A Platform for Software Engineering Research

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

Abstract

Research in the fields of software quality, maintainability and evolution requires the analysis of large quantities of data, which often originate from open source software projects. Collecting and preprocessing data, calculating metrics, and synthesizing composite results from a large corpus of project artifacts 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. Alitheia Core aims to be the basis of an ecosystem of shared tools and research data that will enable researchers to focus on their research questions at hand, rather than spend time on re-implementing analysis tools.

In this paper, we present the Alitheia Core platform in detail and demonstrate its usefulness in mining software repositories by guiding the reader through the steps required to execute a simple experiment.

1. Introduction

A well-known conjecture in software engineering is that a product's quality is related to various product and process metrics. Open source software (OSS) allows any researcher to examine a system's actual code and perform white box testing and analysis [1]. In addition, in most open source projects, researchers 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 artifacts 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 paper, we present Alitheia Core, an extensible platform designed specifically for performing large-scale software engineering and repository mining studies. Alitheia Core is the first platform to offer an extensible, integrated representation of software engineering data and the first to automate and parallelize the execution of custom experiments. Using Alitheia Core, researchers can design and execute quantitative and exploratory studies on empirical data without having to deal with low-level tasks such as pre-formatting data or updating their datasets when new data from projects become available.

The contributions of this work are the following.

- A discussion of the importance of shared experimental infrastructures for the MSR community.
- A presentation of an architecture for efficiently calculating and combining product and process metrics from large data sources.
- The introduction of an integrated software analysis platform that can form the basis of an emerging ecosystem of product and process analysis tools along with a demonstration of its effectiveness.

2. Conducting research with software repositories

The main problem that researchers face when working with software repositories is that it is difficult to setup experiments in a way that can be extended or replicated. Despite the fact that data is free and in theory easy to obtain, there are several hurdles that researchers must overcome in order to experiment with large numbers of projects.

To start with, 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), four bug tracking systems (Bugzilla, SourceForge tracker, Jira, GNATS) and literally hundreds of configurations of mailing list services and Wiki documentation systems. Also, depending on the nature of the evaluated asset, experiments 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 researcher's ability to work with other raw data formats as well, as it is common

for projects that are interesting to study to use custom tools to manage their development.

On the other hand there is the issue of efficiency; in large experiments a researcher must be able to apply on large data volumes algorithms that are often CPU-intensive. An average size for a small to medium project is in the order of 5000 revisions; a few large projects, like FreeBSD, have more than one hundred thousand revisions. Each revision of the project can have thousands of live files (the Linux kernel has about 25000), 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 online 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, it becomes apparent that a naive approach of getting the data from the project's repository on request and processing does not scale; a more sophisticated solution that would combine project local data mirroring and parallel processing and possibly clustering is required.

Furthermore, there is clear a lack of standardization in experiment setups. In other fields of computer science, researchers use predefined data sets (e.g. the ACM KDD cup datasets for data mining research [2]) or software platforms (e.g. JikesRVM in the field of virtual machines [3]) for developing and executing experiments, which leads to experiments that are easy to replicate and also to research results that are comparable and that can be extended by further research. The repository mining community has not yet settled on any standards, neither for rigorous evaluation of hypotheses using common experimental data nor for shared tools for experimentation. In our view, this leads to fragmentation of research focus and, more significantly, to duplication of effort spent on tool development.

As a consequence of the above, many research studies on OSS (including our own, which motivated this work [4], [5]) are one-off shots. There is a large number of studies that investigate a research hypothesis by conducting experiments on small number of projects and a very limited number of exploratory studies that examine properties across projects, while, to the best of our knowledge, there is no independent replication of corresponding empirical research.

These problems have been identified early on and solutions have been proposed [6], albeit to no effect. Recently, there has been a wave of calls for the analysis of large bodies of OSS, the sharing of research results and standardization of research infrastructures [7]. To this end, other researchers propose common repositories for sharing research data [8] or even using e-Science infrastructures (the Grid) for large scale collaborative studies as in the case of physics or medicine [9]. We believe that a shared repository of tools or research results is a good step forward but in the long term would not be sufficient for the MSR community, given the wealth of available data sources and the fact that a lot of tools are already freely available. In our view, there is a clear need for standardized, open experimentation infrastructures.

