Alitheia Core: An extensible software quality monitoring platform

Georgios Gousios, Diomidis Spinellis
Department of Management Science and Technology
Athens University of Economics and Business
{gousiosg, dds}@aueb.gr

Abstract

Research in the fields of software quality and maintainability requires the analysis of large quantities of data, which often originate from open source software projects. Pre-processing data, calculating metrics, and synthesizing composite results from a large corpus of project artefacts is a tedious and error prone task lacking direct scientific value. The Alitheia Core tool is an extensible platform for software quality analysis that is designed specifically to facilitate software engineering research on large and diverse data sources, by integrating data collection and preprocessing phases with an array of analysis services, and presenting the researcher with an easy to use extension mechanism. The system has been used to process several projects successfully, forming the basis of an emerging ecosystem of quality analysis tools.

1. Introduction

A well-known conjecture in software engineering is that product quality characteristics are correlated, or result from, good software development practices, and thus source code metrics provide useful data for the assessment of its quality. Uniquely, open source software (OSS) allows us to examine a system's actual code and perform white box testing and analysis [9]. In addition, in most open source projects, we can access their version control system, mailing lists, and bug management databases and thereby obtain information about the process behind the product. However, deep analysis of those software artefacts is neither simple nor cheap in terms of computing resources. Many successful OSS projects have a lifespan in excess of a decade and therefore have amassed several GBs worth of valuable product and process data. In this demonstration paper, we present Alitheia Core, an extensible platform designed specifically for performing large-scale software quality evaluation studies.

2. Related Work

The continuous metric monitoring approach towards achieving software quality is not new. The first systems that automate metric collection emerged almost immediately after revision control systems and bug management databases were integrated in the development processes. Early efforts concentrated on small scale, centralized teams and product metrics (e.g. [2, 7]), usually to support quality models or management targets set using the Goal-Question-Metric approach [1]. Alitheia Core is able to process more data sources and while it does feature a quality model implementation, it is not tied to it, enabling the user to combine arbitrary software metrics towards a custom definition of quality.

The Hackystat [6] project was one of the first efforts to consider both process and product metrics in its evaluation process. Hackystat is based upon a push model for retrieving data as it requires tools (*sensors*) to be installed at the developer's site. The sensors monitor the developer's use of tools and updates a centralized server. Alitheia Core is similar to Hackystat in that it can process product and process data; it improves over Hackystat as it does not require any changes to the developer's toolchest or the project's configuration while it can also process soft data such as mailing lists.

Finally, a number of projects have considered the analysis of OSS development data for research purposes. Flossmole [5] was first to provide a database of preprocessed data from the Sourceforge OSS development site.

3. Platform Requirements and Architecture

The original aim of the SQO-OSS project was to develop a web site of publicly accessible measurements for OSS software projects.

The key requirement for a system like the Alitheia Core is efficiency; the system must be able to process large data volumes with algorithms that are often CPU-intensive. For the projects we examined before designing the system, the average size of their repositories was in the order of 5000 revisions; a few large projects, like FreeBSD, had more than one hundred thousand revisions. Each revision of the project can have thousands of live files (the Linux kernel has about eight thousand), the majority of which are source code files. A rough calculation for an average 20KB file size shows that the system would read a gigabyte of data just to load the processed file contents into memory. Even this simple operation is prohibitively expensive to do over the Internet as it would introduce large latencies and would hurt the performance of the project hosting servers. Moreover, if a metric requires an average 10s of processing time per revision, processing the average project would take 14 hours on a single CPU computer. After some experimentation with such back of the envelope calculations and a system prototype, it becomes apparent that a naive approach of getting the data from the project's repository on request and processing them would not scale; a more sophisticated solution that would combine project local data mirroring and multi-core processing and possibly clustering is required.

On the other hand, the OSS development tools landscape is very diverse; currently, in use are at least five major revision control systems (including CVS, Subversion, GIT, Mercurial and Bazaar), five bug tracking systems (e.g. Bugzilla, Mantis, Jira, GNATS) and literally hundreds of configurations of mailing list services and Wiki documentation systems. Also, depending on the nature of the evaluated asset, metrics can work with source code file contents, source code repository metadata, bug reports, or arbitrary combinations thereof. Collecting and processing hundreds of gigabytes of data from such a diverse set of data sources requires careful consideration of the storage formats and the mirroring process. Moreover, the choice of the particular data storage format should not hinder the system's ability to work with other raw data formats as well. In our system, we decided to use the least common denominator of features for our storage formats, and standardised on the Subversion repository format for source code data, the Maildir format for mailing list data and the Bugzilla XML format for bug data.

