BuCo Reporter: Mining Software and Bug Repositories

Elvis Ligu Theodore Chaikalis Alexander Chatzigeorgiou

Department of Applied Informatics
University of Macedonia
Thessaloniki
Greece
{mai1315, chaikalis, achat} @ uom.edu.gr

ABSTRACT

Version Control and Bug Tracking Systems are essential tools in contemporary software development methods and are widely employed by development teams for systematic source code revision tracking and effective bug management. By combining information provided from both tools, a maintainer could shed light to various qualitative and quantitative characteristics of software projects. BuCo Reporter is a Java application that mines source code and bug repositories and by combining these kinds of information provides useful reports that describe project history. BuCo also calculates several bug and source code metrics. Its novelty lies in its modular structure which allows for effortless extensibility. Moreover the framework that is provided is very easy to install, use and modify without requiring background knowledge.

Categories and Subject Descriptors

[**Software configuration management and version control systems**] [**Software libraries and repositories**]

Keywords

Mining Software Repositories, Bug Tracking Systems, Version Control Systems, Software Evolution

1. INTRODUCTION

Mining Software Repositories is a broad field of research that addresses the historical analysis of software data in order to investigate the evolution process of a software system and derive useful conclusions about its characteristics. Many approaches have been introduced to extract pertinent information from the vast amount of data that software repositories contain [15]. These pieces of information often regard previous design decisions, bug corrections, committer activity and even refactorings, could be exploited to improve software design and facilitate software maintenance. In general software repositories are comprised by Version Control and Bug Tracking Systems (VCS, BTS) and usually a mailing list. The size of such repositories is often vast (e.g. Sourceforge hosts more than 300,000 projects) and they have been successfully termed the 'new library of Alexandria' [10]. It is obvious that any tool that could automatically exploit the aforementioned systems combine the data and extract useful information would be extremely valuable.

To this end, automatic Software Repository Mining tools have been developed under a web-based architecture defining a new category called Software Engineering Research Platforms. Characteristic examples are the Alitheia Core [11], the BOA tool [10], the BugMaps [12] and the Kenyon [4], which are all attempts to integrate data collection with analysis and presentation services and thus provide researchers with a unified framework for repository mining. In the same category lies BuCo Reporter which is our approach in the field of Software Repository Mining.

BuCo Reporter is an easy-to-use, extensible standalone application that analyses relevant project data such as commits, committers, source code and bugs, and provides useful reports about the project evolution history. By employing VCS and BTS BuCo extracts the corresponding data from project commits and calculates a series of metrics such as Historical Commit Distribution, Average Lines per Commit, Average Commits over a time period and many more. Concerning the bug report analysis BuCo calculates among others, the Average Bug Correction Time and the Rate of Unresolved Bugs. Overall BuCo provides a unified framework that can be easily extended in order to provide seamless BTS and VCS information retrieval.

The rest of the paper is organized as follows: In Section 2 we introduce the required terminology and analyze the main features of BuCo Reporter. Section 3 contains a detailed overview of the structure and the advantages of the underlying architecture while Section 4 presents representative results from BuCo Reporter. Related work is discussed in Section 5. Finally we conclude in Section 6.

2. BUCO REPORTER

The main reason that stimulated the development of BuCo Reporter was the lack of a clean and simple tool which would enable software practitioners to easily combine information from both Version Control and Bug Tracking systems. BuCo is comprised of three main modules, the Version Control System (VCS) module, the Bug Tracking System (BTS) module and the Reporting module. The tool can be downloaded along with detailed information on how to use it from [7].

2.1 Terminology

Next, we define some concepts that are useful for the rest of the paper.

Source Code Tree: The file structure of a project in a repository is represented as a Unix-like file tree.

Commit: The action by a developer to deliver the changes made to the source code. Changes could be modification of existing source code or addition/deletion/renaming of a source file.

Revision: A unique ID that determines a snapshot of the repository. The revision is altered (i.e. stepped up by one) when a commit occurs and can be linked to a set of modified files or to the entire source code tree.

