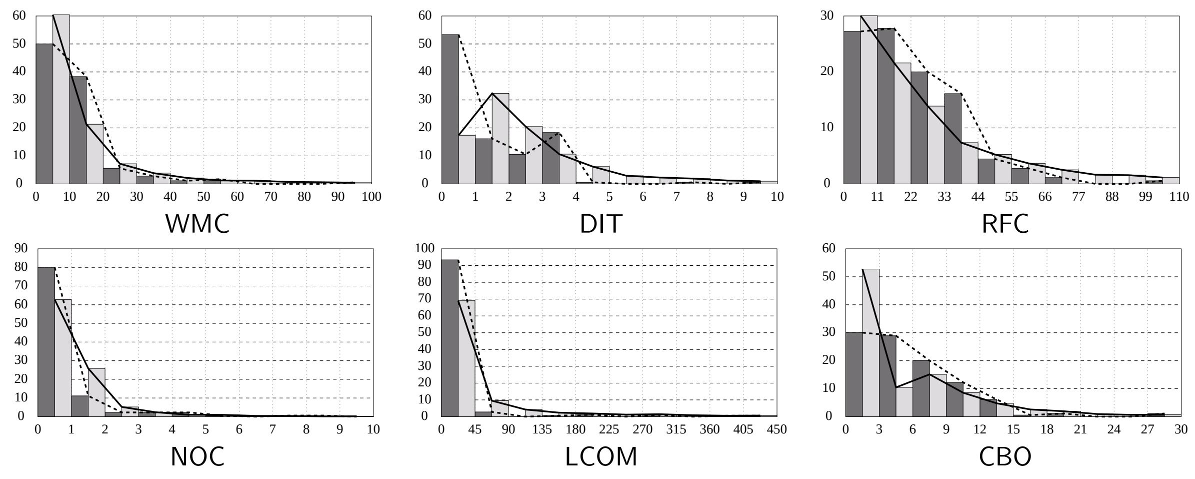
Table 2: System-level metrics of the analyzed Mozilla versions.

It should be mentioned here that we performed complete analyses of the seven versions of Mozilla and built up the full schema instances of them, which can be used for any re- and reverse engineering task like architecture recovery and visualization. Here we just used them for calculating metrics. We did not classify the metrics according to their purpose or usability, instead we used the results of Basili *et al.* and studied the metrics according to [2].

Basili *et al.* studied object-oriented systems written by students in C++. They carried out an experiment in which they set up eight project groups consisting of three students each. Each group had the same task – to develop a small/medium-sized software system. Since all the necessary documentation (for instance, reports about faults and their fixes) was available, they could search for relationships between the fault density and metrics. They used a metrics suite that consisted of six metrics and analyzed the distribution and correlations between them. Afterwards, they made use of logistic regression to analyze the relationship between metrics and the fault-proneness of classes. We refer to their projects as the *reference project*. In the following we list the six metrics they investigated.

- WMC – Weighted Methods per Class.
- DIT – Depth of Inheritance Tree.
- RFC – Response For a Class.
- NOC – Number Of Children.
- LCOM – Lack of Cohesion on Methods.
- CBO – Coupling Between Object classes.

Comparison of reference project with Mozilla



The X axes represent the values of the metrics. The Y axes represent the percentage of the number of classes having the corresponding metrics value.
The darker columns represent the original values of the reference project from [2], while the brighter ones represent the values calculated for Mozilla 1.6.

Figure 6: Distribution of the metrics of the reference project and Mozilla.

Here we compare the metrics calculated for Mozilla 1.6 with those of the reference project. Figure 6 shows a comparison of the *distribution of the metrics*. It can be seen that the distributions of WMC, RFC, NOC and LCOM are quite similar in both cases. On the other hand, the distributions of DIT and CBO are quite different.