3.1. Architectural Components

To address the challenges presented above, the Alitheia Core system is based on an extensible, service-oriented architecture presented in Figure 1. To separate the concerns of mirroring and storing the raw data and processing and presenting the results the system was based on a three-tier architecture. The processing core is modeled on a system bus architecture with services that are attached to the bus and are accessed via a service interface. The OSGi component model was selected to base the system's core layer.

The data access layer consists of two basic components:

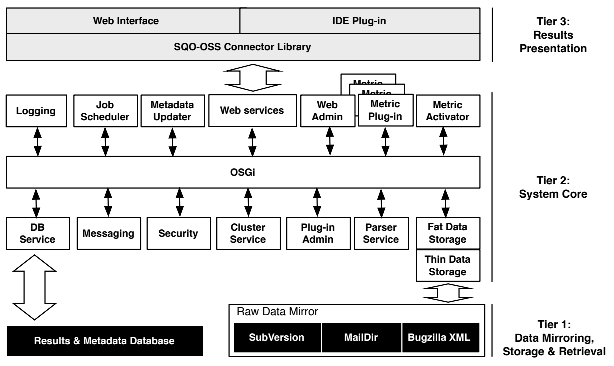

Figure 1. The Alitheia Core system architecture

the database service and the fat/thin data access stack. The database service is central to the system as it serves the triple role of abstracting the underlying data formats by storing metadata, storing metric results, and providing the types used throughout the system to model project resources, such as project versions, project files, mail messages, mail threads, and bugs. It uses an object-relational mapping to eliminate the barrier between runtime types and stored data and has integrated transaction management facilities. Access to raw data is regulated by a plug-in based stack of accessors whose lower layers directly touch and fetch the mirrored data, while the higher parts provide caching and combined raw data and processed metadata access.

Upon project registration, a preprocessing phase converts the raw data to internal system representations, which is what the metric plug-ins work with. During the preprocessing phase, the system extracts metadata from the raw data and stores them into the database. Through the metadata database the system can swiftly respond to metric queries relating to the properties of the examined resources, while the original contents of the resources are still available in the raw data stores. Example metadata queries include the live files in a revision, or the authors of the emails participating in a particular thread. The metadata entities are also used by metrics to store and calculate results; for example, if the metadata updater encounters a new revision, it will notify all metrics that calculate their results on whole project checkouts, and, after the result is calculated, it will be stored against the same database object.

The Alitheia Core has been designed from the ground up for performance and scalability. All metric plug-in executions and a significant number of performance critical functions in the core are modeled as jobs. The job scheduler component maintains a configurable size pool of worker threads, and schedules items from its work queue to idle threads. Jobs can have dependencies and priorities to cater for scenarios where a metric plug-in requires the result of another metric plug-in. In most cases, the execution path is lock-free, which enables high scalability and, given the appropriate hardware, very high throughput rates. The processing core is currently being run on 8 and 16 core machines, exhibiting almost linear scalability and full processor utilisation.

Alitheia Core also includes clustering capabilities through the cluster service. The development of the cluster service was based on the observation that the workloads the system processes are usually embarrassingly parallel, since each project's data is handled independently from the others. For long-lived projects however, the initial metadata synchronization is extremely resource intensive. This means that that after the project import phase, the plug-ins could run on another, perhaps less powerful, host if they could have access to the same database. The clustering service guarantees that all metadata updates are performed on a single node. After the metadata update transaction is committed, the metric jobs on any host will see the latest version of the project metadata. In our experimental setup, all metadata updates run on the data mirroring host to allow fast, disk-based access to the original project data. The system does not yet support automatic load balancing or failover.

Finally, the Alitheia Core system includes support for presenting the calculated results and project metadata through the web services component. The web services service acts as a gateway between the core and the various user interfaces, using a SOAP-based communication protocol. At the moment, two user interfaces are provided; a web interface that enables browsing of the processing results on the web,[1] and an Eclipse plug-in that allows developers to see the results of their work through their work environment.

3.2. Metric Plug-ins

The Alitheia core engine can be extended by plug-ins that calculate metrics. Metric plug-ins are OSGi services that implement a common interface and are discoverable using the plug-in administrator service. In practice, all metric plug-ins inherit from an abstract implementation of the plug-in interface and only have to provide implementations of three methods. Moreover, to hide the intricacies of setting up the OSGi class sharing mechanism, our system provides a skeleton plug-in that is already preconfigured to the requirements of the platform. The net result is that with exactly 30 lines of code, a researcher can write a simple source code line counting metric that fetches a file from the repository, counts its lines, stores the result, and returns it upon request.

