Qualoss

(contract # 033547)
Sponsored through Framework Programme Sixth (Call 5) by

Document Information

Version: **2.0**
Date : Feb 1, 08
Pages : 76

Owning Partner:
CETIC

Author(s):
Jean-Christophe DEPREZ (CETIC), Jose Ruiz (ADACORE), Israel HERRAIZ (URJC), Carlos GARCIA CAMPOS (URJC)

Reviewer(s):
Fraunhofer IESE

To:
CONSORTIUM

Purpose of distribution:
Ready for submission to E.C.

The QUALOSS Consortium consists of: CETIC (BE), Facultés Universitaires Notre Dame de la Paix à Namur (BE), Universidad Rey Juan Carlos (ES), Fraunhofer IESE (DE), ZEA Partners (BE), MERIT (NL), AdaCore (FR), PEPITe (BE)

Printed on 02/01/08 at 11:47:00 AM

Status:

[ ] Draft
[ ] To be reviewed
[ ] Proposal
[X] Final/Released

Confidentiality:

[X] Public - Intended for public use
[ ] Restricted - Intended for QUALOSS consortium only
[ ] Confidential - Intended for individual partner only

Deliverable ID: **D1.1**

Title:

Evaluation Report on Existing Tools and Existing F/OSS repositories

Deliverable: D1.1

Title: Evaluation Report on Existing Tools and Existing F/OSS repositories

Executive Summary:

The strategic objective of the QUALOSS project is to enhance the competitive position of the European software industry by providing methodologies and tools for improving their productivity and the quality of their software products. To this end, QUALOSS plan on developing a tooled method for assessing the evolvability and robustness of Free *libre* Open-Source Software (F/OSS). In turn, this will facilitate the integration and acquisition process of F/OSS in existing systems.

This first workpackage (WP1) performs requirements analysis through prototyping. In particular, requirements analysis is approached from two directions, top-down and bottom-up. task 1.2 is top down, while task 1.1 takes a bottom up angle. The results of tasks 1.1 and 1.2 will later be merged during tasks 1.3, which identifies how to measure the quality characteristics highlighted in task 1.2 using the tools and techniques identified in tasks 1.1. This deliverable describes the outcome of task 1.1.

After an introduction in Section 1, a short glossary is presented in Section 2.

Section 3 enumerates different sources containing data related to FlOSS projects. Beside F/OSS product releases, version control repositories, and bug tracking data, several other lesser known sources are presented such as mailing list archives and vulnerability databases.

Section 4 reviews existing tools for analyzing the data from the source mentioned in Section 3.

Section 5 presents advanced analysis of potential interest for the QUALOSS quality models.

Concluding remark are in Section 6.

Deliverable: D1.1

CHANGE LOG

Ver.	Date	Author	Description
0.1	18/09/2006	Jean-Christophe DEPREZ	Initial Proposal for Structure (section assignment and timeline)
0.2	24/10/2006	Jose RUIZ	Add Ada tool names
0.3	23/11/2006	Israel HERRAIZ	Section 2 updated and finished. Needs to be reviewed
0.4	01/12/2006	Carlos GARCIA CAMPOS	Repository-Data Analysis Tools section finished (section 3.2)
0.5	15/02/2007	Jose RUIZ	Update static analysis tools for Ada
0.6	16/02/2007	Jose RUIZ	Update dynamic analysis tools for Ada
0.7	01/03/2007	Jean-Christophe DEPREZ	Added C++, Python
0.7	12/03/2007	Marcus CIOLKOWSKI, Martín SOTO	First Review
0.8	17/03/2007	Jean-Christophe DEPREZ	Added Sections 5 on advanced analyses + conclusion and Exec Summary
0.9	19/03/2007	Martín SOTO	Second Review
0.10	20/03/2007	Jean-Christophe DEPREZ	Integrated Comments from Second Review
0.11	21/03/2007	Carlos GARCIA CAMPOS	Filled some missing fields in Section 4.2
1.0	21/03/2007	Jean-Christophe DEPERZ	Final check to make deliverable ready for submission
1.1	21/01/2008	Israel HERRAIZ	Some more references and comments for the Advanced Analysis section
1.2	31/01/2008	Jean-Christophe DEPREZ	Review of references and comments added By Israel.

TABLE OF CONTENTS

1. Introduction

The strategic objective of the QUALOSS project is to enhance the competitive position of the European software industry by providing methodologies and tools for improving their productivity and the quality of their software products.

To achieve this objective, QUALOSS notes that many organizations integrate Free *libre* Open Source Software (F/OSS) in their systems. However, there is currently no objective, comprehensive method for assessing F/OSS quality. Hence, QUALOSS plans on developing a tooled method for assessing the robustness and evolvability of F/OSS .

This first workpackage (WP1) performs requirements analysis through prototyping while the other scientific workpackages (WP2-4) improve on the functional prototype build in WP1. The first three tasks of WP1 (T1.1, T1.2 and T1.3) perform requirements analysis while the remaining three tasks (T1.4, T1.5, and T1.6) build the functional prototype and validate the approach.

Requirements analysis is approached in two directions, top-down and bottom-up. In particular, task 1.2 is top down, it starts from our two main quality criteria of interest, evolvability and robustness and investigates how to refine them based on narrower, more specific quality characteristics. On the other hand, Task 1.1 takes a bottom up angle. Independent from the activities of Task 1.2, Task 1.1 reviews the existing tools and techniques that could be of interest for measuring quality characteristics. The results of tasks 1.1 and 1.2 are merged in tasks 1.3, which identifies how to measure the quality characteristics highlighted in task 1.2 using the tools and techniques identified in tasks 1.1.

1.1 Motivation of Task 1.1

Assessing the evolvability and robustness of F/OSS requires automation as there is too much data available for a complete-manual treatment. Thanks to the use of tools, our assessment will be able to evaluate F/OSS on most of its data limiting the manual assessment of small data samples to very specific quality estimation.

Reviewing the quality of existing tools will also help identify where additional efforts are needed to implement new tools or even increase the reliability and robustness of these existing tools. As expected, there exist many tools that perform the same measures on the same kind of data, for example, many tools produce the count of lines of code for the same programming language. In turn, Task 1.1 must identify those worth using vs those to discard.

Before identifying tools, it is also important to identify the kind of data available in F/OSS repositories. In fact, it is only useful to inventory tools that can treat data available in F/OSS repositories or other sources of reliable information. Although the types of data available may vary among F/OSS projects, trends have emerge due to the use of well-known forges such as SourceForge that provide access to particular tool sets. In addition, some F/OSS communities decided to host their own forge such as Apache or AdaCore, a QUALOSS partner. Usually, these groups propose an even broader type of data, although some may not be available publicly. For example, AdaCore's test suite is

not public due to the presence of sensitive data in tests provided by AdaCore's customers.

The systematic inventory of available data and existing tools will help spot missing parts. In particular, we may discover that a tool does not exist for a particular dataset. If Task 1.2 and 1.3 further show that such a tool is useful for measuring a quality characteristic, the tool will need to be implemented.

1.2 Objectives of Task 1.1

This deliverable has the following objectives:

- Determine the type of data found in F/OSS repositories and other reliable sources of information related to F/OSS
- Determine the tools and techniques available to analyze F/OSS project data

1.3 Structure of the Deliverable

Section 2 contains a glossary of F/OSS related terminologies as used in QUALOSS.

Section 3 describes the different types of data found in repositories of F/OSS projects and other reliable sources of information on F/OSS projects. Furthermore, for each type of repository, we define its content, that is, the different datatypes contained in the repository.

We also give a rough estimate on how often F/OSS projects collect each datatype and also check it for validity.

Section 4 inventories analysis tools studied and selected for use in the QUALOSS platform. General information of each tool is presented along with the analyses computed.

Section 5 introduces advanced analysis and techniques for the data sources mentioned in Section 3 and Section 6 presents concluding remarks.

2. Glossary

This section presents a glossary of F/OSS related terminologies. We note that this glossary varies slightly from Section 7.0 of the DoW (Description of Work), which, among other things, defines F/OSS related terminology.

FlOSS:

Definition: FlOSS is short for Free/Open-Source Software.

Note: FlOSS refers to any software licensed under terms compliant with the Free Software Foundation definition of "free software", and the Open Source Initiative definition of "open source software", thus avoiding the controversy between those two terms. In fact, QUALOSS does not aim at studying the variation between Free Software and Open Source Software.

FlOSS license:

Definition: FlOSS license is a license listed by the Open Source Initiative (OSI url:www.osi.org) or a license that is legally compatible with a license listed by the OSI.

Note: QUALOSS does not study legal issues of license. However, if necessary, license types may somehow participate in the quality models defined by QUALOSS.

FlOSS project:

Definition: FlOSS project is a software project that is released under the terms of a FlOSS license.

Example: GNAT Pro, Zope, Linux, Eclipse, CVS are all FlOSS projects.

FlOSS forge:

Definition: A FlOSS forge presents a large container where FlOSS projects publish the product of their development (i.e. software) and also store the archives of communication and development interactions to produce that software.

Example: SourceForge (www.sourceforge.org) , GNU Software site (www.gnu.org), Eclipse site (www.eclipse.org) are three different FlOSS repositories.

Although most FlOSS forges have a web front where many different types of information are accessible, it is also possible for a FlOSS forge to be mostly private, releasing publicly only the data required by the FlOSS license used. The QUALOSS project is mostly interested in FlOSS forges and FlOSS projects that publicly release information of different types.

FlOSS project repository:

Definition: F*l*OSS project repository refers to a particular type of data repository that a F*l*OSS project may use to manage project data. Common types of repositories used by F*l*OSS projects are version control repository, bug tracking repository, mailing list archive, software release repository, a repository of Web pages, or of documentation. Repositories can be categorized according to the kind and structure of the data they contain, in particular structured, semi-structured, or unstructured.

Example: The Tomcat project (http://tomcat.apache.org/) uses several types of repositories: a SVN repository, a bug tracking repository, a mailing list archive repository, a software release repository, a repository for documentation, etc. When discussing the general case of any F*l*OSS project, we refer to one of these repositories as *a F*l*OSS project repository.*

3. F/OSS Repositories Data

Most F/OSS projects use the same tools to support the development and management of the community. All these development tools can be used as data sources for research purposes since they usually contain historical information about both the software product and processes. Moreover, other reliable sources of information may also show useful in measuring the quality or impact of a F/OSS project.

In summary, the main available data sources are the following:

- F/OSS project releases
- Version control system. Typically CVS or Subversion, although some other systems are used by some projects.
- Bug tracking systems. Typically Bugzilla, GNATS.
- Mailing list archives. Forum data.
- Other data sources internal to a F/OSS project, such as web (or wiki) pages, documentation accompanying a release, IRC logs, forums data.
- Other data sources external to a F/OSS project. For example, FLOSSMole or FLOSSMETRICS databases, the Common Vulnerability Database and the National Vulnerability Database, Amazon's book database, F/OSS News website such as FLOSSplanet, Slashdot, etc.

Many projects are stored in *forges*. The most well know forge site is SourceForge.net. This site offers hosting for web pages and files, CVS and Subversion (the two most used version control systems in the F/OSS world), trackers for bugs and other issues, forums, mailing lists, release management, categorization of projects, etc. There are some initiatives that are studying these sites for software research purposes. For instance, FLOSSMole[1] scans SourceForge every six months but it also collects information for other F/OSS forges such as FreshMeat, ObjectWeb, Free Software Foundation. The FLOSSMole database records many different types of data in a fixed database schema. This facilitates measuring and analyzing some software quality properties. Examples of information collected are project name, description, project donors, project license, project operating systems, programming languages and type of user interface. In addition FLOSSMole also provides few statistics about a F/OSS project such as its activity rank, number of downloads, number of opened or closed bugs. The complete schema of FLOSSMole is given at http://ossmole.sourceforge.net/datamodel/ossmole_schemaspy.

In the following subsections, we describe the data sources considered for our quality analysis.

3.1 F/OSS Project Releases

The release of a F/OSS project contains several types of information ranging from binary distribution for several platforms, the source code used to generate the binaries, documentation, tests to verify the proper installation and even a regression test suite.

The notion of release is important because it highlights the important dates for a F/OSS project. Moreover, the release numbers provide additional information; for example, whether a new release is just a minor or a major release. Usually, the notation uses the

[1] http://ossmole.sf.net

convention “Major.Minor”, for example, 1.4 indicates major release 1 and minor release 4. It is also frequent to have a third release number, which usually accounts for bug fixes and minor improvements, e.g. 1.4.528 where 528 indicates a unique number to which bug fixes and small improvements are associated.

One particular interesting set of data available in a release is the source code. There are two advantages of using source code releases over the code in the version control system for certain studies. First, the source code in the version control system may contain code not ready to be delivered; sometimes, this code is removed from the source code tree before the release is offered to the public. Second, the code is only released when the developers think (and even sometimes test) that the project is ready to be released. The release schedule can be studied to guess the periodicity in the growth curves of the project. However, we must also take into account that some projects have a fixed schedule for releases. For instance, they release every six months.