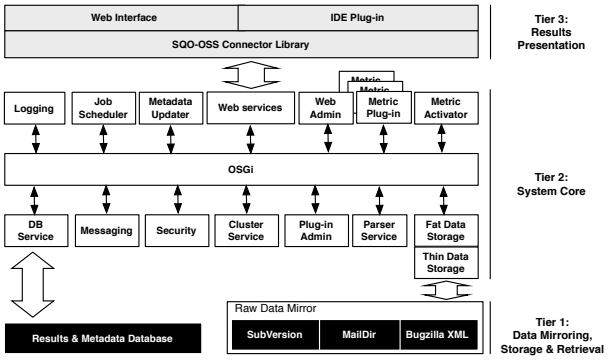

Figure 1. The Alitheia Core architecture

Ideally, such an infrastructure should offer:

- Structured representation of repository data
- Extensibility hooks to allow a researcher to easily write custom experimental tools
- Support for experiment automation

Moreover, the infrastructure should support the sharing of research data through open formats over easily accessible service interfaces and the en masse processing of data.

3. The Alitheia Core platform

Based on the principles presented above, we designed and implemented the Alitheia[1] Core system, an extensible platform targeted to large-scale software quality, evolution and repository mining studies. An overview of the platform's architecture can be seen in Figure 1. To separate the concerns of mirroring and storing the raw data and processing and presenting the results, the system is designed around a three-tier architecture, with separate data acquisition, data processing and presentation layers.

The work presented in this section was first outlined in [10] and various bits of it are described in greater depth in [11]. In this paper, we introduce several system aspects not presented in earlier work, namely the metadata schema, the update process and the system's extension points while also putting more emphasis on how this data can be used by plug-ins.

1. $\alpha\lambda\acute{\eta}\theta\epsilon\iota\alpha$ means "truth" in Greek; the name was given to signify the tool's ability to expose facts about software development.

3.1. The core

The core's role is to provide the services required by metric plug-ins to operate. The core is based on a system bus; various services are attached to the bus and are accessed via a service interface. The OSGi [12] component model was selected to host the system's core layer; consequently the system and its plug-ins are implemented in Java. The choice of the specific implementation technology was made partially because of the wealth of ready-made components being available within the Java application ecosystem and also because OSGi offers a standards based, light weight plug-in platform that would enable the easy sharing of metric plug-ins.

The data access layer consists of two basic components: 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 the corresponding metadata, storing metric results, and providing the types used throughout the system to model project resources. It uses an object-relational mapping to eliminate the barrier between runtime types and stored data [13], 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 the mirrored data (Thin Data Store—TDS), while the higher parts provide caching and combined raw data and processed metadata access (Fat Data Store—FDS).

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 (e.g. metadata updates) are modeled as jobs. 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 utilization. Alitheia Core also includes clustering capabilities through the cluster service. The development of the cluster service was based on the observation that after the initial metadata synchronization, the workloads the system processes are usually embarrassingly parallel. The cluster service restricts access to projects during metadata updates and allows metrics to be run on several nodes in parallel.

The core implements several auxiliary services such as the plug-in manager, the logger for system wide configurable logging, the user account and security manager for regulating access to the system database and finally the messaging service for email based communication with external users. The core also includes a web-based administrative console, selected parts of which are also available through a script-based interface. Finally, under development is a source code parser service that enables metric plug-ins to use a programming language-neutral, XML-based abstract syntax tree representation, specifically designed to hold enough metadata for source code metrics. Currently, the service can fully parse programs written in Java.

3.2. Processing data and storing metadata

The Alitheia Core system uses a database to store metadata about the processed projects. The role of the metadata is not to create replicas of the raw data in the database but rather to provide a set of entities against which plug-ins work with, while also enabling efficient storage and fast retrieval of the project state at any point of the project's lifetime. The original contents of the resources are kept in the raw data stores and can be reached by providing the links stored along with the metadata in the TDS service. The database schema is shown in Figure 2. The schema is composed of four subschemata, one for each project data source and one that deals with metric configuration and measurements storage. Plug-ins can define new entities and thus extend the basic schema arbitrarily. Upon project registration, a preprocessing phase extracts the metadata from the raw data and stores them in the database. There are three metadata updaters in the Alitheia Core system.

The source code metadata updater is the most complex of the three; in its current form, it can process the full set of the Subversion version control system operations. The complexity stems from the fact that Subversion supports multiple operations on a single resource to be recorded in a single transaction (for example, copy-then-delete or delete-then-replace) while also enabling low-overhead copying of resources in the repository. The copying feature is also used to create tags and branches, which means that the updater must be able to differentiate between copying operations according to their purpose. The source code updater stores file metadata differentially; instead of recreating the full file tree in the `ProjectFile` table for each revision, it just stores the changes between revisions. Therefore an entry in the `ProjectFile` table corresponds to a state in the file's lifetime. The mailing list metadata updater works in two steps; the first step consists of simply parsing the email headers and inserting entries in the corresponding tables. During the second step, the updater resolves the parent-child relationships between the recorded emails and organizes them in threads, setting the appropriate thread depth and parent fields in each email entry. The bug metadata updater is straightforward; for each new bug description, it creates entries in the `Bug` table with state information about the processed bug while also keeping the full text for each bug comment. The text will be used to create a full text search engine component in the future.