Delta: The difference between two revisions of a file. Whenever a file is modified and the change is committed, the VCS keeps only

the content differences from the previous revision, in other words, VCS do not store all the contents of each new revision. The main benefit of this approach is the reduced storage required to keep all the revisions for each file.

Log Entry: The information structure that accompanies a commit and contains useful information. The structure of a typical Log Entry is depicted in Figure 1.

Log Entry
Commit Message
Revision
Committer's username
Commit date
List of affected files
Commit type

Figure 1. Structure of a typical Log Entry.

Bug: A report about a defect found in a software module that can be registered either by users or developers. A bug report is comprised of several fields of which the most significant are shown in Figure 2.

Bug Report	
ID	Is Open
Assignee	Last Changed Time
Bug Classification	Operating System
Buggy Component	Priority
Date	Product
Bug Reporter	Remaining Time
Deadline	Summary
Is confirmed	Target Milestone

Figure 2. Structure of a typical Bug Entry.

2.2 Features

Since each of the BuCo modules has a distinct purpose, the framework's features will be listed for each module separately. The **VCS module** of BuCo provides the following features:

- Connection to a source code repository and extraction of information related to software projects such as messages, affected file paths, committer and date of commit and also the revision number.
- Extraction of Delta -the difference of contents between revisions- for any two given revisions.
- Mapping of each project in the remote repository to a local virtual repository.
- The ability to update each local virtual repository to mirror the latest changes in the remote repository.
- Easy access to local repository in order to inspect the stored data.
- Representation of each local repository as a tree for easy manipulation.
- Definition of filters on a project tree in order to isolate a specific software component.
- Extraction of the source code from the local virtual repository to a local folder for easy processing by other programs such as IDEs and UML modeling tools.

The **BTS module** of BuCo provides the following features:

- Connection to a bug tracking repository and extraction of information related to bugs such as id, reporter, assignee, resolution, project/component, date created, date resolved and other.
- Definition of a unified method to create queries based on bug fields in order to extract bugs that conform to given query parameters. For example: download all bugs regarding project "Firefox", with status=fixed prior to 05/16/2010.
- Ability to limit the queried bug information (selection of bug fields) that will be downloaded in order to discard unnecessary bug fields.
- Local bug storage and easy retrieval based on the aforementioned query mechanism.
- Ability to download bug attachments and to decide which of those are code patches based on the Unified Diff Format [9].
- Unified bug representation structure for increased interoperability among different Bug Tracking systems.

The **Reporter module** of BuCo is divided in two sub-modules. The first is responsible for the generation of the source code related metrics, while the second generates reports for bug-related metrics. Each metric is calculated not only for a single revision but for a specified time period and is presented in a chart that depicts the evolution of the metric through the specified period. Table 1 describes all implemented source code metrics.

Table 1. Source code related reports

Name	Description
Commit Impact	The number of lines added/removed in a time period
Average Lines per Commit	Average number of lines modified by each commit
Commit Distribution	Distribution of source code changes over a time period
Commit Type	Type of the commit: Modification, Deletion, Addition, Renaming
Files Modified Per Commit	Average number of files that are affected by each commit
Average Commits over Time	Average number of commits for a time period
File Type	Number of files edited for different type files, over time
Static Code Metrics	Displays all the source code metrics as they are implemented by the Source Monitor tool [8]

The implemented bug-related metrics that are depicted graphically by means of evolution charts are described in Table 2. All metrics can be grouped by field.

Table 2. Bug related reports

Name	Description
Unresolved Defects	Number of unresolved or non-corrected bugs
Resolved Defects	Number of resolved/corrected bugs
Average Correction Time	The average time needed for a bug to be corrected.
Average Age of Unresolved Defects	The average age of unresolved bugs
Escaped Defects Found	Number of bugs found after the release of a specific version
Defect Resolving Efficiency	Ratio of resolved to unresolved bugs
Defect Find Rate	Number of new bugs found over a period of time
Percentage of Documented Defects	The percentage of bugs that contain a patch attachment

Nevertheless, it should be stressed that the aforementioned features constitute an indicative and preliminary set of operations and data that could be of interest regarding a software or bug repository. Given that BuCo has been designed as an extensible framework, as it will be analyzed in the next subsection, its set of features can be enhanced or modified according to the developer needs.