Beside source code, a product release often contain binaries, usually for the MS Windows platform. In some cases, binaries may be a work product of interest usable by analysis tools such as instrumentation tools for dynamic analysis mention in Section 4.

Usually, dynamic analysis requires the presence of tests so as to monitor system execution during test runs. Although complete test suites are rarely distributed as part of a release, they are sometimes available in the version control system of a F/OSS project. In turn, it may be possible to check out from the version control repository a test suite for the given release date. If not available that way, we may still be able to obtain a test suite for a particular release by asking a F/OSS developer community if they could provide it.

3.2 Version Control Systems

Most F/OSS projects use a version control system. Among F/OSS communities, the most used and well known systems are CVS and Subversion.

Version control systems keep track of changes to all registered files. They identify changes with a unique number assigned to each modified files. Along with a change, valuable information is recorded, mainly, the date of change, the full path where the change occurred, username of the developer who is committing the change, and a comment written by the developer explaining the change.

The two version control system of interest for QUALOSS are CVS and Subversion. CVS is older than Subversion. It has been used for many years by many F/OSS projects, and it is still in use. Subversion was born as an alternative to CVS. CVS has a big drawback: it works on a file basis. For example, if a developer has to change several files to make logical changes in the project, CVS will store this as a set of changes (one change per file). On the other hand, Subversion stores this logical change as one only change to the system. This logical changes are usually known as *modification requests* in the research community. In addition, Subversion can work with binary files.

In addition to these two version control systems, there exist other alternatives, among others, Bitkeeper, and GIT. However, they are not very popular in F/OSS projects, so QUALOSS does not currently plan to build tools to study them. Nonetheless, it is worth

noting that some very important F/OSS projects, such as Linux and MySQL, use these less conventional version control systems, in particular, MySQL uses Bitkeeper and Linux uses GIT.

From the version control systems, we can obtain two different types of data:

- Source code, at any date, written by any developer
- Meta information that consists of timestamp of commit, person who performed the commit, the log texts associated to a commit, the unique revision number (and eventual branch information) assigned by the version control system.

3.3 Bug Tracking Systems

Bug tracking systems are very popular among F/OSS projects. These systems are typically used to report and discuss defects (usually known as bugs). Some projects also use them for feature requests. Usually, bug tracking tools such as Bugzilla allow the user to classify her request as either a bug or a feature requests.

The most popular system is Bugzilla. This system has a web interface. Users can register to gain access to the bug tracking system. There are different profiles, the most common are: *users* who report and comment bugs, and *developers* who modify the status of reports and assign them to others developers.

Bug reports are available via web and also as XML. XML files are the best option to parse the bug reports. Usually, anyone can see the bug reports, although in some cases, users have to be registered.

Although fields may vary from one installation to another, common fields of Bugzilla bug reports are:

- Bug number and alias
- Product, component, hardware, operating system and version
- Reporter name and e-mail address
- Status, priority, resolution, severity
- Person to whom the bug is assigned
- Target milestone
- Summary and keywords
- Attachments (usually test cases)
- Additional comments by the reporter
- Additional comments by the owner or other users or developers

Furthermore, whenever a change in the report occurs, the date and time of the change is recorded.

The life cycle of a bug report is usually as follows:

- The bug is reported. A test case is attached. The status is set to UNCONFIRMED.
- The bug is tested by one of the developers. If the bug is confirmed, it is set to NEW; in addition,if it is assigned to one of the developers, and set to ASSIGNED. If it is not confirmed, the status is usually set to CLOSED.
- If the bug has been already reported, it is marked as DUPLICATED, including the number of the original bug, and it is closed.

- Other users or developers comment on the bug. If it was closed and other users observe the same bug, it can be reopen.
- A user can propose a patch, or the bug can be fixed by a developer. If a patch is proposed, it is tested by one of the developers, and eventually added to the source code tree. Then the bug is set to FIXED. The patch is attached to the report. If the code has been added to the version control system, the number of the revision is usually included in the report.

Other popular system is the tracker of SourceForge.net (SF.net). In the essential points, it is very similar to Bugzilla. The main difference is that the SF.net tracker is used not only for bugs, but also for petitions of new features and other inquiries coming from the users.

3.4 Mailing List Archives

Mailing lists are the main communication channel between developers and users. Every message sent to a mailing list stores information that allows to study the relationships and the communication channels between the communities of a F/OSS project. Among other, the following information is found in emails:

- Name of the poster
- E-mail address of the poster
- Date when the message was sent by the poster, and received in the mailing list server
- Subject of the message
- Mailing list address (where the message was sent to)
- Name and e-mail addresses of other recipients different of the list (for example, other people include in the CC field of the message)
- Unique identification tag for the message in the mailing list
- Identification tag of the original message if the message is a reply
- Content of the message, including attachments
- Name of the program used to write the message

All this information is stored in the header tags of the message (also the content and attachments). Although the content of the message itself could be a source of information, the information obtained in the headers is much more interesting. The obvious information that can be obtained is related to the activity (number of messages) and participation (number of people participating in the list) in the mailing list. This information is a good indicator of the activity and participation in the project, but there is much more interesting information to be obtained. If we cross correlate the information in every message with the information in other messages, we can reconstruct the networks of communication between the participants in the list. With these networks, information about how the communication flows in the project can be obtained. For example, we can identify members of the community who join subgroups within the community. A very important point to ensure the health of the project is the communication between users and developers. F/OSS projects are community-driven projects, and so the feedback obtained from the different communities is essential to ensure the survival and future of the project. Therefore, analysis of community

interaction should be further developed and used to study whether or not required communication is indeed occurring in a F/OSS project.

Not all mailing lists are stored in the same format. The most usual format is RFC822[2], which is also the main standard in the Internet to format e-mail messages. However, some projects strip out some headers, leaving just the name, address, subject and date sent. Sometimes even the e-mail address is obfuscated or masked. When the archives are stripped out of the headers, the information that can be obtained from them is much poorer and almost uninformative.

In general, fully automated email analysis is hard to achieve because information may be stripped or even because email titles may be changed in a reply chain

3.5 Other Data Sources Internal to a F/OSS Project

Other data created by the community of a F/OSS project includes:

- on-line documentation, (reference manual, user guide, installation guide, quick start guide, etc.)
- Web and Wiki pages
- IRC logs

These sources of information are completely unstructured and therefore much harder to process automatically. Moreover, the history of changes is not always available and only the most current version is kept, older version being overwritten or erased.

Nonetheless, it seems important to keep in mind the presence of these data sources as they may be useful for measuring certain quality characteristics. For example, the presence of a “good” website is likely to indicate an active and robust community. Furthermore, the structure and the information available on the website is likely to provide some insight into the structure of leadership of the community.

For example, does the community have a explicit procedure to follow when a patch is proposed? Is there a charter that specifes the roles of F/OSS project leaders? What kind of documentation is available on-line? is it the same as that distributed with the latest release?

3.6 Data Sources External to a F/OSS Project

In addition to the information provided by the communities connected to a F/OSS project, there exist several other useful sources of information potentially interesting to gauge the current robustness and potential future of a F/OSS project. Currently, three external kinds of data sources have been identified:

- F/OSS data provided by other projects such as FLOSSMETRICS or FLOSSMOLE
- Vulnerability Databases such as the Common Vulnerability Enumeration (CVE) and the National Vulnerability Database
- Book and publication databases such as Amazon, or bibliography databases
- News websites and archives such as slashdot or FLOSSPlanet

[2] http://en.wikipedia.org/wiki/RFC_822

The Common Vulnerability and Exposures (CVE) is a MITRE dictionary of many vulnerabilities experience and reported by users. In addition, the National Vulnerability Database (NVD) is an effort by the NIST that ranks CVE vulnerabilities on a scale from 1 to 10. The procedure for attributing a severity score to a vulnerability is well documented and objective.

Many F/OSS projects appear in CVE and their vulnerabilities have been assigned scores in NVD. Although CVE and NVD inventory vulnerabilities in all software products, not only F/OSS, they still provide important information to gauge robustness of F/OSS projects. In turn, they seem to be an important data source to consider for QUALOSS. For example, it may seem important for a particular minor release to improve over time so as to increase the confidence of users. Similarly, a new major release may suffer few more bugs but it must still be above a particular threshold so as to encourage migration to the newer version.

Amazon provide records on books published in relation to a particular technology, including F/OSS technologies. Many books have been published on a particular F/OSS technology or mention F/OSS technologies. F/OSS technologies that increase in popularity are likely to be advertised in books, even if a complete book is not dedicated to a single F/OSS technology, a chapter may be dedicated to it. Other publication databases such as The Collection of Computer Science Bibliographies (http://liinwww.ira.uka.de/bibliography/), citeseer (http://citeseer.ist.psu.edu/), or google scholar (http://scholar.google.com/) can also contribute, although it is yet undetermined how these scientific publications can help gauge evolvability or robustness of F/OSS projects.

Other sources of information available on renowned F/OSS news sites and archives may reveal a increased level of professionalism from certain F/OSS communities. In fact, as it is true for all products, advertisement in the press is an important factor to guarantee sustainability in the future, hence it may be a mark of evolvability. It therefore seems to be a potentially interesting source of information.

Additional data sources may be identified at later stages of the project. However, if complex processing is required to retrieve and treat the data to turn it into usable information, we require that such data sources be identified within the first year of the project.

4. Existing Analysis Tools

This section reviews existing tools for analyzing the different data sources mentioned in section 3. Code analysis tools dominate the field and are therefore presented in their own Subsection 4.1. In fact, Section 4.1 is itself partitioned into static and dynamic analysis tools for the different languages of interest in QUALOSS, namely, Ada, C, C++, Java and Python. Section 4.2 describes tools for analyzing other data sources, in particular, version control and mailing list archives.

4.1 Code Analysis Tools

The code analysis tools studied in Section 4.1 focus on a snapshot of the code of a project at a selected point in time; they do not compare multiple version of source code. It is however likely that the QUALOSS platform makes such a feature available in order to measure certain quality characteristics, if this should prove necessary. Tools such as GlueTheos described in Section 4.2 already address this problem in part.

This subsection only presents tools that were analyzed and that have met our expectations in terms of bringing new functionality, and reaching an acceptable level of reliability and efficiency. For example, four static analysis tools for Python were tested: PyMetrics, PyLint, PyChecker and Pythuis. However, only PyMetrics and PyLint made it to our list simply because PyChecker only computed a subset of the PyLint functionality and Pythuis proved to be quite unstable and lacking documentation.

Each tool analyzed in Section 4.1 is presented based on the template below. It contains five main sections: General Information, Input Information, Output Information, Technical and Operational Information, and Tests.

Tool name			
General Information			
Version:		Licenses:	
Authors:		Maturity:	
URL:		Dependencies:	
Description:			
Constraints:			
Input/Output Information			
Language Analyzed:			
Input Types		Input Formats	
Output Types		Output Formats	
Other I/O comments			
Technical Information			
Devel. Lang:		Documentation:	
Information Computed:			
Extensibility:			
Technical Constraints:			

Test Performed	
Reliability:	
Performance:	

4.1.1 Language Independent Analysis Tools

4.1.1.1 SLOCCount

SLOCCount				
General Information				
Version:	2.26	Licenses:	GNU General Public License (GPL)	
Authors:	David A. Wheeler	Maturity:	Stable	
URL:	http://www.dwheeler.com/sloccount/	Dependencies:	Flex (for compilation of sloccount)	
Description:	SLOCCount is a suite of programs for counting physical source lines of code (SLOC) in possibly large software systems. It can count physical SLOC for a wide number of languages. It can take a large set of files and automatically categorize their types using a number of different heuristics, and also comes with analysis tools.			
General Constraint:	Many projects are programmed in more than one language. Using SLOCCount, it is possible to quickly assess the percentage of source lines of code for each language used.			
Input/Output Information				
Language Analyzed:	Ada, Assembly, AWK, Bourne shell and relatives, C, C++, C#, COBOL, Expect, Fortran, Haskell, Java, lex, LISP, Scheme, makefile, ML, Modula3, Objective-C, Pascal, Perl, PHP, Python, Ruby, sed, sql, TCL, yacc			
Input Types	Source Code	Input Formats	Directory containing source code files	
Output Types	Text	Output Formats	dumps reports on stdout	
Other I/O comments				
Technical Information				
Devel. Lang:	mainly: Perl, C, sh	Documentation:	Installation, reference manual and user documentation explaining the application of COCOMO (available at http://www.dwheeler.com/sloccount/sloccount.html)	
Information Computed:	SLOCCount computes the physical source lines of code for each language encountered in a given directory (and its subdirectories) It also computes the estimated effort based on a COCOMO estimate. It			

	is possible to customize COCOMO parameters on the command line.
Extensibility:	Done by modifying SLOCCount source code
Technical Constraints:	The application of the COCOMO model is fairly basic as it directly infers effort from the line counts (independent of programming languages used). Nonetheless due to its ability to read many languages and treat input in a consistent manner, SLOCCount may provide valuable information. If nothing else, lines count for each programming languages used in a project.
Test Performed	
Reliability:	Currently no bugs are found. Count Physical Lines of Source code fairly accurately
Performance:	SLOCCount was tested on distributions of Inkscape , Azeurus, SISSy, CCCC. It computed its results in just a few seconds even for Inkscape for which wc counts more than 400K lines.