The final step of the updating process deals with the resolution of developer identities. In the course of a project, developers use several emails to post to mailing lists or to subscribe to bug tracking systems, but usually can be

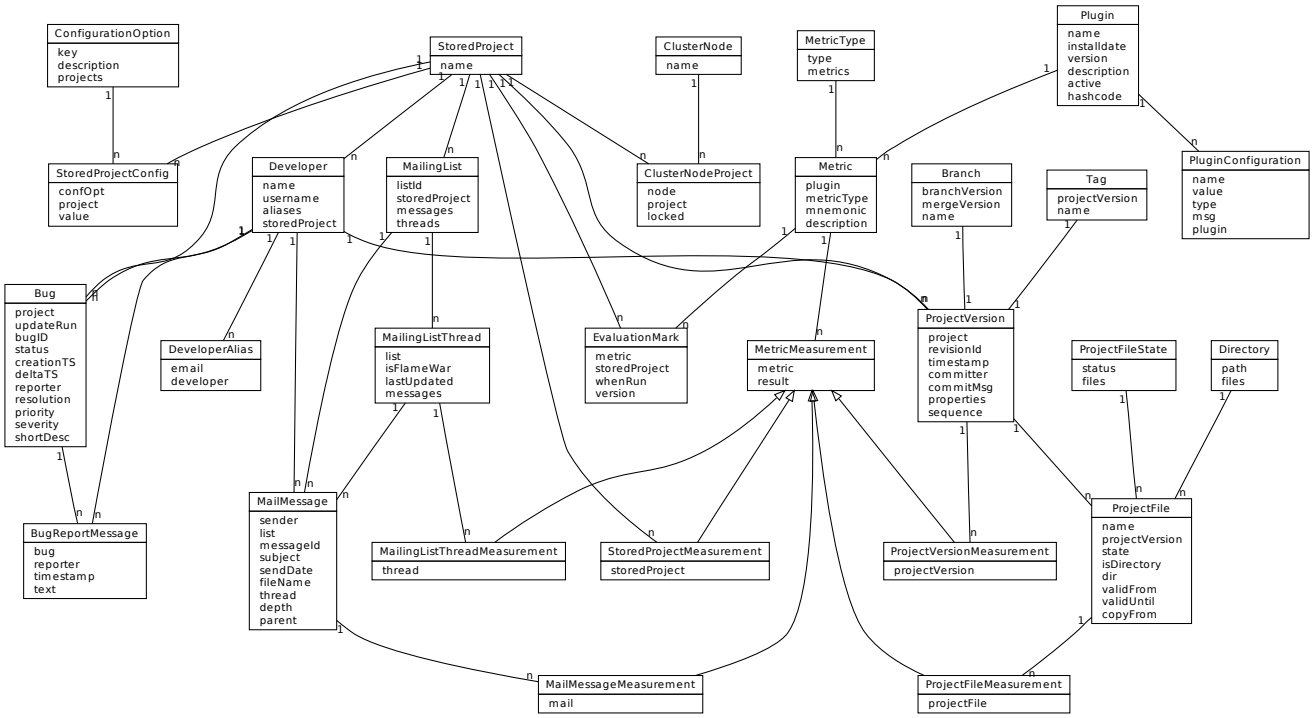

Figure 2. Basic entities in the Alitheia Core database and their relationships

uniquely identified by the name that is attached to an email post or the user name for the project's SCM system. During the updating phase, the Developer table is filled in with all data each updater knows or can infer from the raw data, namely user names, {`real name, email`} tuples and emails for source code management systems, mailing messages and bug reports respectively. The developer resolver uses a set of simple heuristics, the majority of which are described by Robles et al. in [14], to associate developer real names to user names and emails. It improves over earlier work by employing a set of approximate string matching techniques, namely metaphone codes, Levenshtein distances [15] and regular expressions, to improve the matching accuracy and automation. It also takes advantage from the fact that the database can contain several instances of the same developer associated with other projects.