3. ARCHITECTURE

BuCo has been implemented in the Java programming language and it counts over 20 KLOC in more than 350 classes. The main libraries that BuCo utilizes are SVNKit [19] for the communication with subversion repositories, JIRA SOAP client [13] for connecting to JIRA bug tracking systems, Apache POI [2] for the exporting of results in Microsoft Excel format, Apache Xml-Rpc [3] for communication with Bugzilla [6] and other rpc-enabled systems and JFreeChart [14] for the creation of graphical reports.

One of the main concerns during the development of BuCo was to achieve a high level of interoperability and therefore the whole application is based on an extensible framework which enables the seamless integration of new module implementations. As shown in Figure 3, BuCo is composed of three main modules; Report, VCS Service and BTS Service. The other two modules assemble the whole framework.

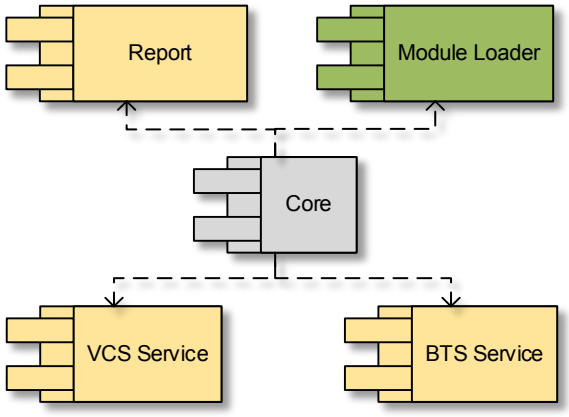

Figure 3. Main modules of BuCo

3.1 Version Control System (VCS) Module

This is a set of Service Provider Interfaces (SPIs) [5] that define the interaction between BuCo and Version Control Systems. The main entry point of this module is an interface called `ControlVersioningSystem`. Any client that will provide communication services with a VCS (e.g. Subversion, Git, Mercurial) must implement this interface. A default implementation for interaction with subversion (SVN) [18] repositories is provided. To overcome the specificities of different version control systems, we have defined a common internal structure for local data storage regardless the underlying repository system. There is a set of other interfaces such as `HistoryClient` and `Delta` which must be implemented in order to convert the data coming from the remote version control system to our common internal structure.

3.2 Bug Tracking System (BTS) Module

This module provides a common interface (SPI) that defines the communication between BuCo and Bug Tracking Systems. The main interface that each client must implement is the `BugTrackingSystem`. This interface defines abstract operations that facilitate communication between BuCo and the given BTS. Currently BuCo provides communication with JIRA [13] and Bugzilla [6] Bug Tracking Systems.

One of the main problems that we had to deal with, is the diversity of bug entries among different Bug Tracking Systems, therefore we used a set of predefined fields which are the most common among the well-known BTSs (Bugzilla, JIRA, Trac). The class that models the bug structure (`Bug`) does not contain a set of fixed fields (one for each of the corresponding BTS fields), instead it maps field names (enumeration `BugField`) to their values. Therefore, the number of fields maintained for each bug is defined by the underlying BTS.

Bug Queries. We have defined a unified query mechanism that is BTS independent, in other words, a BuCo subsystem can query bugs coming from Bugzilla in the same way it would do for bugs coming from JIRA.

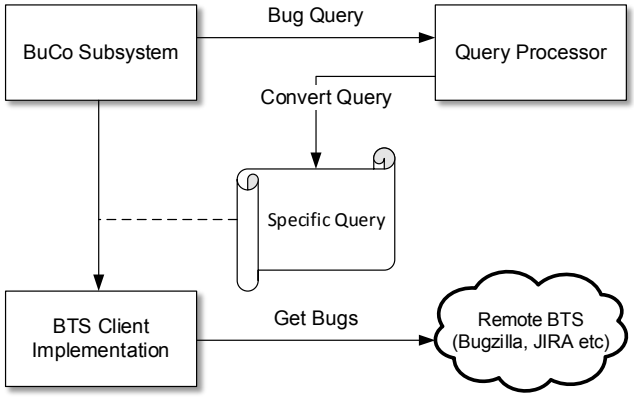

Figure 4. Bug query mechanism

As shown in Figure 4, each BuCo subsystem uses a component called Query Processor (`ParameterProcessor`) to create a specific query for the underlying BTS system. The `ParameterProcessor` interface must be implemented by each BTS client.