4.1.2 Ada Static Analysis Tools

4.1.2.1 GNATmetric

GNATmetric				
General Information				
Version:	6.0.1	Licenses:	GNU General Public License (GPL)	
Authors:	AdaCore	Maturity:	Mature	
URL:	http://www.adacore.com	Dependencies:	GNAT Pro compiler	
Description:	The GNATmetric tool analyzes source code to calculate a set of commonly used industry metrics that allow developers to estimate the size and better understand the structure of the source code. This information also facilitates satisfying the requirements of certain software development frameworks.			
General Constraint:	The input Ada sources must be compilable.			
Input/Output Information				
Language Analyzed:	Ada			
Input Types	Source Code	Input Formats	List of files or project description file	
Output Types	Files	Output Formats	Textual and XML formats	
Other I/O comments	NONE			
Technical Information				
Devel. Lang:	Ada	Documentation:	Documentation is available as part of the GNAT Pro User's Guide (available at http://www.adacore.com/category/developers-center/reference-library/documentation).	
Information Computed:	For any (legal) source file, and for each of its eligible local program units, GNATmetric computes the following metrics: • the total number of lines • the total number of code lines (i.e., non-blank lines that are not comments) • the number of comment lines • the number of code lines containing end-of-line comments • the number of empty lines and lines containing only space characters and/or format effectors (blank lines) If GNATmetric is invoked on more than one source file, it sums the values of the line metrics for all the files being processed and then			

generates the cumulative results.

GNATmetric computes various syntactic metrics for the outermost unit and for each eligible local unit:

- LSLOC ("Logical Source Lines Of Code"). The total number of declarations and the total number of statements.
- Maximal static nesting level of inner program units. According to Ada Reference Manual, 10.1(1), "A program unit is either a package, a task unit, a protected unit, a protected entry, a generic unit, or an explicitly declared subprogram other than an enumeration literal."
- Maximal nesting level of composite syntactic constructs. This corresponds to the notion of the maximum nesting level in the GNAT built-in style checks.

For the outermost unit in the file, GNATmetric additionally computes the following metrics:

- Public subprograms. This metric is computed for package specifications. It is the number of subprograms and generic subprograms declared in the visible part (including in nested packages, protected objects, and protected types).
- All subprograms. This metric is computed for bodies and subunits. The metric is equal to a total number of subprogram bodies in the compilation unit. Neither generic instantiations nor renamings-as-a-body nor body stubs are counted. Any subprogram body is counted, independently of its nesting level and enclosing constructs. Generic bodies and bodies of protected subprograms are counted in the same way as "usual" subprogram bodies.
- Public types. This metric is computed for package specifications and generic package declarations. It is the total number of types that can be referenced from outside this compilation unit, plus the number of types from all the visible parts of all the visible generic packages. Generic formal types are not counted. Only types, not subtypes, are included.
  Along with the total number of public types, the following types are counted and reported separately:
  - Abstract types
  - Root tagged types (abstract, non-abstract, private, non-private). Type extensions are not counted.
  - Private types (including private extensions)
  - Task types
  - Protected types
- All types. This metric is computed for any compilation unit. It is equal to the total number of the declarations of different types given in the compilation unit. The private and the corresponding full type declaration are counted as one type declaration. Incomplete type declarations and generic formal types are not counted. No distinction is made among different kinds of types

	(abstract, private etc.); the total number of types is computed and reported. For a program unit that is an executable body (a subprogram body (including generic bodies), task body, entry body or a package body containing its own statement sequence) GNATmetric computes the following complexity metrics: • McCabe cyclomatic complexity • McCabe essential complexity • maximal loop nesting level The McCabe complexity metrics are defined in http://www.mccabe.com/pdf/nist235r.pdf According to McCabe, both control statements and short-circuit control forms should be taken into account when computing cyclomatic complexity. For each body, we compute three metric values: • the complexity introduced by control statements only, without taking into account short-circuit forms, • the complexity introduced by short-circuit control forms only, and • the total cyclomatic complexity, which is the sum of these two values. When computing cyclomatic and essential complexity, GNATmetric skips the code in the exception handlers and in all the nested program units.
Extensibility:	Done by modifying GNATmetric source code
Technical Constraints:	It is possible to generate project-wise metrics by means of defining the project and the metrics associated to the project in a project file.
Test Performed	
Reliability:	Currently no bugs are found
Performance:	It takes less than 1 minute to compute all line syntax, and complexity metrics for a project with more than 200000 lines of code and nearly 300 units, generating a 5 MB size report. The test has been driven with a Pentium M 2,1 GHz.

4.1.2.2 GNATstack

GNATstack			
General Information			
Version:	6.0.1	Licenses:	GNU General Public License (GPL)
Authors:	AdaCore	Maturity:	Mature
URL:	http://www.adacore.com	Dependencies:	GNAT Pro compiler
Description:	The GNATstack tool statically computes the maximum stack space required by every stack entry point (including tasks) in an application. The computed bounds can be used to ensure that sufficient space is reserved, thus guaranteeing safe execution with respect to stack usage.		
Constraints:	The input Ada project must be compiled with specific options.		
Input/Output Information			
Language Analyzed:	Ada		
Input Types	Object Code	Input Formats	List of object files that make up the project
Output Types	Files or GUI	Output Formats	Textual and VCG (Visualization of Compiler Graphs) formats.
Other I/O comments			
Technical Information			
Devel. Lang:	Ada	Documentation:	Documentation is available as part of the GNATstack User's Guide.
Information Computed:	GNATstack will report the accumulated stack usage information for every entry point. In addition, it will provide the information about the call chain that make up the worst-case paths. Additionally, GNATstack can generate a list of subprograms requiring the biggest local stack usage. GNATstack can also generate a file (in VCG format) containing the annotated call graph. GNATstack can also indicate the list of subprograms that make indirect calls, all subprograms that are reachable from any entry point for which we do not have any stack or call graph information, all subprograms that are reachable from any entry point with unbounded stack requirements, and all the cycles in the call graph. GNATstack can also perform four additional types of analysis: ● Indirect (including dispatching) calls. The tool indicates the		

	number of indirect calls made from any subprogram. • External calls. The tool displays all subprograms that are reachable from any entry point for which we do not have any stack or call graph information. • Unbounded frames. The tool displays all subprograms that are reachable from any entry point with an unbounded stack requirements. The required stack size depends on the arguments passed to the subprogram. • Cycles. The tool can detect all cycles in the call graph. These cycles represent potential recursion and hence potentially unbounded stack consumption.
Extensibility:	By modifying the code of GNATstack
Technical Constraints:	
Test Performed	
Reliability:	Currently no bugs are found
Performance:	It takes less than 18 seconds to compute all call paths together with the accumulated stack usage, list of subprograms that make indirect calls, all subprograms that are reachable from any entry point for which we do not have any stack or call graph information, all subprograms that are reachable from any entry point with unbounded stack requirements, and all the cycles in the call graph, for a program containing more than 17000 subprograms and more than 108000 calls. The test has been driven with a Pentium M 2,1 GHz.

4.1.2.3 GNATcheck

GNATcheck			
General Information			
Version:	6.0.1	Licenses:	GNU General Public License (GPL)
Authors:	AdaCore	Maturity:	mature
URL:	http://www.adacore.com	Dependencies:	GNAT Pro complier
Description:	The GNATcheck tool is an ASIS-based utility that checks properties of Ada source files according to a given set of semantic rules.		
Constraints:	The input Ada sources must be compilable.		
Input/Output Information			
Language Analyzed:	Ada		
Input Types	Source code	Input Formats	List of source files in a project.
Output Types	Files	Output Formats	Text
Other I/O comments	NONE		
Technical Information			
Devel. Lang:	Ada	Documentation	Documentation is available as part of the GNAT Pro User's Guide (available at http://www.adacore.com/category/developers-center/reference-library/documentation).
Information Computed:	Rules implemented in GNATcheck are subdivided into local rules and global rules. A local rule is a rule that is formulated for a well-localized fragment of a program text and that can be checked by analyzing only this fragment (the analysis may use the semantic information related to the components of this fragment). A global rule requires analysis of some global properties of the whole program (mostly related to the program call graph). GNATcheck can detect the use of: abstract types, anonymous subtypes, block statements, relation operations on boolean types, ceiling priority consistency, controlled types, block statements with local declarations, default expressions for subprogram parameters, derived type declarations that does not have a record extension part, calls to the predefined equality operations for floating point types, declarations of record types with discriminants, exit statements containing a loop		

	name that is not the name of the immediately enclosing loop, goto statements, library level subprograms (including subprogram instantiations), local packages in package and generic package specs, protected entries that can be called more then one task, protected objects with more than one entry, function declarations with operator symbols as a defining designators, return statement in procedure bodies, non-qualified aggregates, recursion, the use of some names, functions with side effect, expanded loop names in exit statements, slices, SPARK restrictions, functions returning unconstrained arrays, discrete ranges that are a part of index constraint, constrained array definition, or for-loop parameter specification, and that have both bounds of the universal type integer type, unused subprograms, use package clauses, and volatile object that does not have an address clause, anonymous array types in object declarations, actual parameters for a formal that has a default initialization if this parameter is in positional association, bad discrete ranges, non-named block statements, level of nesting of control structures, use of a range of enumeration literals as a choice in a case statement, exceptions that are raised and handled in the same subprogram body, procedures that can be rewritten as functions, generic units in supbrograms, implicit IN mode in formal parameter specification, exit statements that do not contain the name of the loop being exited, checks if the BEGIN keyword in package body is marked by the trailing comment containing the package name, numeric literals, use of OTHERS choice in extension record and array aggregates, OTHERS choice in a case statement, OTHERS choice in a exception handler, subprogram and entry declarations where the formal parameters are not properly ordered, positional parameter or element association, use of specified pragmas, explicit use of any name of a predefined numeric type or subtype defined in package Standard, raising predefined exceptions, generic instantiations that are done in library package specifications and in subprogram bodies, visibility of exceptions, and use of non-short-circuit boolean operators.
Extensibility:	By modifying the code of GNATcheck
Technical Constraints:	A number of rules are predefined in GNATcheck and are described later in this chapter. New rules can be added by modifying the GNATcheck code and rebuilding the tool. In order to add a simple rule making some local checks, a small amount of straightforward ASIS-based programming is usually needed.
Test Performed	
Reliability:	Currently no bugs are found
Performance:	The use of ASIS semantics queries makes the application to consume a lot of resources (processor, disk, and memory).

4.1.2.4 AdaControl

AdaControl			
General Information			
Version:	1.6r8	Licenses:	GNU General Public License (GPL)
Authors:	Adalog	Maturity:	Mature
URL:	http://www.adalog.fr/adacontrol2.htm	Dependencies:	AdaControl is distributed only as source, so it needs an Ada compiler and ASIS (Ada Semantic Interface Specification) library.
Description:	AdaControl is an Ada rules controller. It is used to control that Ada software meets the requirements of a number of parameterizable rules. It is not intended to supplement checks made by the compiler, but rather to search for particular violations of good-practice rules, or to check that some rules are obeyed project-wide.		
Constraints:	The input Ada sources must be compilable.		
Input/Output Information			
Language Analyzed:	Ada		
Input Types	Source code	Input Formats	List of source files in a project.
Output Types	File	Output Formats	Text
Other I/O comments			
Technical Information			
Devel. Lang:	Ada	Documentation:	Documentation is available as part of the AdaControl User Guide (available at http://www.adalog.fr/compo/adacontrol_ug.html).
Information Computed:	AdaControl can check: functions without a return or a raise statement, dynamic allocations, properties of array types and array objects declarations, expressions used in barriers of protected entries, sizings in case statements, the occurrence in the source file of control characters, the usage of certain Ada declarations, subprogram calls or generic instantiations that use (or conversely do not use) the default value for the indicated parameter, global variables that are accessed outside of dedicated callable entities, uses of indicated entities, exception handlers that contain references to one or several Ada entities specified as parameters, subprograms, tasks, or all declarations that can		

	propagate exceptions, usage of certain forms of expressions, access to global variables from several entities, header comments, usage of if statements that could be replaced by case statements, instantiations of generics, parameters passed in positional notation, local declarations that hide outer declarations, instantiations in local scopes, number of consecutive blank lines, depth of subprograms (or entry) calls, length of source lines, level of nesting of declarative constructs, number of parameters in subprograms, nesting of compound statements, statements that can be moved outside accept statements, naming convention, variables and/or out parameters that are not safely initialized, non-static expressions, elaboration calls, aliased use of variables in subprogram calls, semantic dependencies, potentially blocking operations from within protected operations, use of specific pragmas, declarations that could be moved to some inner scope, usage of representation clause, functions returning complex objects, subprogram calls or generic instantiations where different actual parameters call functions known to have side effects, exception handlers that can cause exceptions to silently disappear, expressions that can be simplified, comments that match one of the given patterns, usage of certain Ada statements, Ada coding style, tasks that can terminate, missing units, unnecessary use clauses, calls to operations that are normally paired, use of Unchecked_Conversion, use of certain entities, and usage of with and use clauses.
Extensibility:	
Technical Constraints:	
Test Performed	
Reliability:	Currently no bugs are found
Performance:	The use of ASIS semantics queries makes the application consume a lot of resources (processor, disk, and memory).