Ref. | **Moz.**	WMC		DIT		RFC	
Maximum	99.00	**337.00**	9.00	**33.00**	105.00	**1,074.00**
Minimum	1.00	**0.00**	0.00	**0.00**	0.00	**0.00**
Median	9.50	**7.00**	0.00	**2.00**	19.50	**21.00**
Mean	13.40	**14.12**	1.32	**2.39**	33.91	**48.95**
Standard deviation	14.90	**22.16**	1.99	**2.90**	33.37	**81.99**
Ref. | **Moz.**	NOC		LCOM		CBO	
Maximum	13.00	**1,213.00**	426.00	**55,198.00**	30.00	**70.00**
Minimum	0.00	**0.00**	0.00	**0.00**	0.00	**0.00**
Median	0.00	**0.00**	0.00	**15.00**	5.00	**2.00**
Mean	0.23	**1.06**	9.70	**273.82**	6.80	**5.11**
Standard deviation	1.54	**17.44**	63.77	**1,597.53**	7.56	**7.49**

The bold numbers represent the values of Mozilla 1.6, while the normal ones are the values of the reference project.

Table 3: Descriptive statistics of the classes in the reference project and Mozilla.

We also compared the statistical information we obtained with the measurements. Table 3 shows basic statistical information about the two systems. The *Minimum* values here are almost the same but the *Maximum* values have increased dramatically. This is not surprising because Mozilla has about 30 times more classes than the reference project. Since LCOM is proportional to the square of the size (number of methods) of a class, its very large value is to be expected. In Mozilla there are about five thousand classes, so the extremely high value of NOC may seem surprising at first. But the second biggest value of NOC is just 115, hence we think that the class with the largest value is probably a common base class from which almost all other classes are inherited. *Median* and *Mean* express "a kind of average" and they are more or less similar, except for the LCOM value (similar to the Maximum value). Since in Mozilla there are many more classes and these are more variegated, the metrics change over a wider range. The *Standard Deviation* values suggest this bigger variety of the classes.

Reference	WMC	DIT	RFC	NOC	LCOM	CBO
WMC	1	0.02	**0.24**	0	**0.38**	0.13
DIT		1	0	0	0.01	0
RFC			1	0	0.09	**0.31**
NOC				1	0	0
LCOM					1	0.01
CBO						1

Mozilla	WMC	DIT	RFC	NOC	LCOM	CBO
WMC	1	0.16	**0.53**	0	**0.64**	**0.39**
DIT		1	**0.54**	0	0.08	**0.23**
RFC			1	0	**0.31**	**0.51**
NOC				1	0	0
LCOM					1	0.16
CBO						1

The bold numbers denote significant correlations.

Table 4: Correlations between the metrics of the reference project and Mozilla.

Basili *et al.* [2] also calculated the *correlations* of the metrics (see Table 4). They found that the linear Pearson's correlation between the object-oriented metrics studied are, in general, very weak. Three coefficients of determination appear somewhat more significant, but they conclude that these metrics are mostly statistically independent. We also calculated the same correlations in the case of Mozilla 1.6 but we obtained different results. NOC is independent of the other metrics just like in the reference project, but all the others have some correlation with each other. There are three very weak correlations but the rest represent more or less significant correlations. What is more, there are some very large values (for instance, between WMC and LCOM), so it follows that these metrics are not totally independent and represent redundant information. This is surprising because Basili *et al.* found that some of these metrics could be used for detecting fault-proneness while the others were not significant.

Studying the changes in Mozilla's metrics

Basili *et al.* drew up six hypotheses (one for each metric) that represent the expected connection between the metrics and the fault-proneness of the code [2]. They tested these hypotheses and found that some of the metrics were very good predictors, while others were not.

We present all the hypotheses and conclusions about the "goodness" of the metrics for detecting fault-proneness as stated in [2] and examine the changes in Mozilla based on their conclusions.

WMC hypothesis: *"A class with significantly more member functions than its peers is more complex and, by consequence, tends to be more fault-prone."* The WMC was found to be somewhat significant in [2]. In Mozilla the rate of classes with a large WMC value decreased slightly but not significantly. We can only say that Mozilla did not get worse according to this metric.

DIT hypothesis: *"A class located deeper in a class inheritance lattice is supposed to be more fault-prone because the class inherits a large number of definitions from its ancestors."* The DIT was found to be very significant in [2], which means that the larger the DIT, the larger the probability of fault-proneness. In Mozilla the rate of classes with seven or more ancestors increased slightly, but the number of these classes is tiny compared to the classes as a whole. On the other hand, the rate of classes which have no ancestors or only one or two increased significantly, while the rate of classes with more than two but fewer than seven ancestors decreased markedly. This suggests that in more recent versions of Mozilla there might be fewer faults.