Bug queries are based on parameters where each parameter is usually assigned to a bug field. For example, to get all resolved

and fixed bugs for product “Axis” we define the following list of parameters:

```
Parameter {name: product, type: STRING, value: ”Axis”}
Parameter {name: resolution, type: STRING, value: ”RESOLVED”}
Parameter {name: status, type: STRING, value: ”FIXED”}
```

This list is passed to the implementation of the `ParameterProcessor` provided by the BTS client, which in turn will create a specific query that will be used to query bugs from the Bug Tracking system.

3.3 Reporter

This module is a set of interfaces that can be implemented in order to produce various data analyses and reports. The main entry points are two interfaces `ReporterSuite` and `Reporter` where each reporter suite is composed of many reporters. A client must implement these two interfaces in order to add reporting features. BuCo already provides implementation for a plethora of bug and source code metrics and analyses.

3.4 Maintainability Concerns

Particular effort has been put in making BuCo an easily extensible platform. As shown in Figure 5, we have introduced an additional module (Module Loader) that is responsible for the assembling and loading of all other components. The Module Loader provides a linking mechanism that simplifies the plugging of new implementations of the corresponding interfaces.

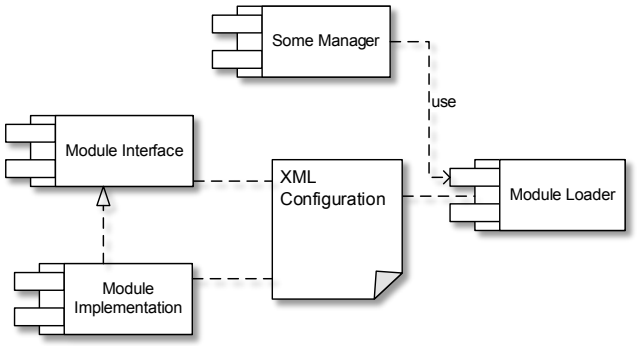

Figure 5. Module Loading Mechanism

The Loader has a very simple mechanism which is based on the contents of an XML configuration file. It demands from modules to keep separately their interfaces from their implementations. By using XML tags the module describes the implemented interfaces and assign them to the corresponding implementations. The Loader then reads the configuration file and loads by demand the respective implementation for a given module interface.

The BuCo Reporter application is structured by complying with the Model-View-Controller (MVC) pattern. For each module there is a corresponding controller and each controller is initialized by a manager as shown in Figure 6. This architecture simplifies the addition of new features to the application, that is, in order to add a new feature we can define a new module interface and provide its implementation. Finally for the new module a new controller and a new manager will be needed. In other words, the addition of new feature can be made without changing the current system structure, thus complying with the open-closed principle [16].