4.1.3 C/C++ Static Analysis Tools

4.1.3.1 SISSy (-cpp)

SISSy (-cpp)			
General Information			
Version:	0.40	Licenses:	LGPL
Authors:	Adrian Trifu, Mircea Trifu, Olaf Seng, and Peter Sulzman	Maturity:	Stable
URL:	http://sissy.fzi.de/SISSy/CMS/index_html	Dependencies:	ANTLR, JArgs, jTDS, PostgreSQL-jdbc, Recoder (The required libraries are distributed with SISSy)
Description:	(taken from the documentation of the SISSy distribution) The tool for structural investigation of software systems (SISSy) is an open-source platform for the automated detection of structural flaws. It was designed to be integrated in the build process in order to regularly provide reports on the internal quality of the developed system. If in the course of development, problems arise in the structure, they are immediately identified and reported, giving developers the opportunity to fix them before they get unmanageable.		
Constraints:	SISSy works out of the box on MS Windows systems but small changes to the code had to be performed so it ran on Linux.		
Input/Output Information			
Language Analyzed:	C and C++		
Input Types	Source	Input Formats	root directory containing source files
Output Types	Text, Database (postgres or MS SQL)	Output Formats	database records for divers source code elements, problematic partner in text files, clone analysis in text files
Other I/O comments	SISSY also analyzes source code in Java (up to 1.4) and Delphi. Clone Analysis is run separately from the other analyses SISSy performs clone analysis as well as a range of pattern analyzes looking for bad programming patterns in the code. Bad patterns analysis can be requested via the command line option -queries or the actual SQL queries may also be run on the exported database records.		
Technical Information			
Devel. Lang:	Java	Documentation:	Reference manual

Information Computed:	SISSy performs Clone analysis and Pattern Analysis for 52 poor programming styles. Additional queries to compute traditional metrics could easily be implemented in SQL queries and run against the database of code elements created by SISSy.
Extensibility:	Additional analysis can easily be implemented by accessing SISSy's result stored in a database. It is also possible to modify SISSy's source code so it performs additional analyses
Technical Constraints:	
Test Performed	
Reliability:	SISSy was tested on three C++ systems, namely, wgrep3, inkscape 0.45, and 7Zip. wgrep3 is a small application consisting of 3 files, 1 .h and 2 .cpp with just under 1600 lines (as reported by wc). Inkscape is a large application with 1448 files (.cpp, .hpp, .c and .h). wc reports 432.819 lines in these files. 7Zip is a MS C++ application whose wc counts around 128KLOC in .cpp, .c and .h files. SISSy's parsing reliability outperforms that of CCCC. Furthermore, SISSy allows for more control as include.txt enable specifying a list of header files and definitions.prop allows specifying macro definition for the preprocessing of source code to account for during parsing. SISSy is based on a CDT parser. Our test shows that it handles C++ for gcc and for MS C++.
Performance:	SISSy took around 3 hours to analyze inkscape and insert all the data in the database when provided a long list of header files in include.txt. This analysis was performed on a laptop with 1MB of RAM and a CPU clock at 1.8 Ghz. SISSy took just under 15 minutes to analyze 7Zip without specifying an include.txt file pointing to headers from libraries.

4.1.3.2 CCCC

CCCC			
General Information			
Version:	3.1.4	Licenses:	GNU General Public License (GPL)
Authors:	Tim Littlefair	Maturity:	Stable but Inactive
URL:	http://cccc.sourceforge.net	Dependencies:	PCCTS (Purdue Compiler Construction Toolset (Terence Parr). PCCTS is an old version of the renowned parser generator ANTLR. However, CCCC distribution includes the needed PCCTS code.
Description:	CCCC (C and C++ Code Counter) analyzes and reports measurements on source code written in C, C++, and Java. Metrics supported include lines of code, McCabe's complexity and metrics proposed by Chidamber&Kemerer and Henry&Kafura.		
Constraints:	Although the MS Windows distribution comes in binaries, CCCC must be recompiled on POSIX. In this test, CCCC was compiled with gcc 4.1.2. The compilation did not produce problems.		
Input/Output Information			
Language Analyzed:	C, C++		
Input Types	Source code in C and C++	Input Formats	list of files (specified on the command line)
Output Types	files in a directory (.cccc by default)	Output Formats	xml and html
Other I/O comments	Other Input-related information: CCCC can also handle Java (up to 1.4) but we did not test that functionality. CCCC understands wild card (*) such as *.cpp CCCC also generate a internal cccc.db files to store results across runs		
Technical Information			
Devel. Lang:	C++	Documentation:	“cccc –help” provides the generic help message that explains how to specify information on the command line. cccc accepts wildcards on the

			command line however, every directory containing code must be listed with wildcards in order to read in all the input files. readme.txt that comes with the CCCC distribution only contains general information about the tools and does not provide any useful help.
Information Computed:	CCCC computes the following metrics: Number of Modules, Lines of Code, Lines of Comments, an approximation of McCabe's Cyclomatic Complexity, Information Flow (coupling between modules), Weighted Method per Class, Depth of Inheritance, Number of Children, Coupling between Objects, Fan In and Fan out.		
Extensibility:	it is possible for another tool to interact with cccc based on the output xml files or even reading .db and .opt files		
Technical Constraints:	CCCC reports the number of lines rejected by the compiler The results computed as McCabe Cyclomatic Complexity does not follow the traditional definition, in particular, CCCC increase McCabe CC by 1 for each return statement, unlike the original definition. Furthermore, as documented in cccc_tok.cpp, the computation for switch statements is correct only if every case statement is terminated by a beak statement.		
Test Performed			
Reliability:	CCCC was tested on two C++ systems, namely, wgrep3 and inkscape 0.45. wgrep3 is a small application consisting of 3 files, 1 .h and 2 .cpp with just under 1600 lines (as reported by wc). Inkscape is a large application with 1448 files (.cpp, .hpp, .c and .h). wc reports 432.819 lines in these files. Even when it cannot treat the whole input, CCCC still produces an output for the input parts analyzed Beside reports on metrics, it also mentions the number of lines rejected by the parser, for our two examples, around 10% of the code was rejected by the parser. For the 90% rate of processed input, the measurements are accurate on the samples investigated.		
Performance:	CCCC analyzed wgrep in just a few seconds while the analysis of inkscape 0.45 took around 5 minutes on a laptop with 1MB of RAM and a CPU clock at 1.8 Ghz.		

4.1.4 Java Static Analysis Tools

4.1.4.1 SQUAL

SQUAL			
General Information			
Version:	0.4	Licenses:	GPL and LGPL
Authors:	CETIC	Maturity:	Beta but quite stable
URL:	http://www.cetic.be	Dependencies:	ANTLR
Description:	SQUAL is a lightweight and easy pluggable workflow engine written in Python. Its current purpose is a source code analysis tool for Java and C#.		
Constraints:	The analysis is memory consuming		
Input/Output Information			
Language Analyzed:	Java 1.1 to 1.6		
Input Types	Source code	Input Formats	Directory structure, zip file, cvs or subversion repository
Output Types	File, Database	Output Formats	Csv, MySQL, XML, Text
Other I/O comments			
Technical Information			
Devel. Lang:	Python	Documentation:	Installation and User manual
Information Computed:	More than 100 metrics, mainly: – NumberOfClasses, NumberOfInterfaces of Packages – LinesOfCode, LinesOfComments, CommentsPercentage of Methods, Classes, Packages – AfferentCoupling, EfferentCoupling, CyclomaticComplexity, NumberOfAttributes, NumberOfMethods, NumberOfInnerClasses of Classes – EfferentCoupling, CyclomaticComplexity, Parameters, LocalVariables of Methods – MaximumDepthOfInheritance, NomberOfSubclasses of Classes		
Extensibility:	The workflow engine is highly configurable. New metrics can be defined by the user in Python.		
Technical Constraints:	Memory consuming, mainly due to the use of the psyco optimizer (psyco does not work on 64bit architectures)		

Test Performed	
Reliability:	Beta version, but ran on large projects reaching 1 million lines of code.
Performance:	Limited by the fact that python is an interpreted language, however thanks to psyco performance is quite reasonable. Squal analyzes 200KLOC in about 15 minutes.

4.1.4.2 CheckStyle

Checkstyle			
General Information			
Version:	4.2	Licenses:	GNU Library or Lesser General Public License (LGPL)
Authors:	Oliver Burn	Maturity:	Mature
URL:	http://checkstyle.sourceforge.net	Dependencies:	None (beside JVM install)
Description:	Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code to spare humans this boring (but important) task. This makes it ideal for projects that want to enforce a coding standard. Checkstyle is highly configurable and can be made to support almost any coding standard. An example configuration file is supplied supporting the Sun Code Conventions. Other sample configuration files are supplied for other well known conventions.		
Constraints:			
Input/Output Information			
Language Analyzed:	Java (including Java 5)		
Input Types	Source Code	Input Formats	The command line accepts access paths to a file or to a directory (using the -r option)
Output Types	Files	Output Formats	Text and XML
Other I/O comments	Checkstyle can also be invoke from Ant scripts Output files can become quite large, for example, the size of the XML" file generated by Checkstyle when analyzing the source code of Azureus (500 KLOC) is about 100MBytes.		
Technical Information			
Devel. Lang:	Java	Documentation:	User manual available at the URL above

Information Computed:	The checks made by checkstyle are dealing with Javadoc Comments, Naming Conventions, Headers, Imports, Size Violations, Whitespace, Modifiers, Block Checks, Coding, Class Design, Duplicate Code, Metrics, Miscellaneous, J2EE Checks.
Extensibility:	It is possible to write checks and configuration files. The existing checks can also be modified.
Technical Constraints:	Checkstyle only performs pattern matching, it does not perform type resolution hence rules cannot check for type information.
Test Performed	
Reliability:	Currently no bugs are found.
Performance:	For example the Azureus project (about 500 kloc) takes 5 minutes to produce 100MByte size output. The test has been conducted on a Pentium 4 1.5GHz.

4.1.4.3 JDepend

JDepend			
General Information			
Version:	2.9.1	Licenses:	BSD License
Authors:	Mike Clark	Maturity:	Mature
URL:	http://clarkware.com/software/JDepend.html	Dependencies:	JVM
Description:	JDepend traverses Java class file directories and generates design quality metrics for each Java package. JDepend allows you to automatically measure the quality of a design in terms of its extensibility, reusability, and maintainability to manage package dependencies effectively.		
Constraints:	JDepend need to be build with ANT. (the “build.xml” is provided)		
Input/Output Information			
Language Analyzed:	Java		
Input Types	java bytecode (.class or .jar)	Input Formats	Path to a directory containing java source files or java “.class” files
Output Types	Files, GUI	Output Formats	Text, XML, and GUI
Other I/O comments	JDepend analyzes java classes or sources. It sometimes encounters problems with the sources (documentation is missing to explain the limitation).		
Technical Information			
Devel. Lang:	Java	Documentation:	User documentation is available on the web site but information is missing concerning the analysis of Java sources.
Information Computed:	JDepend traverses Java class file directories and generates design quality metrics for each Java package, including: Number of Classes and Interfaces, Afferent Couplings, Efferent Couplings, Abstractness, Instability, Distance from the Main Sequence (This metric is an indicator of the package's balance between abstractness and stability), Package Dependency Cycles.		
Extensibility:	An analysis can be customized through the properties file so as to apply filters on the package to consider. JDepend task may be invokes in Ant script		
Technical Constraints:	For unknown reasons, when analyzing source code, the analysis works in some cases and not in others.		

Test Performed	
Reliability:	Currently no bugs are found
Performance:	Analyzing a 1MLOC project took 7 minutes and produced 4Mbyte of output. The test was performed on a Pentium 4 1,5ghz.