At the end of the update process, the Alitheia Core system is fully equiped with the appropriate metadata to respond to metadata queries much faster than comparable approaches that involve access to the raw data stores.The relative volume of the stored metadata is comparable to that of raw data. Table 1 presents a comparison of the items stored our project mirror with the items stored in the database after processing. The metadata enable complex metrics that need to read the contents of project artifacts to benefit from the database's ability to filter out unwanted items before they reach for the data retrieval subsystem; for example a metric interested in a subset of the project files (e.g. all source code files) can request just those and the system will automatically filter out irrelevant entries, thereby saving the time to fully checkout and then clean up a full project revision. The time savings are significant: on our system, a query to retrieve all source code files for version 135332 of the FreeBSD project executes in two seconds. A comparable approach would entail checking out all files from the repository and then selecting the required ones: on our system this takes 14 minutes for version 135332 of FreeBSD. Furthermore, the metadata entities are also used by metrics to store and calculate results in an incremental fashion; for example, when the source code 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 can be stored against the same database object.

3.3. 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

Table 1. Raw vs processed data sizes

Data	Evince	Orbit2	FreeBSD
Raw Data (items)			
Revisions	3588	2108	190854
Emails (Lists)	1430 (1)	4834 (4)	—
Bugs	5555	246	—
Processed data (DB lines)			
Developer	3862	843	514
DeveloperAlias	4209	1564	0
ProjectVersion	3336	1770	143533
ProjectFile	16875	10879	830792
MailMessage	1430	4833	—
MailingListThread	631	2467	—
Bug	5555	246	—
BugReportMessage	21326	1657	—

plug-ins inherit from an abstract implementation of the plug-in interface and only have to provide implementations of 2 methods for each binding datatype (`run()` and `getResult()`) and the `install()` method to register the plug-in to the system. 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.

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 the corresponding project asset; this is the name of the database entity that models the asset and therefore the metric is activated each time a new entry is added to the database table. 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 set of resources this metric is calculated against: for example files, namespaces, mailing lists or directories. 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,
- 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,
- get all actions performed by a single developer across all project data sources, and
- 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 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 8% of the lines of code of the Alitheia Core.

3.4. Presentation of results

The Alitheia Core system also includes support for presenting the calculated results and project metadata through the web services component. This 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, and an Eclipse plug-in that allows developers to see the results of their work through their work environment.

A new REST-based data access component is currently under design. The new framework is targeted to opening up the preprocessed data residing on the Alitheia Core master servers to the research community using easy to handle data formats (XML and JSON), and will enable various Alitheia Core instances to exchange metadata and plug-in results through a new set of Alitheia Core to Alitheia Core updaters. It will also form the basis of a new web-based interface that will support custom report generation through a web-based query builder.

3.5. Extension Points

Alitheia Core was designed to be highly extensible. The system hides all service implementations behind well-defined interfaces and instantiates at runtime through a set of instantiation protocols, based on configuration parameters or data access URLs, hidden in factory classes. Following is a non-exhaustive list of extension points that are present in the system:

- The data accessor stack can be extended to provide access to project assets not currently in the Alitheia Core system, for example to IRC backlogs and wiki systems or to alternative raw data storage formats (e.g. SCM accessors for GIT or Mercurial repositories).
- The metadata updater stack can be extended to incorporate changes to the data accessors by importing new kinds of data in the database.
- Plug-ins and core services can extend the database schema with custom tables. The system is more fragile

Table 2. List of currently implemented metrics.

Plug-in	Data Source	Description	Activator	Metrics	LoC
Size	Product	Calculates various project size measurements, such as number of files and lines for various types of source files.	`ProjectFile` `ProjectVersion`	11	642
Module	Product	Aggregates size metrics per source code directory.	`ProjectFile` `ProjectVersion`	3	417
Code structure	Product	Parses source code to a language neutral intermediate representation and evaluates structure metrics, such as the Chidamber and Kemerer metric suite [16], on the intermediate representation.	`ProjectVersion`	15	958
Contribution	Process	Analyzes repository, mailing list and bug database developer activity and extracts a measurement of the developer contribution to the development process [17].	`MailingListThread` `ProjectVersion` `Bug`	4	670
Multigrep	Product	Applies a regular expression to source code files and reports the matches as a measurement. The applied regular expression is configurable at run-time.	`ProjectFile`	Configurable	346
Testability	Product	Identifies and counts testing cases for common unit testing frameworks.	`ProjectFile`	1	561
Quality	Both	A custom quality model implementation that aggregates the results of various structure, size and process metrics into an ordinal scale evaluation [18].	`ProjectVersion`	1	2408

to changes to the core schema, but such changes are possible if they are carefully hidden behind method calls inside data access objects. In fact, throughout the course of the project, we did change the source code metadata schema twice without making significant changes to plug-ins or other system parts.