RFC hypothesis: *"Classes with larger response sets implement more complex functionalities and are, therefore, more fault-prone."* The RFC was shown to be very significant in [2]. The larger the RFC, the larger the probability of fault-proneness. In Mozilla the rate of classes whose RFC value is larger than ten decreased (more than 70% of the classes fall into this group), while the rate of the rest of the classes increased. Overall, this suggests an improvement in quality (so it is less fault-prone).

NOC hypothesis: *"We expect classes with large number of children to be more fault-prone."* The NOC appeared to be very significant but the observed trend is contrary to what was stated by the hypothesis. The larger the NOC the lower the probability of fault-proneness [2]. In Mozilla the number of classes with three or more children is negligible and it did not change significantly. Hence, we only examined the remaining classes. The rate of classes with no or two children decreased while the rate of classes with one child increased. According to this, Mozilla slightly improved in this respect.

LCOM hypothesis: *"Classes with low cohesion among its methods suggests an inappropriate design which is likely to be more fault-prone."* The LCOM was stated to be insignificant in [2], but according to the hypothesis Mozilla got slightly worse because the rate of classes with the value eleven or more increased slightly while the rest remained about the same.

CBO hypothesis: *"Highly coupled classes are more fault-prone than weakly coupled classes."* The CBO was said to be significant in [2] but it is hard to say anything about Mozilla using this metric. The rate of classes whose CBO value is one, two or three increased and the rate of classes whose CBO value is four, five or six decreased, which is good and may suggest an increase in quality. On the other hand, the rate of classes with large CBO values increased, which suggests more faults.

Own contribution

This work describes three key results, where the first one and the third one are my own contributions: (1) I showed that my compiler wrapper toolset realizes the fact extraction process in practice on real-world software; (2) using the collected facts, object-oriented metrics were calculated and a previous work [2] was supplemented with measurements made on the real-world software package called Mozilla; and (3) using the calculated metrics I studied how Mozilla's predicted fault-proneness changed over seven versions covering one and a half years of development.

$\mathcal{N}o.$	[16]	[7]	[12]	[13]	[1]	[15]
1.	●	●	●			
2.			●			●
3.				●	●	
4.						●

Table 5: The relation between the thesis topics and the corresponding publications.

Conclusions

With the fact extraction process and framework I achieved it is now possible to reverse engineer the source code of large, real-world software systems written in the C++ programming language. What is more, this can be done without having to modify any source code, not even the makefiles or project files.

During the fact extraction process our framework prepares a model of the analyzed C++ system according to my well-defined schema. Having such a schema enables the community and industry to improve existing reengineering tools and develop new ones that can seamlessly exchange information about the software system being studied. I supplemented our framework with an application programming interface based on the schema with which the extracted information can be easily accessed and used. Output files in various formats can also be produced to promote tool interoperability.

Containing various algorithms for producing derived outputs – like object-oriented metrics – as well, the framework provides a complete solution for reverse engineering C++ code so that it removes the burden of having to write parsers for different purposes and permits researchers to focus on their own research topic.

I have successfully utilized our fact extraction and representation technology to achieve two further goals. First, I developed new methods for recognizing design pattern occurrences in C++ source code and second, I analyzed several versions of the open-source internet suite Mozilla to study the changes in its predicted fault-proneness and quality.

Lastly, Table 5 above summarizes which publications cover which results of the thesis.

Acknowledgements

First of all, I would like to thank my supervisor, Dr. Tibor Gyimóthy for supporting my work with useful comments and letting me work at an inspiring department, the Department of Software Engineering. I would also thank all my colleagues and friends Árpád Beszédes, Ferenc Magyar, László Vidács, Fedor Szokody, Gábor Lóki, István Siket, Péter Siket, Zsolt Balanyi and László Müller – who all participated in developing and testing the Columbus framework – for striving so hard on this application. I would also like to express my gratitude to David Curley for scrutinizing and correcting this thesis from a linguistic point of view and András Kocsor for his useful advice.

Last, but not least, my heartfelt thanks goes to my wife Györgyi for providing a secure family background during the time spent writing this work.

Rudolf Ferenc, November 20, 2004.

References

[1] Zsolt Balanyi and Rudolf 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, September 2003.

[2] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. In *IEEE Transactions on Software Engineering*, volume 22, pages 751–761, October 1996.

[3] E. J. Chikofsky and J. H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. In *IEEE Software 7*, pages 13–17, January 1990.