4.1.5 Python Static Analysis Tools

4.1.5.1 PyMetrics

PyMetrics			
General Information			
Version:	0.7.6	Licenses:	GNU General Public License (GPL)
Authors:	Reginald B. Charney (Project Administrator)	Maturity:	Stable but inactive
URL:	http://sourceforge.net/projects/pymetrics/	Dependencies:	Python interpreter
Description:	PyMetrics produces metrics for Python programs. Metrics include McCabe's Cyclomatic Complexity metric, LoC, %Comments, etc. Users can also define their own metrics using data from PyMetrics. PyMetrics outputs SQL command files and CSV output.		
Constraints:			
Input/Output Information			
Language Analyzed:	Python		
Input Types	Source Code	Input Formats	Lists of Files
Output Types	Text Files	Output Formats	text with csv or SQL commands
Other I/O comments			
Technical Information			
Devel. Lang:	Python	Documentation:	-h option displays reference manual
Information Computed:	2 Types of information are provided: The information about each file parsed • Basic Metrics (blockCount, numClasses, numComment, numFunction, ... comment associated with a methods, a class) • McCabe Complexity Metric • Sources lines of code (SLOC) from Cocomo 2's The information about each token parsed • each token is identified as (operator, new line, name, identifiant,...) and basic metrics are computed (blocknum, blockDepth, fctDepth,...)		
Extensibility:	By modifying the source code		
Technical Constraints:			

Test Performed	
Reliability:	Currently no bugs are found
Performance:	3500 lines of code took 10 seconds to produce its output on a Pentium 4 1,5ghz

4.1.5.2 PyLint

PyLint			
General Information			
Version:	0.12.2	Licenses:	GPL
Authors:	Alexandre Fayolle	Maturity:	Stable and Active
URL:	http://www.logilab.org	Dependencies:	logilab-astng and logilab-common packages. They should be compatible with python version greater than 2.2.0 (python 2.2 users will have to install the optik package).
Description:	Pylint is a tool that checks for errors in python code, tries to enforce a coding standard and looks for code smells. This is similar but nevertheless different from what pychecker_ provides, especially since pychecker explicitly does not bother with coding style. Pylint will display a number of errors and warnings as it analyzes the code, as well as some statistics about the number of warnings and errors found in different files. If you run pylint twice, it will display the statistics from the previous run together with the ones from the current run, so that you can see if the code has improved or not.		
Constraints:			
Input/Output Information			
Language Analyzed:	Python		
Input Types	Source Code	Input Formats	List of files
Output Types	text or html	Output Formats	stdout
Other I/O comments			
Technical Information			
Devel. Lang:	Python	Documentation:	Installation and User guide included in the PyLint distribution
Information Computed:	PyLint generates two output sections, a source code section and a report section. The source code section identifies problems in Python source code. For each problem it gives the type, the line, the object and the message. The message type can be: [R]efactor for a "good practice" metric violation, [C]onvention for coding standard violation, [W]arning for stylistic problems, or minor programming issues, [E]rror for important programming issues (i.e. most probably bug) or [F]atal for errors which		

	prevented further processing. Reports Section list the following information: • Duplication code + difference between current and previous version • Raw metrics: (code, docstring, comment, empty) + difference between current and previous version • External dependencies • Statistics by type (module, class, method, function) + difference between current and previous version • Messages by category (convention, refactor, warning, error) + difference between current and previous version • % errors / warnings by module • Global evaluation (rate of the program)
Extensibility:	By modifying the code, it is possible to add new checkers
Technical Constraints:	
Test Performed	
Reliability:	Currently no bugs are found
Performance:	PyLInt took 110 seconds to produce the output 3500 lines of Python code on a Pentium 4 1,5ghz.

4.1.5.3 The Metrics (for Python)

The Metrics (for Python)			
General Information			
Version:	1.0	Licenses:	GPL
Authors:	annelih, Anders Storsveen, Morten Svendsen, Rune E. J., Thomas Oesterlie	Maturity:	Stable but Inactive
URL:	http://sourceforge.net/projects/pythonmetric	Dependencies:	Python
Description:	This program will calculate and output metrics on code written in Python. • Metrics with different levels of granularity : class, function, module • Reports can be generated in text or XML-files. • A plug-in system lets new metrics be added to the program. • Includes Cyclomatic complexity, Lack of Cohesion from Chidamber-Kemerer and from Henderson-Sellers		
Constraints:			
Input/Output Information			
Language Analyzed:	Python		
Input Types	Source Code	Input Formats	File or Directory
Output Types	text and xml	Output Formats	file in directory (the directory by default is named defaultreport)
Other I/O comments	It only analyzes a single file or a single directory		
Technical Information			
Devel. Lang:	Python	Documentation:	python Main.py serves as reference manual
Information Computed:	Number of Classes, Number of functions per Class, funtion Cyclomatic complexity, Lack of Cohesion of Classes based on definitions from Chidamber-Kemerer and from Henderson-Sellers. The Metrics and PyMetrics have a subset of common metrics. However The Metrics computes lack of cohesion, which is not provided by PyMetrics		
Extensibility:	Metrics are plugged-in so new metrics can be added to The Metrics by coding them in Python and including the Python file in the Plugin directory.		

	The output reports are available in XML hence output can easily be read for further processing
Technical Constraints:	Each metrics computed as code in the Plugin directory
Test Performed	
Reliability:	The Metrics run on all input provided. The Metrics compute metrics as documented in the text file in the Plugin directory.
Performance:	The Metrics was used to analyze a few modules in the Zope Dependencies directory. For each module, it took less than 2 seconds for the analysis although modules were fairly small less than 2KLOC. However, Python code always tend to be small and compact compare to other programming languages.

4.1.6 Dynamic Analysis

We present fewer dynamic analysis tools hence there is less of a need to separate them on a language basis. Currently, we are still actively searching for tools that perform dynamic analysis on Python. There exist a few tools (PyCover) or python modules (coverage.py)

4.1.6.1 GCOV

GCOV			
General Information			
Version:	4.1.2	Licenses:	GNU General Public License (GPL)
Authors:	Free Software Foundation	Maturity:	Mature
URL:	http://gcc.gnu.org	Dependencies:	GCOV must be used in conjunction with the GCC backend.
Description:	GCOV is a test coverage program that analyzes the number of times each line of a program is executed during a run. It generates binary-level code instrumentation and results are given at the source level (annotated source code).		
Constraints:	The program to analyze must be compiled with specific options.		
Input/Output Information			
Language Analyzed:	Ada, C, C++, Objective-C, Fortran.		
Input Types	Object Code	Input Formats	Executable files
Output Types	Annotated source files containing how often each file is executed.	Output Formats	Text
Other I/O comments			
Technical Information			
Devel. Lang:	C	Documentation:	Reference manual is available as part of the GCC Manual available at http://gcc.gnu.org/onlinedocs/gcc.
Information Computed:	GCOV produces annotated source files with the following basic information: • how often each line of code executes • which lines of code are actually executed • how much computing time each section of code uses (needs		

	gprof) • branch frequencies Results can be cumulative against several executions of the same executable, or against execution of several executables using the same objects (test suites).
Extensibility:	
Technical Constraints:	
Test Performed	
Reliability:	Currently no bugs are found
Performance:	The execution time of the instrumented program is slightly increased (around 7%). The postprocessing of the generated information (in order to produce the final report) is very fast.

4.1.6.2 GNATmem

GNATmem			
General Information			
Version:	6.0.1	Licenses:	GNU General Public License (GPL)
Authors:	AdaCore	Maturity:	Mature
URL:	http://www.adacore.com	Dependencies:	GNAT Pro compiler
Description:	The GNATmem utility monitors dynamic allocation and deallocation activity in a program, and displays information about incorrect deallocations and possible sources of memory leaks.		
Constraints:	Available only on AIX, HP-UX, GNU/Linux, Solaris and Windows NT/2000/XP (x86).		
Input/Output Information			
Language Analyzed:	Ada		
Input Types	Executable Code	Input Formats	Files
Output Types	Text	Output Formats	report on Stdout
Other I/O comments			
Technical Information			
Devel. Lang:	Ada	Documentation:	Reference Manual and User documentation is available as part of the GNAT Pro User's Guide available at http://www.adacore.com/category/developers-center/reference-library/documentation).
Information Computed:	The GNATmem utility monitors dynamic allocation and deallocation activity in a program, and displays information about incorrect deallocations and possible sources of memory leaks. It provides three types of information: • General information concerning memory management, such as the total number of allocations and deallocations, the amount of allocated memory and the high water mark, i.e. the largest amount of allocated memory in the course of program execution. • Backtraces for all incorrect deallocations, that is to say deallocations which do not correspond to a valid allocation. • Information on each allocation that is potentially the origin of a memory leak.		

Extensibility:	
Technical Constraints:	
Test Performed	
Reliability:	Currently no bugs are found
Performance:	The execution time of the instrumented program can be several times slower if dynamic memory is used extensively. The postprocessing of the generated information (in order to produce the final report) is also slow.

4.1.6.3 Emma

Emma			
General Information			
Version:	2.1.5320	Licenses:	CPL v1.0
Authors:	Vlad Roubtsov	Maturity:	Stable
URL:	http://emma.sourceforge.net	Dependencies:	JVM
Description:	EMMA is an open-source toolkit for measuring and reporting *Java code coverage*. EMMA can instrument classes for coverage either *offline* (before they are loaded) or *on the fly* (using an instrumenting application classloader). It supports coverage types at the level of *class*, *method*, *line*, *basic block*. EMMA can detect when a single source code line is covered only partially. Coverage stats are aggregated at method, class, package, and "all classes" levels. Output report types: plain text, HTML, XML. All report types support drill-down, to a user-controlled detail depth. The HTML report supports *source code linking*. Output reports can highlight items with coverage levels below user-provided thresholds. Coverage data obtained in different instrumentation or test runs can be merged together.		
Constraints:			
Input/Output Information			
Language Analyzed:	Java		
Input Types	java bytecode	Input Formats	.class and .jar files
Output Types	Text, html, and XML	Output Formats	coverage report in a file
Other I/O comments			
Technical Information			
Devel. Lang:	Java	Documentation:	Reference manual, User Guide, Quick Start, Sample Report (on website and in distribution)
Information Computed:	Emma reports on the coverage of classes, methods, and basic block. Reports are generated presented at the level of methods, classes, packages and “all classes”.		
Extensibility:	Extension can only be implemented by modifying the source code		
Technical Constraints:			
Test Performed			

Reliability:	Emma was tested on two large proprietary Java applications of between 200 and 200 KLOC. The coverage information were reported accurately.
Performance:	Target program instrumented by Emma are a bit slower (less than 20% slower). Report generation is quite fast and is accounted for in the 20% overhead.

4.2 Analysis Tools for Other Repository-Data

In this section, we include a set of tools that are available on the Internet to analyze the repositories mentioned in section 3. Tools are divided into different categories, depending on the kind of repository data they analyze:

- Version Control
- Mailing lists archives

All tools mentioned are released under a F/OSS license.

4.2.1 Version Control Analysis Tools

4.2.1.1 CVSAnaly

CVSAnaly			
General Information			
Version:	0.9.3	Licenses:	GPL
Authors:	Alvaro Navarro, Gregorio Robles	Maturity:	Stable
URL:	http://cvsanaly.tigris.org	Dependencies:	cvs,mysql-server, python, python-mysql, python-mysqldb, python-imaging, gnuplot, ploticus.
Description:	CVSAnalY is a tool that extracts statistical information out of CVS (and recently Subversion) repository logs and transforms it in database SQL formats. It has a web interface - called CVSAnalYweb - where the results can be retrieved and analyzed in an easy way.		
Constraints:	Some features included for CVS are not included yet for SVN.		
Input/Output Information			
Language Analyzed:	Not Applicable		
Input Types	CVS or Subversion module	Input Formats	CVS or Subversion repository
Output Types	Graphics	Output Formats	PNG
Other I/O comments	CVSAnaly is executed as follows. $ python cvsanaly.py		
Technical Information			
Devel. Lang:	Python	Documentation:	Reference manual at http://cvsanaly.tigris.org/servlets/ProjectDocumentList

Information Computed:	Code repository log data: commits, committers, etc.
Extensibility:	Extension can only be implemented by modifying the source code
Technical Constraints:	
Test Performed	
Reliability:	Stable and reliable
Performance:	Good, it carries out the analysis quite fast

4.2.1.2 GlueTheos

GlueTheos			
General Information			
Version:	rev. 4907	Licenses:	GPL
Authors:	Gregorio Robles, Jesus M. Gonzalez-Barahona	Maturity:	Beta
URL:	http://libresoft.urjc.es/Tools/GlueTheos	Dependencies:	python, mysql, cvs, gnuplot, sloccount, textutils
Description:	GlueTheos has been developed to coordinate other tools to implement the methodology. It extracts snapshots from the CVS repository at several points in the past, and use the other tools to analyze them. It also normalizes information in an XML format, suitable for use as a detailed description of a project. From that format, it can get statistical data, graphical information and other formats (for instance, SQL tables) for further analysis. In addition, new applications can be added easily to obtain new data.		
Constraints:			
Input/Output Information			
Language Analyzed:	Not Applicable		
Input Types	CVS module	Input Formats	CVS repository
Output Types	Graphics and database	Output Formats	PNG, SQL

Other I/O comments	• Execution mode:
	  • Edit configuration file "config.py"
	  • Execute: gluetheos.py		

Technical Information			
Devel. Lang:	python, sh	Documentation:	File "README" and "REQUIRES" included in the repository.
Information Computed:	Code repository log data: commits, committers, etc		
Extensibility:	Extension can only be implemented by modifying the source code		
Technical Constraints:	• Unable to make diffs between distinct revisions of the project • Does not include rsync option for first downloading repository locally • Only works with CVS repository • Does not generate graphs		
Test Performed			
Reliability:	Poor		

Performance:	Low

4.2.1.3 Wholine

Wholine			
General Information			
Version:		Licenses:	GNU General Public License (GPL)
Authors:	Jorge Gascón	Maturity:	Stable
URL:	https://svn.libresoft.es/svn/projects/trunk/wholine2	Dependencies:	python, cvs, subversion
Description:	Wholine2 is a tool which analyzes a CVS/SVN repository, obtains all revisions of one repository and does an intensive analysis of the modifications of each line of code.		
Constraints:			
Input/Output Information			
Language Analyzed:	Not Applicable		
Input Types	CVS/SVN module	Input Formats	CVS/SVN repository
Output Types		Output Formats	Files with Statistics and graphs in the “Results” directory. Each repository is in a directory.