- New administrative actions can be defined in the administration service, to cater for custom installation or clustering scenarios. Administrative actions automatically benefit from the administrative service facilities such as input validation for common data types and provision of the service through programmatic or URL interfaces.

3.6. Project data

The Alitheia Core itself is not concerned with mirroring data from projects; it expects data to be mirrored externally. This choice was made at the beginning of the project to compensate for the large number of different data sources which the system should work with. Several of those already offer means of synchronizing data across sites, for example Subversion offers the `svnsync` tool for mirroring repositories while distributed SCM systems already copy the full repository history on checkout. Mailing lists can be mirrored by configuring the mail delivery subsystem on the mirroring host to store incoming messages to a specified directory and then subscribing to each mailing list. Mailing lists archive mirroring requires custom scripts per project site, although various mailing list archiving web sites (e.g. MARC) offer a unified view over thousands of mailing lists for common projects. Finally, bug tracking tools usually offer means to retrieve individual bug histories through web-based or RPC-based interfaces.

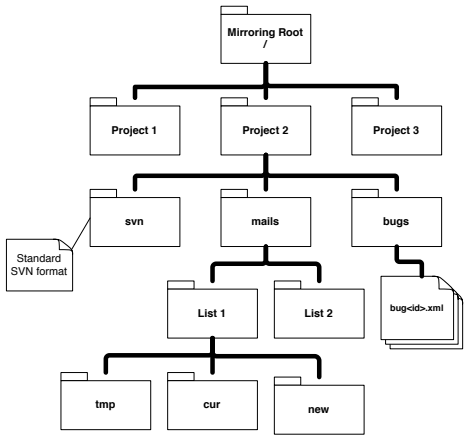

Figure 3. Data mirror layout

To construct a standardized research dataset to use with Alitheia Core, we setup a project mirroring infrastructure, an outline of which can be seen in Figure 3. We are actively mirroring 712 projects and so far we have gathered 4674236 Subversion revisions, 895815 emails in 1381 mailing lists and 341546 bug reports, summing up to 213GB of data. We are actively maintaining and expanding the project mirror, mainly by introducing custom scripts to mirror mailing list data and bug databases from various projects.

4. Case study: Shut Up and Code! [2]

To demonstrate the flexibility of Alitheia Core as a research platform, we present the setup of a small experiment and the plug-in implementation of the data extraction algorithm we employ for the measurements we perform. For the sake of brevity, we skip validation for the various assumptions we make and we do not perform any analysis of the experiment results.

4.1. Research hypothesis

A well known and well studied (both from a social [19] and technical [20] perspective) phenomenon in electronic communications is heated discussions, usually refereed to as "flame wars". Flame wars happen on mailing lists but can also happen on IRC channels or other forms of instant electronic communications. During a flame war the electronic communication etiquette collapses: the topic of discussion stirs away from software development, the rate of message exchanges increases sharply, and the exchanged messages rather than debating the technical aspects of the argument often target directly other participants. Heated discussions do not necessarily signify a flame war; a technical or managerial issue might also elevate the discussion's rate of exchanges. When the point being pondered is insignificant the discussion is often referred to as a "bikeshed argument" [21], after Parkinson's Law of Triviality [22]. Heated discussions are known to affect the community's social structure; what we try to investigate is whether they also affect the project's evolution in the short term by altering the rate at which source code line changes while the discussion is active.

The first step would be to define heated discussions, by examining their characteristics. From observing the evolution of mailing lists, we can assume that intense discussions will take place in the context of a single thread. Also, threads with intense discussions are usually several levels deep and contain more messages than the average thread. Therefore, if we obtain a list of threads sorted by the number of messages and one sorted by the maximum depth, we could assume that the threads that belong in the top quartile of both lists could be classified as intense discussions. Along similar lines, the effect of an intense discussion on a project's short term evolution can be evaluated by comparing the rate of changed source code lines in a period of e.g. one month before the start of the discussion to the rate of line changes immediately after the discussion start and of a period of one week.

4.2. Plug-in implementation

To create a plug-in for Alitheia Core, a researcher must first identify the potential experiment variables and their relationships with the entities that Alitheia Core maintains in its database, probably following the GQM method [23]. Alitheia Core calculates metrics incrementally; for each identified resource change, all metrics that are bound to the resource type are calculated for the changed instances of the resource. The plug-in we describe will need to calculate the depth and the number of emails of a thread, so naturally it will be bound to the `MailingListThread` entity. Each time a `MailingListThread` is updated (e.g. when new emails arrive) the plug-in will be called; to avoid recalculation of threads that have already been evaluated, we will need to cache the result in the database. This is done by defining a new metric.