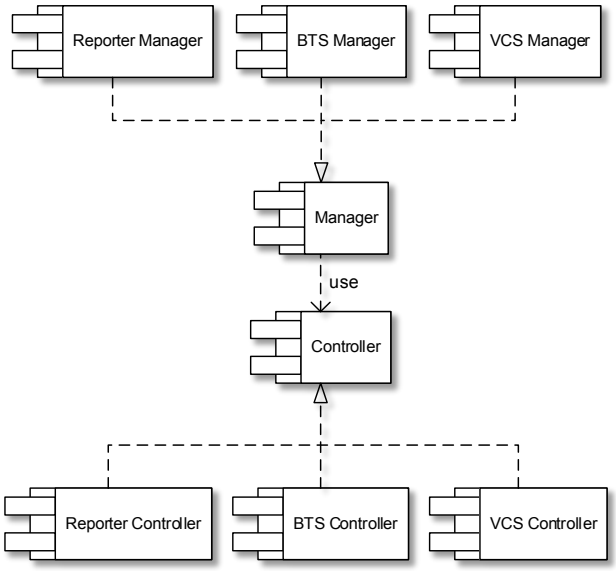

Figure 6. Model View Controller

3.5 Interoperability

After the successful extraction of data from the Version Control and Bug Tracking systems the next process concerns the storage. The selection of storage format and physical medium is critical not only due to the huge amount of data, but also because a high interoperability level is desired. For these reasons we selected an XML format as an internal representation. Figure 7 depicts the representation of a Bug Entry for the project Commons IO.

```
<bug>
 <field name="CLASSIFICATION">IO-351</field>
 <field name="COMPONENT">Utilities</field>
 <field name="CREATION_TIME">23/10/2012</field>
 <field name="CREATOR">wallyqiao</field>
 <field name="ID">12613128</field>
 <field name="LAST_CHANGED_TIME">24/10/2012</field>
 <field name="OPERATING_SYSTEM">Linux/Win</field>
 <field name="PRIORITY">3</field>
 <field name="PRODUCT">IO</field>
 <field name="STATUS">Open</field>
 <field name="SUMMARY">The Tailer keeps closing and re-
 opening file, leads to logs lost. </field>
 <field name="TARGET_MILESTONE">none</field>
 <field name="VERSION">2.4</field>
 <comments> <comments/>
 <attachments> <attachments/>
</bug>
```

Figure 7. XML Representation of a Bug Entry

Apart from bug related data, the source code of each retrieved project is also represented internally in a tree form, where each nested directory contains Delta files. An example is shown in Figure 8 where a project named `projectA` is shown. An external program could reconstruct the structure of the entire project by traversing the tree and merging for each leaf node (file) the deltas which are included in the corresponding folder. The same process can be performed in order to reconstruct a system up to a particular version. This information is fed to the BuCo reporter in order to generate source code related charts and calculate code metrics.

```
/projectA
  /org
    /example
      /Class1.java (folder)
       1.diff (diff in revision 1 is
               essentially the first change)
       5.diff (next diff)
      /Class2.java
       2.diff (first delta)
       5.diff
       6.diff
      /Class3.java
       1.diff
       3.diff
       5.diff
```

Figure 8. Internal representation of Source Code

4. RESULTS

As an illustration of BuCo Reporter capabilities we have analyzed several characteristics of the Commons IO project [1]. BuCo retrieved all the project's bugs from the corresponding BTS and the log entries along with revision contents from the CVS. Then, using the Reporter a set of results has been generated, for each characteristic of the project. The extracted results, as shown in Figures 9 - 19 are indicative of the BuCo Reporter capabilities.

Figure 9 displays the unresolved bugs grouped by priority (priority numbers are assigned by project manager to a String such as MINOR, MAJOR, CRITICAL) while Figure 10 shows the unresolved bugs over time. As it can be observed in general the number of unresolved issues increases over time (as it is reasonable to expect since they accumulate), however there are certain dates when the introduction of a new version abruptly reduced the number of unresolved defects. Figure 11 displays the average time in days needed for a bug to be fixed. The results are grouped by component. We notice that bugs in Filters component usually need more days to be fixed and those in Streams/Writers component need fewer.