Other I/O comments	• Execution mode:
	  • python wholine project_name protocol protocol_project_url [OPTIONS]
	  • $ python wholine wholine svn https://svn.libresoft.es/svn/projects/trunk/wholine2		

Technical Information			
Devel. Lang:	Python	Documentation:	reference manual at https://svn.libresoft.es/svn/projects/trunk/wholine2/Doc/
Information Computed:	Code repository log data and differences between files: commits, committers, etc		
Extensibility:	Extension can only be implemented by modifying the source code		
Technical Constraints:	• If we need to run Wholine again with the same project we have to delete "wholine2/Results/<nombre proyecto>/log/*" files manually.		
Test Performed			
Reliability:	Stable and reliable		
Performance:	Good		

4.2.1.4 Carnarvon

Carnavron			
General Information			
Version:		Licenses:	GNU General Public License (GPL)
Authors:	Alvaro Navarro, Carlos González	Maturity:	Stable
URL:	http://carnarvon.tigris.org	Dependencies:	python, cvs, subversion, python-2.x-MySQLdb, gnuplot.
Description:	Carnarvon analyzes how old the software system is on a per-line basis and extracts figures and indexes that make it possible to identify how `old' the software is, how much it has been maintained and how much maintenance effort it may undergo in the future.		
Constraints:			
Input/Output Information			
Language Analyzed:	Not Applicable		
Input Types	CVS/SVN modules	Input Formats	CVS/SVN repository
Output Types	HTML including Graphs	Output Formats	Directory of HTML files

Other I/O comments	• Execution mode:
	  • Install: python setup.py install
	  • Configure: carnarvon -w my.conf
	  • Run: carnarvon my.conf
	  • Analysis: After carnarvon finishes the analysis of the given project, run the other 40 tools in order to get graphs and a nice website: carnarvon2web my.conf		

Technical Information			
Devel. Lang:	Python	Documentation:	Quick Start Guide available at http://carnarvon.tigris.org/documentation/quick-guide.html
Information Computed:			
Extensibility:	Extension can only be implemented by modifying the source code		
Technical Constraints:	• May be interesting that Carnarvon can do the repository check out by itself. But, authors say that this is a feature.		
Test Performed			
Reliability:	Stable and reliable		

Performance:	Good

4.2.1.5 CvsGraph

CvsGraph			
General Information			
Version:	1.6.1	Licenses:	GPL
Authors:	B. Stultiens	Maturity:	Stable
URL:	http://www.akhphd.au.dk/~bertho/cvsgraph	Dependencies:	yacc, libgd2
Description:	CvsGraph is a utility to make a graphical representation of all revisions and branches of a file in a CVS/RCS repository. It has been inspired by the 'graph' option in WinCVS, but I could not find a stand-alone version of this graph code.		
Constraints:			
Input/Output Information			
Language Analyzed:	Not Applicable		
Input Types	CVS module and file	Input Formats	CVS repository
Output Types	Image	Output Formats	png file

Other I/O comments	• Execution mode:
	  • cvsgraph [options] <file>
	  • i.e: `cvsgraph -r /home/to/repository -m module -o mygraph.png myfile.c,v`		

Technical Information			
Devel. Lang:	C	Documentation:	Installation and reference manual at http://www.akhphd.au.dk/~bertho/cvsgraph
Information Computed:	Graph showing the change history of a file in a CVS repository		
Extensibility:	Extension can only be implemented by modifying the source code		
Technical Constraints:			
Test Performed			
Reliability:			
Performance:			

4.2.2 Mailing Lists Archives Analysis Tools

4.2.2.1 MailingListStats

MailingListStats			
General Information			
Version:	0.3.1	Licenses:	GNU General Public License
Authors:	Israel Herraiz	Maturity:	Stable
URL:	https://svn.libresoft.es/svn/projects/trunk/mailingListStat/	Dependencies:	python, MySQL
Description:	MailingListStats is a tool for mapping mbox files of any mailing list to a database.		
Constraints:			
Input/Output Information			
Language Analyzed:	Not Applicable		
Input Types	URL or mboxes	Input Formats	URLS of each mailing list or a directory with the mboxes of each mailing list separated by directories.
Output Types	Database	Output Formats	MySQL Database
Other I/O comments	A database with information about headers of each parsed mbox and its mailing list		
Technical Information			
Devel. Lang:	Python, MySQL	Documentation:	HOW-TO guide (=quick start guide) available at https://svn.libresoft.es/svn/projects/trunk/mailingListStat/doc/MLS_Howto.txt
Information Computed:	Mboxes from public mailing lists		
Extensibility:	Extension can only be implemented by modifying the source code		
Technical Constraints:	• We can not update the database of one mailing list without re-parsing all mailing list again. • Duplicated information is introduced in the database when running the tool on the same mbox several times. Thus, the database must be emptied before running updated mboxes. • No error tolerance: MLS can not resume its work when the a system shutdown is unannounced. We have to run MLS from the		

	beginning for all projects(mailing lists) because MLS does not support resume and then we could find consistency problems in MLS database.
Test Performed	
Reliability:	Stable and reliable
Performance:	Good

4.2.2.2 SEAL

SEAL			
General Information			
Version:		Licenses:	GNU General Public License
Authors:	Gregorio Robles, Roberto Andradas Izquierdo	Maturity:	Alpha
URL:	https://svn.libresoft.es/svn/projects/branches/improvedSeal	Dependencies:	Python, MySQL
Description:	SEAL is a tool for identifying people in a mailing list conserving their privacy.		
Constraints:			
Input/Output Information			
Language Analyzed:	Not Applicable		
Input Types	Files	Input Formats	XML
Output Types	Database MySQL	Output Formats	records in db MySQL
Other I/O comments	Execution Mode: • Export data about a mailing list from MailingListStat? database: python xx2xml.py > data.xml (xx2xml.py is a python script which must be configured in order to connect to the MLS database) • Run SEAL as follows: $ python seal-feed.py data.xml		
Technical Information			
Devel. Lang:	Python, MySQL	Documentation:	reference manual available at https://svn.libresoft.es/svn/projects/branches/improvedSeal/README
Information Computed:	Personal identities (Name, email, etc.)		
Extensibility:	Extension can only be implemented by modifying the source code		
Technical Constraints:	• SEAL has to recalculate all matches (related identities) each time new identities are introduced. However, this task is not slow. • No error tolerance: SEAL can not resume its work when the system crashes. In other words, SEAL does not support a resume operation.		
Test Performed			
Reliability:	Medium		

Performance:	Good

5. Advanced Analysis Tools and Techniques

This section identifies techniques and tools for more advanced processing of F/OSS data. In several cases, the existing tools provide building blocks for producing information useful for QUALOSS quality models while in other case, analysis are currently conceptual and tools will have to be implemented during WP2. We further note that current tools for advanced analysis often compute results too imprecise to be directly used in QUALOSS quality models. Before using these results, we must then find ways to improve their precision or alternatively conduct several studies to show that even when imprecise, these results can still provide valuable information for quality models. Furthermore, we highlight the work by Gasser et al. (2004) on the requirements of empirical studies of software repositories. Basically, those requirements are (1) direct reflection of the reality, (2) adequate coverage, (3) examination of representative levels of variance, (4) demonstration of adequate statistical significance, (5) comparability across projects, (6) repeatability and (7) testability and evaluability of results. For the case of the empirical validation of the QUALOSS model, those requirements should be taken in account.

Our presentation follows the same topic break down as Section 3, that is, based on types of data sources. We describe advanced tools and techniques for F/OSS product release in Section 5.1, for version control data in 5.2, for Mailing List Archives in 5.3, for other data internal to F/OSS project repositories in 5.4, and for external data in 5.5. Finally, we added Section 5.6 to discuss potential analysis based on data from multiple sources.

5.1 F/OSS-Product-Release Advanced Analysis

A product release always includes the source code. Furthermore, it often makes available binaries, different types of documentation and possibly, regression tests at the unit and system levels are also distributed as part of a product release. Below we describe advanced analysis for measuring the source code statically and dynamically, i.e., using test runs. We also suggest analyses of potential interest for documentation.

5.1.1 Advanced Static Analysis

By definition of F/OSS, a product release includes a snapshot of the source code at a given release date. Based on tools mentioned in Section 4.1, basic metrics can be obtained. In addition to these measurements, techniques have been developed to search code for more advanced information such as:

- deadlock,
- buffer overflow,
- array out-of-bound,
- pointer dereferencing,
- memory leaks,
- problematic allocation and deallocation of memory,
- different code smells, etc.

A good summary of the available tools and techniques to study potential problems in source code (like those mentioned above) is the third chapter of Spinellis (2007).

Identifying these problems in a product release could be used to estimate quality characteristics related to robustness and even evolvability. For example, a product release with higher density of potential deadlocks may be considered less robust or particular code smells may reveal rigidity hence code less evolvable. Evidence of this could be corroborated by bug reports or feature requests stored in the bug tracking system.

There exists an extensive list of F/OSS tools to check for the problems listed above, among others,

- Jlint (http://jlint.sourceforge.net/) checks for thread synchronization problems in Java bytecode.
- Flawfinder (http://www.dwheeler.com/flawfinder) checks for code smells likely to introduce vulnerabilities in the code.
- C-Code Analyzer (http://www.drugphish.ch/~jonny/cca.html) also highlights code issues likely to expose software components to vulnerabilities.
- Bandera (http://bandera.projects.cis.ksu.edu/) perform model checking on Java code to identify potential deadlock and buffer overflow
- ESC/Java (http://secure.ucd.ie/products/opensource/ESCJava2) uses a theorem proofer to identifie potential runtime error due to potential null pointers or arrays out-of-bound.
- FindBugs (http://findbugs.sourceforge.net) and PMD (http://pmd.sourceforge.net) both scan the code for bad smells that are likely bugs or will likely lead to bugs in the future.

The main challenge in using results from these tools is the lack of precision. That is, detecting all real possibilities of dead lock, buffer overflow, etc. in any given program is an undecidable problem. Thus, these tools usually limit their scope. Due to different trade-offs in their algorithms (for decreasing false positives while not eliminating true negative), these tools produce different warnings even when performing similar checks (Rutal et al. 2004).

Rutal et al. propose a meta-checker that combines results from different tools. QUALOSS could use a similar approach by studying if measures based on counts of common warnings across several tools provide more accurate quality indicators.

In addition to analyses to solve the problem mentioned above, indicators related to ***software architecture*** may also provide interesting information regarding the evolvability or robustness of a product release (Whitmire 1997). The assumption is that software architectural information will help to highlight evolvability and also robustness since architectural patterns propose proven solutions to particular software design problems. However, not every pattern applies to every F/OSS project, for example, the software architecture of a device driver is based on different constraints than the architecture of a web server. So the challenge in using architectural information in QUALOSS quality models is to determine what patterns benefit what software. This information is needed in order to transform software architectural data into dependable data to use in quality models. Since QUALOSS plans on developing quality models based on the Goal Question Metric paradigm, we may also ask stakeholders if and why software architecture information is important with respect to evolvability and robustness. Of course, not all F/OSS integrators may care about or have enough

expertise on architectural data. Those most likely to provide valuable feedback regarding software architecture are F/OSS developers or technical F/OSS integrators who, as part of their business, customize F/OSS components to clients' needs.