The calculation of the rate of line changes for the period before the start of the heated discussion and immediately after it is trickier, as Alitheia Core does not currently have any means to store the results of a measurement against a set of entity states. There are two workarounds to this issue:

- store the result against all entity states in the required period, or
- synthesize the result on request, by incrementally combining individual state measurements.

It is usually cheaper, more precise and future-proof to follow the second route. Alitheia Core plug-ins are designed for re-use; if a basic plug-in exports individual measurements of small parts of each assessed resource state, then a high-level plug-in can combine measurements from various other plug-ins to a greater effect. In our case, we designed the described plug-in to calculate the number of lines that have changed in each new project version (and therefore be bound to the `ProjectVersion` entity) and then to use those measurements to calculate the rate of changed lines when a heated discussion is discovered. As expected, the measurement of the lines of code for the changed files is not performed inside our plug-in, but instead we re-use the size plug-in to obtain them for each changed file, as shown in the following code extract:

```
int getLOCResult(ProjectFile pf) {
  AlitheiaPlugin plugin = AlitheiaCore
      .getInstance()
      .getPluginAdmin()
      .getImplementingPlugin("Wc.loc");
  List<Metric> metrics = new ArrayList<Metric>();
  metrics.add(Metric.getMetricByMnemonic("Wc.loc"));
  Result r = plugin.getResult(pf, metrics);
  return r.getRow(0).getCol(0).getInteger();
}
```

Overall, the plug-in is bound to two entities (or activators) and defines three metrics; the "verloc" metric for storing the number of lines changed per revision, the "hotness" metric for storing an indicator of how heated a discussion is for each mailing list thread and the "hoteffect" metric for storing the changes in the line commit rates for the threads that have been identified as heated discussions. The plug-in developer needs to provide implementations for exactly five methods,

2. This is the message of the day on the FreeBSD project's main login server.

two for each activation type (`run()` and `getResult()`) and the `install()` method to register the exported metrics to the database.

After designing the plug-in outline, the implementation itself is easy given the wealth of features offered by the Alitheia Core API. Metric declarations take a single line of code in the `install()` method:

```
super.addSupportedMetrics("Locs changed in version",
    "VERLOC", MetricType.Type.PROJECT_WIDE);
```

Returning a result stored in the standard Alitheia Core schema is equally straightforward. The following code is the actuall implementation of the `getResult()` method for the `ProjectVersion` activator.

```
public List<ResultEntry> getResult(ProjectVersion pv,
    Metric m) {
        return getResult(pv, m,
          ResultEntry.MIME_TYPE_TYPE_INTEGER);
}
```

The following extract is from the `run()` method implementation for the `ProjectVersion` activator. It demonstrates how object relational mapping used in Alitheia Core simplifies access to project entities. The code calculates the total number of lines changed in a specific project version and then stores the result to the database.

```
for (ProjectFile pf : pv.getVersionFiles()) {
  if (pf.isDeleted()) {
    linesChanged += getLOCResult(pf.getPrevVer());
  } else if (pf.isAdded()) {
    linesChanged += getLOCResult(pf);
  } else { // MODIFIED or REPLACED
     linesChanged += Math.abs(
         getLOCResult(pf)
       - getLOCResult(pf.getPrevVer());
  }
}
ProjectVersionMeasurement pvm =
   new ProjectVersionMeasurement(m, pv,linesChanged);

dbs.addRecord(pvm);
```

The discussion heat plug-in implementation in total consists of 270 lines of Java code.

4.3. Results

We run the plug-in on 48 projects from the Gnome ecosystem. For each project, we imported the full source code repository and mailing list archives (current in January 2009) in our Alitheia Core installation. In total, the system's metadata updaters processed 151383 revisions (summing up to 976154 file changes) and 79 mailing lists with 106469 email messages organized around 41310 threads. The combined size of all data was 2.5GB. The preprocessing phase for the whole dataset took about 12 hours on a 8 core server, which also hosts the system database. In comparison, a shell script that checks out the first version of a local mirror of the Evince project repository and then loops over the first 2000 revisions invoking the `sloccount` program on every revision, takes about 60 minutes on the same hardware.

Table 3. Results from the discussion heat plugin.