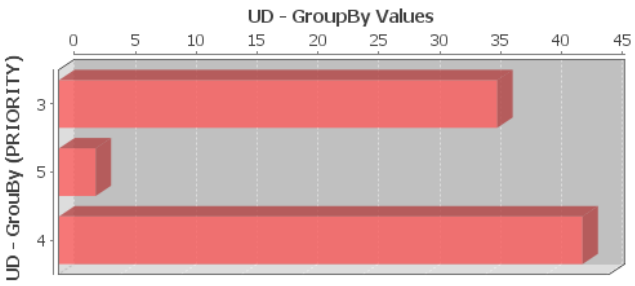

Figure 9. Unresolved defects grouped by priority

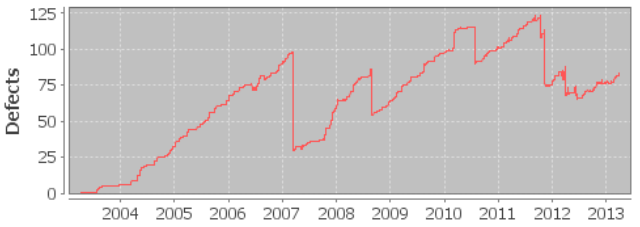

Figure 10. Unresolved defects over time

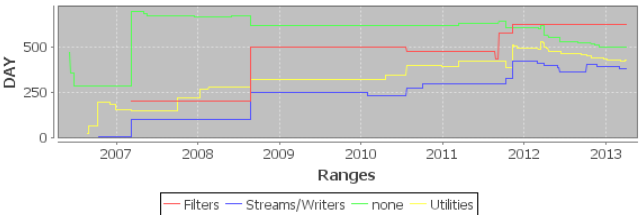

Figure 11. Average time to fix grouped by component

Figure 12 displays the total number of bugs found in each of the components and those that do not belong to any component (according to the information provided by the developer who reported the bug). Figure 13 depicts the cumulative number of bugs over time. A steep line slope means that the number of bugs in that period was increased heavily (e.g. the top line). On the contrary, a moderate or low slope indicates a relatively low bug identification rate.

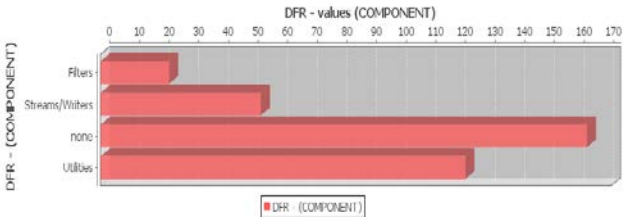

Figure 12. Total defects found in each component

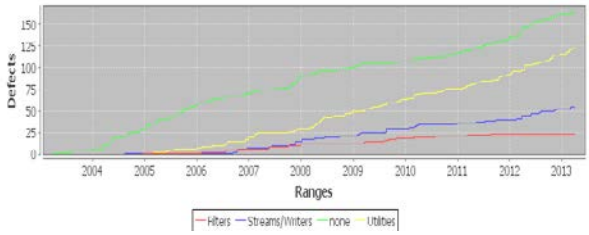

Figure 13. Cumulative defects found, grouped by component

Using BuCo reporter we analyzed information gathered from the project's repository and found that, the highest commit impact (lines added/removed) occurred during February 2012 when 18.848 lines of code were added (Figure 14). On the other hand the maximum average lines of code added/modified per commit, as shown in Figure 15, appeared in October 2012 where on average, 648 lines were added/modified during each commit. As Figure 16 depicts, the month with the highest number commits was September 2010 (98 commits) followed by February 2012 (79 commits). Another valuable metric is the average number of files modified per commit, shown in Figure 17. According to the

results the maximum average number of files modified per commit occurred in April (13 files) and in December (9.3 files) of 2008.