[1]An example installation can be found online at `http://demo.sqo-oss.org`

Each plug-in is associated with a set of *activation types*. An activation type indicates that a plug-in must be activated in response to a change to a corresponding project asset; it is a database object that maps a resource that has changed in the project's data mirror to an entry in the metadata database. A metric plug-in can define several metrics, which are identified by a unique name (*mnemonic*). Each metric is associated with a scope that specifies the resource this metric is calculated against: files, namespaces, directories, or mailing lists. Metrics can also declare dependencies on other metrics and the system will use this information to adjust the plug-in execution order accordingly through the metric activator service. The system administrator can also specify a set of policies regulating the recalculation frequency for each metric plug-in. Metric results are stored in the system database either in predefined tables or in plug-in specific tables. The retrieval of results is bound to the resource state the metric was calculated upon.

A plug-in can use a wealth of services from the core to obtain project related data using simple method calls. For example, a plug-in:

- can request a checkout for a specific project revision or opt for a faster in-memory representation of the file tree and load the content of the required files on demand,
- can ask the system to return a list of files that match a given pattern, for example all Java files across project versions or across projects,
- can obtain a language-agnostic Abstract Syntax Tree (AST)-like representation of the parsed source code for a specific file or revision (currently only for Java source code),
- can request a list of all threads a specific email has been sent to and then navigate from the returned objects to the parent threads or to the mailing lists,
- can get all actions performed by a single developer across all project data sources, and
- can request for a measurement calculated by another plug-in. The system will automatically invoke the other plug-in if the requested measurement cannot be found in the database.

We have already developed a number of metric plug-ins; the most important are listed in Table 3.2. To judge the magnitude and contribution of the developed infrastructure note that the sum of the lines of code for all plug-ins is less than 15% of the lines of code of the Alitheia Core.

Metric	Description	Metrics	LoC
Size	Calculates various project size measurements, such as number of files and lines for various types of source files.	11	642
Module	Aggregates size metrics per source code directory.	3	417
Code structure	Parses source code to a language neutral intermediate representation and evaluates structure metrics, such as the Chidamber and Kemerer metric suite [3], on the intermediate representation.	15	958
Contribution	Analyzes repository, mailing list and bug database developer activity and extracts a measurement of the developer contribution to the development process [4].	1	1451
Multigrep	Applies a regular expression to source code files and reports the matches as a measurement. The applied regular expression is configurable at run-time.	Configurable	346
Testability	Identifies and counts testing cases for common unit testing frameworks.	1	561
Quality	A custom quality model implementation that aggregates the results of various structure, size and process metrics into an ordinal scale evaluation [8].	1	2408

Table 1. List of currently implemented metrics.

4. Conclusions

Analysing and evaluating software development process and source code characteristics is an important step towards achieving software product quality. The Alitheia Core is a platform modeled around a pluggable, extensible architecture that enables it to incorporate various types of data sources and be accessible through various user interfaces.

Future work on the platform will include expansion of the data accessors plug-ins to include support for other source code management systems and a web service that will allow external plug-in submissions to be run against the pre-processed data currently hosted on our servers.

The full source code for the Alitheia Core and the plug-ins can be found at `http://www.sqo-oss.org`.

Acknowledgements

This work was partially funded by the European Community's Sixth Framework Programme under the contract IST-2005-033331 "Software Quality Observatory for Open Source Software (SQO-OSS)". Project contributors include the Aristotle University of Thessaloniki, Prosyst Software GmbH, Sirius plc, Klarälvdalens Datakonsult AB and members from the KDE project community.

References

[1] V. Basili and D. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, 10(3):728–738, Nov 1984.

[2] V.R. Basili and H.D. Rombach. The tame project: towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, 14(6):758–773, June 1998.

[3] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[4] Georgios Gousios, Eirini Kalliamvakou, and Diomidis Spinellis. Measuring developer contribution from software repository data. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 129–132, New York, NY, USA, 2008. ACM.

[5] J. Howison, M. Conklin, and K Crowston. Flossmole: A collaborative repository for floss research data and analyses. *International Journal of Information Technology and Web Engineering*, 1(3):17–26, 2006.

[6] P. M. Johnson, M. G. Paulding H. Kou, Q. Zhang, A. Kagawa, and T. Yamashita. Improving software development management through software project telemetry. *IEEE Software*, Aug 2005.

[7] Seija Komi-Sirviö, Päivi Parviainen, and Jussi Ronkainen. Measurement automation: Methodological background and practical solutions-a multiple case study. *IEEE International Symposium on Software Metrics*, page 306, 2001.

[8] Ioannis Samoladas, Georgios Gousios, Diomidis Spinellis, and Ioannis Stamelos. The SQO-OSS quality model: Measurement based open source software evaluation. In Ernesto Damiani and Giancarlo Succi, editors, *Open Source Development, Communities and Quality — OSS 2008: 4th International Conference on Open Source Systems*, pages 237–248, Boston, September 2008. Springer.

[9] Diomidis Spinellis. *Code Quality: The Open Source Perspective*. Addison-Wesley, Boston, MA, 2006.