Project	Cases	Avg. HotEffect
Evince	8	384
F-Spot	12	-114
Garnome	2	105
gedit	6	31
Gnome-Screensaver	3	29
Gnome-System-Tools	5	65
LSR	2	437
Meld	3	16
MlView	1	141
Muine	3	-368
Orbit2	14	44
Planner	7	39
Sabayon	4	-54
Sawfish	4	-182
Seahorse	3	3167
Tracker	19	2950
Vala	9	244

The discussion heat plug-in workload is mostly database bound; to evaluate its scalability we run the plug-in on a machine featuring a 16-thread 1GHz SPARC CPU and 8GB of memory. The machine is underpowered by modern standards for sequential workloads but provides a very good benchmark for application scalability. We configured Alitheia Core to run 24 parallel jobs to compensate for the extensive I/O caused by frequent accesses to the database. During execution, the processor utilization was steadily more than 80%. The database was run on our dedicated 8 core database server on which the CPU load never exceeded 50%. The total execution time was 2 hours and 40 minutes. Obviously, the presented performance numbers are plug-in specific, and in fact other plug-ins use significantly more CPU power.

The results of the application of the plug-in on the aforementioned dataset can be seen in Table 3. The table lists the number of email threads that were identified as heated discussions and their average effect in terms of the rate of lines of code changed in the project's repository before and during the intense discussion. From the 48 measured projects, only 17 appeared to have intense discussions using our relatively strict definition presented above. The surprising finding of this experiment is that for the projects we measured, intense discussions appear to have an overall positive rather than negative effect in most cases. This means that after or during an intense discussion the rate of lines of code committed to the repository increases. This result seems to contradict expectations and requires further investigation.

We used this experiment to demonstrate Alitheia Core's efficiency in conducting software engineering research by combining raw data and results from other tools. A relatively complex experiment was reduced to a set of simple to implement methods, thanks to the rich abstractions that Alitheia Core exposes to the developer. The discussion heat

plug-in is now part of our Alitheia Core installation and is being continuously run as new data arrive to our data-mirroring system.

5. Related work

The continuous metric monitoring approach towards achieving software quality is not new. The first systems that automate metric collection emerged as 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. [24], [25]), usually to support quality models or management targets set using the Goal-Question-Metric approach [23]. Alitheia Core is able to process more data sources and while it does feature a quality model implementation, it is not tied to it, allowing the user to combine arbitrary software metrics towards a custom definition of quality.

Two early systems that attempted the merging of heterogeneous data from software repositories are the Release History Database [26] and Softchange [27]. Both where based on a collection of special purpose scripts to populate databases with data coming from from certain projects. As such, they were not adopted by the research community. Hipikat [28] collected data from both repositories, mailing lists and bug databases but its scope and purpose was different: to provide guidance to developers based on the extraction of prior organizational knowledge. The overall architecture of Hipikat bears a resemblance to that of Alitheia Core as both systems were designed to handle generic sets of large data volumes. Hipikat is a purpose specific system; Alitheia Core offers more generic abstractions of the underlying data sources and defers processing to external clients (plug-ins).

The Hackystat [29] 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.

A number of projects have considered the analysis of OSS development data for research purposes. Flossmole [30] was first to provide a database of preprocessed data from the Sourceforge OSS development site. The FlossMetrics project [31] runs various independent analysis tools on a large set of software repositories and publishes the resulting datasets. Both of these projects aim to provide datasets to the community, while Alitheia Core focuses more on the provision of research infrastructure platform.

6. Conclusions

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

Future planned work on the platform includes the expansion of the data accessors plug-ins to include support for other source code management systems, a web service that will allow external plug-in submissions to be run against the preprocessed data currently hosted on our servers and a REST-based API for accessing project metadata and measurements in a simple way.

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. The authors would like to thank Stavros Grigorakakis for his help in organizing the mirrored project data and Efthymia Aivaloglou for her contribution in optimizing HQL/SQL queries.

References

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

[2] ACM-SIGKDD, "The knoweledge discovery and data mining cup contest." [Online]. Available: http://www.sigkdd.org/kddcup/

[3] B. Alpern, S. Augart, S. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind *et al.*, "The Jikes research virtual machine project: building an open-source research community," *IBM Systems Journal*, vol. 44, no. 2, pp. 399–417, 2005.

[4] D. Spinelis, "Global software development in the FreeBSD project," in *International Workshop on Global Software Development for the Practitioner*, P. Kruchten, Y. Hsieh, E. MacGregor, D. Moitra, and W. Strigel, Eds. ACM Press, May 2006, pp. 73–79.

[5] D. Spinellis, "A tale of four kernels," in *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds. New York: Association for Computing Machinery, May 2008, pp. 381–390.