Another static analysis consists in studying ***copyright ownership*** of source code files. Depending on the F/OSS license used by a project, copyright ownership could provide interesting information as to the risk for a F/OSS project to turn into a proprietary project. For example, certain licenses allow copyright owners to switch the license of code segments they wrote. In turn, software product copyrighted by just a few people would show a higher risk to turn into a proprietary project in the future. COOD is a tool that extracts code copyright ownership (http://vipul.net/perl).

The static analysis mentioned above attempts to be fully automated. Conversely, some source code analysis could be performed manually. However, due to the size of source code, manual analysis would only inspect small source code samples. An important factor related to manual analysis is that it must remain objective to eliminate the human factor as much as possible. In turn, such analyses must therefore rest on specific procedures and checklists. Given the use of GQM, it might be possible to let users tailor QUALOSS models by adapting checklists to their needs however, the actual manual procedure for selecting code samples and for applying the manual analyses must be strictly imposed and respected.

Currently, we envisage using manual analysis to verify the quality of ***source code comments***. Comment quality may influence code robustness and evolvability. In addition to randomly verifying source code comments, we may also perform more advanced manual checks such as select a few bugs in the bug tracking system and then verify that code modifications related to bug correction have comments including the unique bug report numbers of these bugs.

5.1.2 Advanced Dynamic Analysis

Section 4.1 describes tools performing the simplest kind of dynamic analysis i.e., code coverage. Although simple, this analysis often provides very important indicators on the quality of testing, which in turn helps in estimating robustness. In addition to coverage, it is possible to perform more advanced dynamic analysis. In order to do so, we may have to develop custom instrumentations for monitoring specific aspect of execution.

Software code may be instrumented at different levels, the most common being source code and bytecode. For Java technology, there exists several interesting bytecode instrumenting tools. These tools are BCEL (http://jakarta.apache.org/bcel), SOOT (http://www.sable.mcgill.ca/soot), and JavaAssist (http://www.csg.is.titech.ac.jp/~chiba/javassist). For instrumenting various types of languages such as Java, C, C++, Python, there is TAU, which also provides an instrumentation API (http://www.cs.uoregon.edu/research/tau/home.php).

The exact kind of instrumentation useful to the QUALOSS quality models is currently unknown. However, if, from the work of task 1.3, it is determined that a certain instrumentation would help in measuring certain quality metrics then the tools above will provide the needed building block.

Most of the analysis mentioned in the previous subsection on advanced static analysis have also been approached using dynamic analysis, for instance, (Ruwase-Lam 2004). Dynamic approaches for identifying deadlock buffer over flow, memory leaks, etc. are usually more precise, in the sense that errors discovered are real problems. However, these dynamic analyses are usually unsafe since they do not identify all problems in the code but only those highlighted in test runs.

5.1.3 Hybrid Analysis

Hybrid analysis consists in combining static and dynamic analyses. It is usually applied in one of two ways: (1) a static analysis is performed on an entire program and then dynamic analyses are used to prune or prioritize results or (2) dynamic analysis is first performed to identify a few code subcomponents on which static analysis needed to be computed.

Many hybrid analysis efforts have been applied to the problem of buffer overflow, deadlock detection, pointer dereferencing, and other (Artho-Biere 2005), (Aggrawal-Jalote 2006). In (Ernst 2003), Ernst presents possible synergies between static and dynamic analysis tools.

Beside these advanced hybrid analysis, QUALOSS could definitely benefit from even much simple checks, for example, by dynamically measuring the most executed segments of code during regression testing and then statically verifying that these code segments are well documented and display a low complexity.

5.1.4 Analysis of Build-Install Mechanism

A product release often comes with binaries, commonly for MS Windows systems. However for many other operating-system platforms, a F/OSS product release must be built from sources. It is therefore fundamental that the product release can be build automatically and easily.

Automated analysis of the simplest kind could scan for the common build files such as configure, makefile, and build.xml then run the traditional appropriate command to build the product. such as `configure; make` potentially followed by `make test; make install; make clean`.

If required, more thorough analysis of the content of build files could be developed to verify their range of applicability.

5.1.5 Analysis of Product-Release Documentation

Beside code and test suite, a product release often includes a series of documentation documents. It is paramount to verify that documentation is available for the specific product release, and that it is up to date and reaches a certain level of quality. It may be quite hard to automate such verification. However, it is feasible to develop a standard manual procedure to measure documentation quality.

Although measuring documentation quality seems more related to usability, it also has an impact on evolvability and robustness. For example, the documentation may explain extensively how to tune the product release to improve its robustness. In addition, the

robustness and evolvability of documentation itself may be considered. For example, is the user guide broken down logically in small independent subsections?, does it avoid redundancy and are cross references between related sections of the documentation explicitly stated so as to ease documentation maintenance? Are there different types of documentation to address the concerns of new users as well as users familiar with previous product releases?

5.2 Version-Control Advanced Analysis

Section 3.2 specifies that version control systems keep track of two different types of data:

- Files under version control such as source code files
- Meta information about commits

Analysis on both types of data are discussed below.

5.2.1 Historical Analysis of Version-Controlled Files

Regarding Source Code Files, a version control system makes it possible to obtain the whole code history from the beginning of the project until the present day. A version control system also allows obtaining source code files for a given date.

Section 4.1 already presented tools to measure source code and Section 5.1 also mentions different advanced analysis that could be performed on source code at a selected point in time. What version control data brings is the ability to study historical evolution of source code. Some effort have already studied the evolution of size and complexity of F/OSS projects (Godrey and Tu, 2000; Robles *et al.,* 2005; Koch, 2005). Moreover, size was also used to estimate the cost of substitution of the project[3] (Amor *et al.,* 2005). QUALOSS can leverage on the information produced by these works and even take them one step further by applying advanced analysis over time and verifying whether a selected F/OSS project shows improving or regressing trends.

One particular interesting study could verify whether or not it is common for large, popular F/OSS projects to go through phases of reengineering, refactoring or rewrites at various moments of their life.

Regarding other files, version control may also contain test suite data such as unit tests and system tests. It is possible to use these tests for the dynamic analysis mentioned previously. It is also possible to study the historical growth of the test suite.

F/OSS projects may also store documentation files in version control systems. In such cases, the historical evolution of documentation may reveal interesting patterns. For example, do we see that documentation files are brought up to date before a product release? If not, this could indicate mismatch between the system and its documentation.

5.2.2 Version Control Metadata Analysis

Version control system metadata allows to obtain very valuable information about the activity in the different parts of the source code tree, about the productivity of the

[3]The cost of substitution is the amount of money that a company should spend to develop a program of the same size in a “closed” environment.

developers, about code ownership. Furthermore, when related to other data source such as mailing list archives or bug tracking data, it is also possible to reconstruct the interactions among community members of a F/OSS project in order to discover the existence of sub-communities or sub-groups (Lopez-Fernandez *et al.,* 2006).

Log texts in commits have traditionally been used to identify bug fixes. When a commit is a bug fix, a given pattern is supposed to appear in the log text. For instance, the number of the bug report, the words “bug fixing”, etc. Log texts have also been used to classify the changes in the version control system, using the least common words in English which appear in the text, and clustering analysis to identify the different categories. (Amor *et al.*, 2006)

Meta information and source code logs recorded by version control systems may also be used to study project turnover and takeover aka generation analysis. By measuring the activity of the core group of a F/OSS project over time, we can determine if a generational relay has occurred in the project. This finding is important, because a generational relay is supposed to be needed in healthy projects. In other words, the project needs new people to take leadership; it can not rely on the shoulders of “code gods” (Robles and Gonzalez-Barahona, 2006).

Meta information can also be used to categorize committers as coders, translators or artists depending on the types of files they edit and modify. It is also possible to study code ownership at the level of files, directories or modules.

Zimmerman and Weissberger propose a methodology to deal with the particularities of CVS when empirically studying such repositories (Zimmerman and Weissberger 2004) . In particular, the quality of results is heavily influenced by how the data is preprocessed before preforming analysis on it. The commonly performed 4 preprocessing steps are: (1) data extraction, (2) transaction recovery, (3) mapping of changes to fine grained entities (e.g. mapping changes to other code entities that files such as functions, classes, etc.) and (4) data cleaning. The QUALOSS methodology will carefully consider how each of these steps is performed when analyzing CVS repositories so as to obtain high quality data from the preprocessing phase. This will be crucial in order to obtain reliable measurements on the metrics mentioned hereafter.

For step 1, we will use CVSAnaly, a tool developed by URJC that has reach a good level of maturity when performing extraction for CVS and also Subversion.

Concerning the step 2 of CVS preprocessing, the meta information associated to every commit can be used to reconstruct the modification requests made to several files. In particular, algorithms have been proposed for this purpose (German, 2004).

Step 3 will depend on the metrics selected by QUALOSS and step 4 may in certain cases require interaction with community members of the select F/OSS projects so as to validate how to clean the data or even validate the already cleaned data (knowing the cleaning operation performed on it.)

Below, we enumerate metrics obtained from the analysis of version control system metadata, All the proposed metrics are intended to be measured over time, for instance

on a monthly basis or on a product-release-time basis. It is worth noting that in many cases, the evolution of measurements gives more interesting indicators than measurements only taken at a single point in time.

- Number of developers making changes to the project
- Number of non-active developers
- Number of changes made to the project
- Number of modules present in the project
- Number of modules changed in the project
- Number of files present in the project
- Number of files changed in the project
- Relationships between developers at various granularity level (Files and Modules)
- Relationships between modules
- Generation analysis

All the proposed metrics could be measured discriminated by file type using the meta information associated to every commit.

Comparison of source code modifications between two or more product releases not using version control data enables studying trends in source code evolution. However, using the finer grain data found in version control systems, the study of changes can be performed at a much lower level. At this low level, interesting interaction patterns may become observable, for example, interaction between developers, or systematic modifications of a group of files.

5.3 Bug Tracking Advanced Analysis

Data obtained from bug tracking systems is fundamental for performing quality analysis. Below is a list of metrics automatically extractable from bug tracking systems. It is actually more meaningful to study the evolution of these metrics over time; for example, in the last six months or in the period since the last major or minor release:

- Number of bugs, differentiate by status
- Number of bugs fixed vs opened
- Mean and standard deviation of time elapsed to fix or to close a bug.
- Number of comments in the bug report.
- Number of reporters (number of people who found and reported at least one bug)
- Number of developers attending to bugs

In addition to the simple metrics above, the complexity of bugs resolution can also be studied almost automatically when a patch for fixing the bug is attached to the final bug report.

The relationships between users reporting bugs and developers attending those reports could provide information concerning the procedure followed by the community. Techniques could be developed to discover the actual the communication paths between users and developers, (are there clusters of developers related to cluster of users?)

Also, correlations between bug tracking reports and other data sources such as mailing list archives are likely to yield interesting information for QUALOSS quality models

(Herraiz *et al.*, 2005). Another interesting cross correlation is to try to track the relationships among bug reports and changes in the source code repository. This problem has been already addressed by Mockus et al. (Mockus et al. 2002) and German and Mockus (German and Mockus 2003) with partial results.

Beside fully automated measurements of bug tracking data, it may be interesting to study the semantic quality of bug reports. This analysis is likely to be performed manually hence only small representative samples can be verified feasibly. An example of an interesting point to check is whether or not bug reports initially well detailed and explained have more chances to be addressed by developers compared to brief bug reports. For instance, reports automatically generated when an application crashes are often very precise including system information and potentially a memory snapshot. So are these report taken into account and their bugs fixed faster?

5.4 Mailing List Archives Advanced Analysis

As mentioned in Section 3.4, when mailing list archives are stored in RFC822 format and are not stripped out then the following metrics can be obtained automatically:

- Number of messages over time
- Number of people writing in the list over time
- SNA methods to study the flow of information within the community and their evolution over time
- Mean and standard deviation of length of the threads over time.
- Statistics of usage of the different programs in the mailing list

Extracting common words from the content of email messages could also help to identify topics discussed by the community. Studies comparing keywords evolution may reveal important reoccurring topics. Correlating topics with people involved in the discussion of those topics may also augment existing analysis related to the discovery of social networks within the community of a F/OSS project.

Manual analysis of message content may also provide interesting information. For example, how helpful are email messages for answering a support question. Obviously, there are usually too many messages in an archive to analyze them all by hand however, a small, representative sample of emails in the archives may provide valuable information for QUALOSS quality models.

5.5 Advanced Analysis on Other Data Internal to a F/OSS Project

Section 3.5 list three other sources of data commonly found in F/OSS project:

- On-line documentation
- Web and Wiki pages
- IRC logs

5.5.1 Analysis of On-Line Documentation

Analysis of documentation was already discussed as part of Section 5.1. Similar analyses could be performed on on-line documentation. It may also be interesting to verify that different version of on-line documentation match each product release, and not just documentation for the most recent version.

5.5.2 Analysis of Web and Wiki Pages

Simple analysis of web sites may possibly be automated, for example, an analysis to verify the graph of page reachability to guarantee that information can be obtained with an acceptable number of clicks. In other words, such an analysis would ensure that a website follows the recognized website guidelines. However, given the unstructured nature of websites, advanced analysis are likely to be manual.