[4] Thomas R. Dean, Andrew J. Malton, and Ric Holt. Union Schemas as a Basis for a C++ Extractor. In *Proceedings of WCRE'01*, pages 59–67, October 2001.

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

[6] J Ebert, R Gimnich, H H Stasch, and A Winter. GUPRO – Generische Umgebung zum Programmverstehen, 1998.

[7] Rudolf Ferenc and Árpád 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, March 2002.

[8] Rudolf Ferenc and Árpád Beszédes. Az Objektumvezérelt Szoftverek Elemzése. In *VIII. Országos (Centenáriumi) Neumann Kongresszus Elõadások és Összefoglalók*, pages 463–474. Neumann János Számítógép-tudományi Társaság, October 2003.

[9] Rudolf Ferenc, Árpád Beszédes, and Tibor Gyimóthy. Extracting Facts with Columbus from C++ Code. In *Tool Demonstrations of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 4–8, March 2004.

[10] Rudolf Ferenc, Árpád Beszédes, and Tibor Gyimóthy. Fact Extraction and Code Auditing with Columbus and SourceAudit. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, page 513. IEEE Computer Society, September 2004.

[11] Rudolf Ferenc, Árpád Beszédes, and Tibor Gyimóthy. *Tools for Software Maintenance and Reengineering*, chapter Extracting Facts with Columbus from C++ Code, pages 16–31. Franco Angeli Milano, 2004.

[12] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor 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, October 2002.

[13] Rudolf Ferenc, Juha Gustafsson, László Müller, and Jukka Paakki. Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa. *Acta Cybernetica*, 15:669–682, 2002.

[14] Rudolf Ferenc, Ferenc Magyar, Árpád Beszédes, Ákos Kiss, and Mikko Tarkiainen. Columbus – Tool for Reverse Engineering Large Object Oriented Software Systems. In *Proceedings of the 7th Symposium on Programming Languages and Software Tools (SPLST 2001)*, pages 16–27. University of Szeged, June 2001.

[15] Rudolf Ferenc, István Siket, and Tibor Gyimóthy. Extracting Facts from Open Source Software. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, pages 60–69. IEEE Computer Society, September 2004.

[16] Rudolf Ferenc, Susan Elliott Sim, Richard C Holt, Rainer Koschke, and Tibor Gyimóthy. Towards a Standard Schema for C/C++. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, pages 49–58. IEEE Computer Society, October 2001.

[17] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The Software Bookshelf. In *IBM Systems Journal*, volume 36, pages 564–593, November 1997.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Pub Co, 1995.

[19] Ric Holt, Ahmed E. Hassan, Bruno Lagu, Sbastien Lapierre, and Charles Leduc. E/R Schema for the Datrix C/C++/Java Exchange Format. In *Proceedings of WCRE'00*, November 2000.

[20] Ric Holt, Andreas Winter, and Andy Schürr. GXL: Towards a Standard Exchange Format. In *Proceedings of WCRE'00*, pages 162–171, November 2000.

[21] IBM Jikes Project.
`http://oss.software.ibm.com/developerworks/opensource/jikes`.

[22] E Mamas and K Kontogiannis. Towards Portable Source Code Representations Using XML. In *Proceedings of WCRE'00*, pages 172–182, November 2000.

[23] K. Mehlhorn and S. Naeher. LEDA: A Platform for Combinatorial and Geometric Computing. In *Cambridge University Press*, 1997.

[24] Hausi A Müller, Kenny Wong, and Scott R Tilley. Understanding Software Systems Using Reverse Engineering Technology. In *Proceedings of ACFAS*, 1994.

[25] L. Nenonen, J. Gustafsson, J. Paakki, and A.I. Verkamo. Measuring Object-Oriented Software Architectures from UML Diagrams. In *Proceedings of the 4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, pages 87–100, 2000.

[26] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, and A.I. Verkamo. Software Metrics by Architectural Pattern Mining. In *Proceedings of he International Conference on Software: Theory and Practice (16th IFIP World Computer Congress).*, pages 325–332, 2000.

[27] Christian Robottom Reis and Renata Pontin 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, February 2002.

[28] The Rigi Homepage. `http://www.rigi.csc.uvic.ca`.

[29] Claudio Riva, Michael Przybilski, and Kai Koskimies. Environment for Software Assessment. In *Proceedings of ECOOP'99*, 1999.

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

[31] 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.