[6] L. Gasser, G. Ripoche, and R. J. Sandusky, "Research infrastructure for empirical science of f/oss," in *Proceedings of the First International Workshop on Mining Software Repositories (MSR 2004)*, Edinburg, Scotland, UK, 2004, pp. 12–16.

[7] L. Gasser and W. Scacchi, *Open Source Development, Communities and Quality*, ser. IFIP International Federation for Information Processing. Springer Boston, Jul 2008, vol. 275, ch. Towards a Global Research Infrastructure for Multidisciplinary Study of Free/Open Source Software Development, pp. 143–158.

[8] J. Howison, A. Wiggins, and K. Crowston, *Open Source Development, Communities and Quality*, ser. IFIP International Federation for Information Processing. Springer Boston, Jul 2008, vol. 275, ch. eResearch Workflows for Studying Free and Open Source Software Development, pp. 405–411.

[9] A. Bosin, N. Dessí, M. Fugini, D. Liberati, and B. Pes, *Open Source Development, Communities and Quality*, ser. IFIP International Federation for Information Processing. Springer Boston, Jul 2008, vol. 275, ch. Open Source Environments for Collaborative Experiments in e-Science, pp. 415–416.

[10] G. Gousios, V. Karakoidas, K. Stroggylos, P. Louridas, V. Vlachos, and D. Spinellis, "Software quality assessment of open source software," in *Proceedings of the 11th Panhellenic Conference on Informatics*, May 2007.

[11] G. Gousios and D. Spinellis, "Alitheia core: An extensible software quality monitoring platform," in *Proceedings of the 31rst International Conference of Software Engineering - Research Demos Track*, 2009, to appear.

[12] *OSGi Service Platform, Core Specification*. OSGi Alliance, 2007.

[13] E. J. O'Neil, "Object/relational mapping 2008: Hibernate and the entity data model (edm)," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2008, pp. 1351–1356.

[14] G. Robles and J. M. Gonzalez-Barahona, "Developer identification methods for integrated data from various sources," in *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*. New York, NY, USA: ACM, 2005, pp. 1–5.

[15] G. Navarro, "A guided tour to approximate string matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, 2001.

[16] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[17] G. Gousios, E. Kalliamvakou, and D. Spinellis, "Measuring developer contribution from software repository data," in *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*. New York, NY, USA: ACM, 2008, pp. 129–132.

[18] I. Samoladas, G. Gousios, D. Spinellis, and I. Stamelos, "The SQO-OSS quality model: Measurement based open source software evaluation," in *Open Source Development, Communities and Quality — OSS 2008: 4th International Conference on Open Source Systems*, E. Damiani and G. Succi, Eds. Boston: Springer, Sep. 2008, pp. 237–248.

[19] M. Dery, Ed., *Flame Wars: The Discourse of Cyberculture*. Duke University Press, 1994.

[20] K. Coar, "The sun never sits on distributed development," *Queue*, vol. 1, no. 9, pp. 32–39, 2004.

[21] K. Fogel, *Producing Open Source Software*. Sebastopol: O'Reilly Media, Inc, 2005, pp. 261–268.

[22] C. N. Parkinson, *Parkinson's Law: The Pursuit of Progress*. John Murray, 1958.

[23] V. Basili and D. Weiss, "A methodology for collecting valid software engineering data," *IEEE Transactions on Software Engineering*, vol. 10, no. 3, pp. 728–738, Nov 1984.

[24] V. Basili and H. Rombach, "The TAME project: towards improvement-oriented software environments," *IEEE Transactions on Software Engineering*, vol. 14, no. 6, pp. 758–773, June 1998.

[25] S. Komi-Sirviö, P. Parviainen, and J. Ronkainen, "Measurement automation: Methodological background and practical solutions-a multiple case study," *IEEE International Symposium on Software Metrics*, p. 306, 2001.

[26] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *ICSM '03: Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2003, p. 23.

[27] D. M. German, "Mining CVS repositories, the Softchange experience," in *Proceedings of the First International Workshop on Mining Software Repositories*, Edinburg, Scotland, UK, 2004, pp. 17–21.

[28] D. Cubranic and G. Murphy, "Hipikat: recommending pertinent software development artifacts," in *Proceedings of the 25th International Conference on Software Engineering*, May 2003, pp. 408–418.

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

[30] 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*, vol. 1, no. 3, pp. 17–26, 2006.

[31] GSyC/LibreSoft, "The FlossMetrics project." [Online]. Available: http://flossmetrics.org/