Website Analysis will also have to take into account the age of project. In fact, initially, it may be acceptable for small, young F/OSS projects not to provide much information on their website. However, as they grow in popularity, their websites will mostly need to grow. To ensure a healthy growing community, popular projects are expected to present certain information explicitly, for example, transparency in vision, management decision. Web and wiki pages often reflect such information by having pages dedicated to the mission statement, the structure of the steering committee of a F/OSS project, and many other data.

5.5.3 Analysis of IRC Logs

IRC logs widely vary in their content. In some cases, they are used for support or tutorial session while in other managerial issues may discussed; yet in other cases, design decisions may be addressed. Analysis of IRC logs will likely have to be manual to be of any value.

When IRC logs hold the results of decisions, it may be interesting to verify who participates in the IRC session and whether the decision making process respected the procedure decided by the leadership of a F/OSS project (for example, as it is stated in the decision-making procedures on the website of the F/OSS project.)

Due to the unstructured nature of information found in IRC logs, it is unlikely that QUALOSS finds a way to formalize the use of such information. However, this a priori expectation may not reflect the real world usage of IRC and, in turn, thorough investigation of IRC log content is need before deciding whether or not QUALOSS quality models can use their information.

5.6 Advanced Analysis on Other Data External to a F/OSS Project

Beside the data collected and shared by a F/OSS project, several other sources of information mentioned in Section 3.6 also provide interesting F/OSS project data. Potential analysis on data of alternate sources currently identified are discussed in the subsections below.

5.6.1 Analysis of F/OSS Data Provided by Other Projects

FLOSSMOLE (Howison et. al, 2006) is a project that collects a series of information about F/OSS projects available on SourceForge, FreshMeat, and Savannah. FLOSSMETRICS is another project that will also provide a similar repository of F/OSS projects data.

We currently anticipate a much stronger collaboration with FLOSSMETRICS since several FLOSSMETRICS partners are also involved in QUALOSS. This creates a feedback mechanism between the two projects. In other words, QUALOSS may influence the database schema created by FLOSSMETRICS. Such a feedback does not exist with

FLOSSMOLE. However, FLOSSMOLE data may still be of interest either to validate FLOSSMETRICS data or because it provides data not collected by FLOSSMETRICS.

FLOSSMETRICS data are interesting because they provide filtered information. In many cases, version control repositories, bug tracking data and email archives are cluttered with noisy data. Thanks to FLOSSMETRICS, a first level of filtering will improve data quality. When possible, measurements may be obtained directly from FLOSSMETRICS data. In other cases, FLOSSMETRICS data will provide valuable information so that QUALOSS tools can access relevant, valid data in F/OSS project repositories avoiding noisy, erroneous data.

QUALOSS may also provide tools to FLOSSMETRICS so that certain analyses can be computed in the scope of FLOSSMETRICS enabling QUALOSS to simply retrieve results.

5.6.2 Analysis of Vulnerability Databases

Currently, two related, open data sources inventorying vulnerabilities have been identified, namely, the Common Vulnerability Enumeration (CVE) hosted by MITRE Corporation at http://cve.mitre.org/cve and the National Vulnerability Database (NVD) hosted by the National Institute of Standards and Technology (NIST) and searchable at http://nvd.nist.gov/nvd.cfm?advancedsearch.

NVD augments CVE vulnerabilities with a series of information such as severity score, created based on explicit and objective procedures. It also posts patches when made available by the vulnerable product representatives. Furthermore, NIST has developed a language called OVAL to define new vulnerabilities and to facilitate the search of the OVAL repository for particular vulnerabilities.

Information about vulnerabilities in a F/OSS product release could provided valuable information for use in robustness quality models. Furthermore, studies of historical evolution of vulnerabilities in a F/OSS product could also provide information regarding the ability of the community to produce reliable software.

5.6.3 Analysis of Publication Databases

One particular sign of popularity and maturity, which may show signs of evolvability and robustness, is the publication of a new book. Given the broadness of the Amazon database, it could be used to identify book publications related to a particular F/OSS product.

In addition to Amazon, scientific publications databases, which improve F/OSS products, could reveal the presence of an active research community behind the F/OSS product. Given that innovation is an important factor in evolution, information related scientific articles could provide a useful indicator to QUALOSS quality models.

5.6.4 Analysis of News Websites and Archives

Presence in the press is an important factor to help the success of F/OSS projects. Some interesting analysis could categorize and count press articles inventoried by a few trusted sources such as Slashdot or FLOSSPlanet.

The challenge in creating objective news analysis is that many factors influence the impact of a news article, for example, the broadness of distribution of the newspaper and the credibility of the writer. However, media presence is a clear indicator of the health of a F/OSS project hence we must further study the possibility to create objective procedures to measure news impact.

5.7 Multiple Data Source Analysis

When possible to cross reference data between multiple data sources, the added value of the analysis is likely to increase. For example, cross referenced information between version control repositories, bug tracking reports, and mailing list archives could reveal elaborate software development procedures between the community members of a F/OSS project. These procedures would likely not be observable when analyzing a single data source.

Fully automated analysis for multiple data sources is unlikely. However, if the mechanism for cross referencing data is known, it may be possible to automate part of the analysis. For example, if version control logs always include a field mentioning a unique bug report or unique feature request, then it may be possible to automatically cross reference version control and bug tracking data.

Except for a few F/OSS projects, we do not anticipate that traces between data of multiple repositories will be kept. In turn, further investigation is required before justifying the implementation of tools or analyses based on data cross reference as input.

Further cross referencing may be done between release data and news archive websites. In fact, having community members active in promoting a new product release in the news at release time shows community versatility, which is definitely an indication of robustness. It may also be possible to cross-relate data in vulnerability databases and in bug reports.

6. Conclusion

This deliverable presents three important pieces of information to take into account when building QUALOSS quality models.

First, the different types of data available on F/OSS projects are presented. The main F/OSS data sources are product releases, version control repositories, bug tracking systems, mailing list archives, and other unstructured sources such as on-line documentation, website, and IRC logs. Furthermore, there also exists information about F/OSS projects that is stored in external repositories, that is, data not controlled by the F/OSS projects. In particular, FLOSSMETRICS and FLOSSMOLE provide filtered data extracted automatically from renown forges like SourceForge, FreshMeat and Savannah. Other external repositories are those inventorying software vulnerabilities such as the one provided in the National Vulnerability Database. Finally, publication databases and news archives provided by trusted sources may reveal interesting information related to evolvability and robustness of F/OSS projects.

Second, this document enumerates existing tools to process the data from sources mentioned above. From this enumeration, we find that many tools for processing code are available. Similarly, there are also a few tools for processing version control data. However, much fewer tools are available to extract and analyze other data sources. Only a single tool was found for analyzing mail archives. Concerning bug tracking systems, data extraction should be fairly simple if access to the bug tracking database is granted. However this would make QUALOSS dependent on F/OSS projects granting the appropriate access permissions. This is why other data sources such as FLOSSMOLE and FLOSSMETRICS databases may provide appropriate alternatives since they have already negotiated these access rights with well-know F/OSS forges.

Third, tools and advanced analyses are described for each data source. Advanced analyses are likely to provide valuable information to the QUALOSS quality models. Finally, we note that some of these advanced analysis may be manual. Although QUALOSS plans on automating the application of most quality models, some manual analysis is acceptable, especially if they yield highly important information regarding the evolvability and robustness of F/OSS projects.

At this stage, this deliverable presents the necessary tools and analysis to process F/OSS data from the mentioned sources. Nonetheless, additional tools and analysis may augment our current list; especially as a result of task 1.3, which combines the outcomes of tasks 1.1 and 1.2.

7. References

(Artho-Biere 2005) C. Artho, A. Biere. *Combined Static and Dynamic Analysis.* Technical Report 466, Dept. of Computer Science, ETH Zürich, 2005, full version of paper that appeared in AIOOL'05.

(Aggrawal-Jalote 2006) Ashish Aggarwal, Pankaj Jalote, *Integrating Static and Dynamic Analysis for Detecting Vulnerabilities* In Proceeding of COMPSAC pp. 343-350, 30th Annual International Computer Software and Applications Conference (COMPSAC'06), 2006.

(Amor et al., 2005) Amor, González-Barahona, Robles and Herraiz. Measuring Libre Software using {D}ebian 3.1 (Sarge) as a Case Study: preliminary results. *Upgrade Magazine.* Vol VI, issue 3. 2005.

(Amor et al., 2006) Amor, Robles and Gonzalez-Barahona. *Discriminating Development Activities in Versioning Systems: A Case Study.* Proceedings of the International Workshop on Predictor Models in Software Enginnering. 2006.

(Ernst 2003) Michael D. Ernst. *Static and dynamic analysis: Synergy and duality,* In *WODA 2003: ICSE Workshop on Dynamic Analysis*, (Portland, OR), May 9, 2003, pp. 24-27.

(Gasser et al., 2004) Les Gasser, Gabriel Ripoche, and Robert Sandusky. *Research infrastructure for empirical science of FOSS*. In Proceedings of the International Workshop on Mining Software Repositories, Edinburgh, Scotland, UK, 2004.

(German, 2004). German. *An empirical study of fine-grained software modifications.* Proceedings of the International Conference on Software Maintenance. 2004.

(German and Mockus, 2003) Daniel M. German and Audris Mockus. *Automating the measurement of open source projects*. In Proceedings of the 3rd Workshop on Open Source Software Engineering, Portland, Oregon, USA, 2003.

(Godfrey and Tu, 2000) Godfrey and Tu. *Evolution in Open Source Software: A Case Study.* Proceedings of the International Conference on Software Maintenance. pp. 131-142. 2000.

(Herraiz *et al.* 2005) Herraiz, Robles and Gonzalez-Barahona. *Towards Predictor Models for Large Libre Software Projects.* Proceedings of the International Workshop on Predictor Models in Software Engineering. 2005.

(Herraiz *et al.* 2006) Herraiz, Robles, Amor, Romera and Gonzalez-Barahona. *The Processes of Joining in Global Distributed Software Projects.* Proceedings of the International Workshop on Global Software Development for the Practitioner. 2006.

(Howison et al. 2006) Howison, J., Conklin, M., Crowston, K. (2006). FLOSSmole: A collaborative repository for FLOSS research data and analyses. *International Journal of Information Technology and Web Engineering. 1(3)*. July-September, 2006. pp 17-26.

(Koch, 2005) Koch. *Evolution of Open Source Software Systems – A Large Scale Investigation.* Proceedings of the 1st International Conference on Open Source Systems. 2005.

(Li et al. 2005) Tong Li, Carla S. Ellis, Alvin R. Lebeck, and Daniel J. Sorin. *Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution,* USENIX Annual Technical Conference, April 2005.

(Lopez-Fernandez *et al.*, 2006) Lopez-Fernandez, Robles, Gonzalez-Barahona and Herraiz. Applying Social Network Analysis Techniques to Community-Driven Libre Software Projects. *International Journal of Information Technology and Web Engineering*, Vol. 1, Issue 3. 2006.

(Mockus et al., 2002) Audris Mockus, Roy T. Fielding, and James D. Herbsleb. *Two case studies of Open Source software development: Apache and Mozilla*. ACM Transactions on Software Engineering and Methodology, 11(3):309–346, 2002.

(Robles *et al.*, 2005) Robles, Amor, Gonzalez-Barahona and Herraiz. *Evolution and Growth in Large Libre Software Projects.* Proceedings of the International Workshop on Software Evolution. pp. 165-174. 2005.

(Robles and Gonzalez-Barahona, 2006) Robles and Gonzalez-Barahona. *Contributor Turnover in Libre Software Projects.* Proceedings of the International Conference on Open Source Systems. 2006.

(Rutal et al. 2004) Nick Rutal, Christian B. Almazan, and Jeffrey S. Foster, *A Comparison of Bug Finding Tools for Java*, Procedings of the International Symposium on Software Reliability Engineering. 2004, pp 245-256, (URL: http://doi.ieeecomputersociety.org/10.1109/ISSRE.2004.1)

(Ruwase-Lam 2004) O. Ruwase and M. S. Lam, *A Practical Dynamic Buffer Overflow Detector,* In Procedings of the 11th Annual Network and Distributed System Security Symposium, February 2004.

(Spinellis 2007) Diomidis Spinellis. *Code Quality: The Open Source Perspective*. Addison Wesley, 2006. ISBN 0-321-16607-8.

(Whitmire 1997) Scott A. Withmire, Object-Oriented Design Measurement, Wiley & Sons, Sepetember, 1997, (ISBN: 978-0471134176)

(Zimmermann and Weissgerber, 2004) Thomas Zimmermann and Peter Weissgerber. *Processing CVSdata for fine-grained analysis*. In Proceedings of the International Workshop on Mining Software Repositories, Edinburgh, Scotland, UK, 2004.