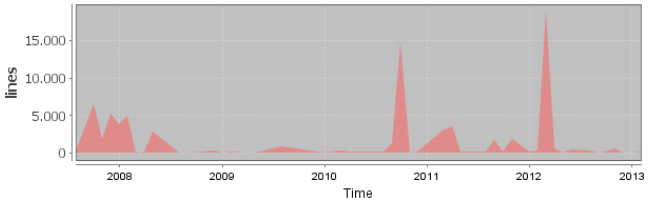

Figure 14. Commit impact in lines of code

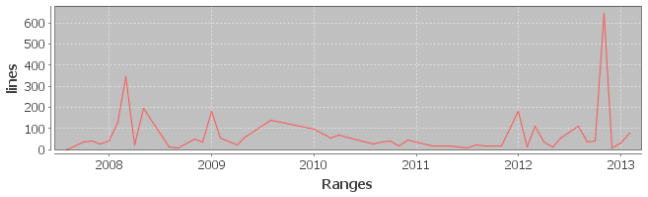

Figure 15. Average lines added/modified per commit

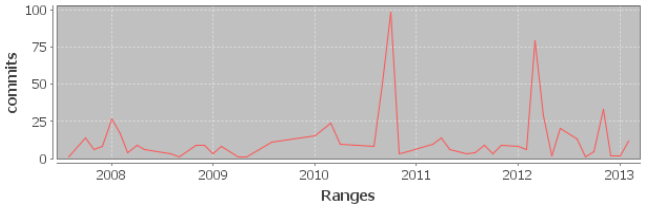

Figure 16. Commits distribution over time

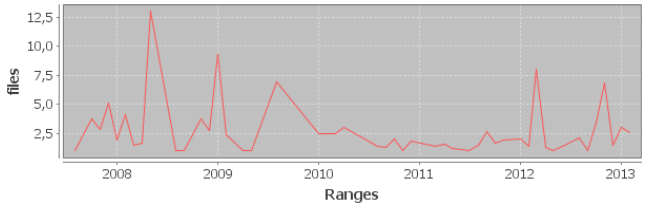

Figure 17. Average number of files modified per commit

Beyond the generation of charts illustrating the evolution of selected metrics during a project's history, BuCo reporter can facilitate inferential statistics by depicting graphically the correlation between selected measures. As an example, with the help of BuCo reporter we examined the average complexity of source files against the number of commits. Generally speaking, the higher the complexity of a file is, the lower the number of modifications (commits) is expected. As Figure 18 reveals, most of the files with high complexity (above 4) have low to medium number of modifications (under 20 commits). However this cannot be regarded as a strict rule. For example, we found that the file `XmlStreamReader.java` has a relatively high complexity (4.22) and has been frequently modified. Another file that falls in the same category, is `FileUtils.java` with 42 modifications and an average complexity of 3.51.

In project Commons IO file `FileUtils.java` is the main entry of the API and therefore it is reasonable to exhibit a large number of modifications. On the other hand file `XmlStreamReader.java` required further examination by the maintenance team to find out the cause for the frequent changes.

Finally, in Figure 19 an overview of the average complexity over time of the entire project is presented. As we can observe the average complexity remains rather unchanged over time (this is evident from the small range of the changes)

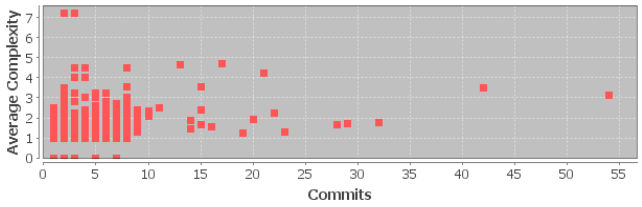

Figure 18. Average complexity vs. Commits

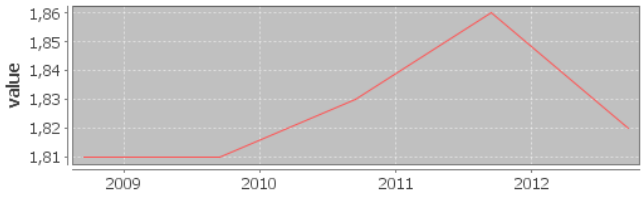

Figure 19. Average complexity over time

5. RELATED WORK

Although the field of Mining Software Repositories is rather mature there is still a shortage of general frameworks that will integrate and ease the mining of the repositories, the analysis of source code, commits and bugs and the reporting capabilities. The main obstacle that prevents the creation of such tools (also known as automated software engineering tools) is the difficulty in scaling. That is the weakness to handle the enormous amount of data that modern repositories contain [17], however in the last years some approaches did emerge. From our personal experience a further weakness is the difficulty in setting up, configuring and using these tools.

Gousios and Spinellis [11] introduced the "Alitheia Core", a platform that integrates data collection (repository mining) and analysis services such as source code and software development metric calculation. It follows a three-tier service oriented architecture where the first tier is responsible for mirroring the data from the remote repositories, the second is the system core where the analysis takes place and the third is the presentation tier. It supports two presentation layers, through a web interface and through an Eclipse IDE plug-in. Alitheia Core can be easily extended by developing new metric calculation plug-ins as OSGi services.

Another approach is the one made by Dyer et al. [10] who developed BOA, a language and framework for the analysis of large-scale software repositories. BOA framework mirrors the repository to a local cluster and provides monthly snapshots of the data. Next, the user can exploit the BOA domain specific language to extract useful information for the local data cluster. All the above operations are carried out through a web interface.

Kenyon is a tool for Automated Software Engineering developed by Bevan et al. [4]. Its main feature is the ability to facilitate the creation of new evolution analysis tools as well as the data sharing among them. Kenyon provides a scalable infrastructure that can assist third party tools in software repository mining, fact extraction and database management.

6. CONCLUSIONS

Contemporary software repositories offer a vast amount of information regarding the past evolution of software projects that can be extremely valuable to software practitioners and researchers. However, the plethora of available information (source code, bug reports, mailing lists), the availability of this data for numerous versions and the distribution in multiple physical locations hinders the access and retrieval of this knowledge. To this end, we introduced BuCo Reporter, a framework for mining source code and bug repositories which is capable of providing various reports regarding the evolution of software projects. Its key advantage is its extensible and easily maintainable modular architecture upon which the framework has been built. Moreover, compared to other existing platforms, BuCo Reporter allows the retrieval of information, the execution of queries on the extracted data and the generation of charts in a few, simple steps without requiring any background knowledge from the end user.

7. ACKNOWLEDGEMENTS

This research has been co-financed by the European Union (European Social Fund – ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thalis – Athens University of Economics and Business - SOFTWARE ENGINEERING RESEARCH PLATFORM.

8. REFERENCES

[1] Apache Commons IO, http://commons.apache.org/proper/commons-io, March 2013.

[2] Apache POI - the Java API for Microsoft Documents, http://poi.apache.org , March 2013.

[3] Apache XML-RPC, http://ws.apache.org/xmlrpc , March 2013.

[4] Bevan, J., Whitehead, E. J., Kim, Jr., S. and Godfrey, M. 2005. Facilitating software evolution research with Kenyon. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (Lisbon, Portugal, September 05-09). 177–186.

[5] Bloch J. 2008. Effective Java. Addison-Wesley.

[6] Bugzilla, http://www.bugzilla.org, March 2013.

[7] BuCo Reporter, http://java.uom.gr/buco, March 2013.

[8] Campwood Software – Source Monitor Version 3.3, http://www.campwoodsw.com/sourcemonitor.html , March 2013.

[9] Diffutils - GNU Project - Free Software Foundation, http://www.gnu.org/software/diffutils, March 2013.

[10] Dyer, R., Nguyen, H., Rajan. H. and Nguyen, T. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *proceedings of the 35th International Conference on Software Engineering* (San Francisco, CA, May 22, 2013).

[11] Gousios, G. and Spinellis, D. 2009. Alitheia Core: An extensible software quality monitoring platform. In *Proceedings of the 31st International Conference on Software Engineering — Formal Research Demonstrations Track*, (Vancouver, Canada, May 16-24, 2009) 579–582.

[12] Hora, A., Anquetil, N., Ducasse, S., Bhatti, M., Couto, C., Valente., M. T. and Martins, J. 2012. BugMaps: A Tool for the Visual Exploration and Analysis of Bugs. In *proceedings of the 16th European Conference on Software Maintenance and Reengineering* (Szeged, Hungary, March 27-30, 2012).

[13] Issue & Project Tracking Software | Atlassian JIRA, http://www.atlassian.com/software/jira, March 2013.

[14] JFreeChart, http://www.jfree.org/jfreechart, March 2013.

[15] Kagdi, H., Collard, M. L. and Maletic, J. I. 2007. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*. 19, 2, 77–131.

[16] Martin, R.C., 2003. Agile Software Development: Principles, Patterns and Practices. Prentice Hall.

[17] Shang, W., Adams, B. and Hassan. A. E. 2010. An experience report on scaling tools for mining software repositories using MapReduce. In *Proceedings of the IEEE/ACM international conference on automated software engineering*. (New York, NY, USA), 275-284.

[18] Subversion - Tigris.org, http://subversion.tigris.org , March 2013.

[19] SVNKit :: Subversion for Java, http://svnkit.com , March 2013.