Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Conducting Quantitative Software Engineering Studies with Alitheia Core

Georgios Gousios · Diomidis Spinellis

Received: February 19, 2013/ Accepted: date

Abstract Quantitative empirical software engineering research benefits mightily from processing large open source software repository data sets. The diversity of repository management tools and the long history of some projects, renders the task of working those datasets a tedious and error-prone exercise. The Alitheia Core analysis platform preprocesses repository data into an intermediate format that allows researchers to provide custom analysis tools. Alitheia Core automatically distributes the processing load on multiple processors while enabling programmatic access to the raw data, the metadata, and the analysis results. The tool has been successfully applied on hundreds of medium to large-sized open-source projects, enabling large-scale empirical studies.

Keywords quantitative software engineering · software repository mining

1 Introdution

During the last decade, the availability of open source software (OSS), has changed not only the software development landscape (Spinellis and Szyperski 2004), but also the way software is being studied. OSS projects make their software configuration management (SCM), mailing lists and bug tracking systems (BTS) publicly available. Researchers have tapped on the availability of such rich process and product data sources to investigate empirically a number of important research questions; (see for instance Mockus et al 2002; Samoladas et al 2004). However, research with software repositories is not a trivial process. Collecting and preprocessing data, calculating metrics, and synthesizing composite results from a large

G. Gousios (Work performed while at Athens University of Economics and Business)
Department of Software and Computer Technology
Delft University of Technology
E-mail: G.Gousios@tudelft.nl

D. Spinellis
Department of Management Science and Technology
Athens University of Economics and Business
Tel: +30-210-8203981
E-mail: dds@aueb.gr

corpus of project artifacts is a tedious and error prone task lacking direct scientific value (Robles 2005). To make the results credible and generic, the formulated hypotheses must be validated against data from multiple projects (Perry et al 2000).

In this paper, we present Alitheia Core, an extensible platform that enables large-scale software engineering quantitative studies with data from software repositories. The design of Alitheia Core facilitates researchers to deal with the chores of analyzing data from software repositories. To do so, it provides a software platform that implements large parts of the required data preprocessing functionality while remaining extensible to new data and analysis types. In addition, it can handle the large-scale data processing involved, by exploiting multi-core and multi-machine hardware configurations.

The objective of this paper is to present our approach toward conducting large-scale quantitative software engineering studies through Alitheia Core. We describe the domain's challenges, the design and implementation of Alitheia Core and investigate its performance and applicability on the problem domain it was designed for. The first version of the Alitheia Core design, input data and plug-in design have been reported in other work (Gousios and Spinellis 2009). Since then, Alitheia Core has been used to study developer contribution in OSS projects (Kalliamvakou et al 2009), behavioural characteristics of developers (Lin et al 2010) and the evolution of security properties of software projects (Mitropoulos et al 2012). The main contributions of this work are the introduction of an extensible platform that enables large-scale software engineering studies and the presentation of Alitheia Core as its main data processing component. Alitheia Core is also evaluated through comparison with existing approaches, extensive performance analysis and a case study through which we demonstrate that such large-scale experimentation is indeed possible. We also discuss the shortcomings of our approach and present the lessons learned while building it.

2 Problem Statement

According to several authors (Basili 1996; Wohlin and Wesslen 2000; Perry et al 2000; Sjøberg et al 2005), software engineering is an empirical science, as both the studied artifacts and the developed methods are (or are applied on) real, existing systems. Currently, there is a growing interest in software engineering research to use data originating from software repositories. The developed techniques and tools are referred to as techniques for Mining Software Repositories (MSR), after the name of the eponymous series of workshops and conferences. The wealth of combined process and product data residing in OSS repositories has attracted the interest of many researchers, as this is the first time in the history of software engineering that large-scale empirical studies can be performed with real data outside the walls of large software development organizations. However, even though they are free to use, OSS data come with a cost for researchers.

– Each project uses its own combination of SCM, mailing list, BTS and other project management tools, such as Wikis and documentation systems (Robles 2005). Without the appropriate abstractions, it is challenging to build tools that can process data from multiple projects at the same time.

– Empirical studies are often conducted in several phases, and a diverse set of tools can be applied in each phase (Robles et al 2004). Intermediate results must be stored and retrieved efficiently in formats suitable for use by chains of tools (Fischer et al 2003; Bevan et al 2005).
– During a lifetime spanning multiple decades, several OSS projects have amassed gigabytes of data worth studying (Mockus 2009). The computational cost for processing such large volumes of data is not trivial. For fast analysis turnaround, researchers must design their analysis tools to exploit modern multiprocessor machines. However, those efforts are often not enough, as the capacities of single machines are easily saturated by the processed data volumes.

Perhaps as a result of the above, several studies (Zelkowitz and Wallace 1997; Sjøberg et al 2005; Zannier et al 2006), show that empirical studies in software engineering neither take full advantage of the data on offer nor (or at least, seldom) base their experimental results on artifacts of prior studies. The same studies urge software engineering researchers to validate their models more rigorously, as this is what drives the general applicability of the performed studies (Perry et al 2000).

It is widely believed that software engineering as a discipline should strive toward more rigorous experimentation (Tichy et al 1995; Basili 1996; Seaman 1999; Glass et al 2002; Deligiannis et al 2002; Shaw 2003; Sjøberg et al 2005; Zannier et al 2006; Šmite et al 2010). Data from OSS projects present unique opportunity in that respect, as they provide researchers with access to rich historical process and product data originating from some extensively used, often high quality software projects (Mockus et al 2002; Samoladas et al 2004; Spinellis 2008). Existing studies only partially take advantage of the existing wealth of data, thus rendering the presented results vulnerable to falsification and precluding their re-use in other studies.

The main problem we are trying to tackle with this work is how software engineering researchers can utilize the vast quantities of freely available data efficiently, while allowing fast analysis turnaround. For this, we propose an open research platform specifically designed to facilitate experimentation with large software engineering datasets derived from diverse data sources. Our idea is based upon four basic principles.

– Rigorous testing of software engineering research theories, models, and tools with empirical data is a prerequisite for obtaining credible research results.
– Large volumes of empirical data do exist and are freely available.
– Sharing of research tools and data accelerates research innovation.
– User ecosystems are formed around software platforms.

The research platform idea is not new; in several other empirical science fields either shared infrastructures (e.g. the Large Haldron Colider and the International Space Station) or common datasets (e.g. DNA sequences) have been used for decades, because producing and maintaining those is resource intensive and expensive. In the field of computer science, research platforms exist in several fields (e.g. JikesRVM (Alpern et al 2000) for virtual machine research, LLVM (Lattner and Adve 2004) for compiler research, Weka (Witten and Frank 2005) for machine learning research), especially in cases where exploring novel ideas requires a significant amount of groundwork. Few works (Gasser and Scacchi 2008) have discussed

the need for common experimentation infrastructures in the context of software engineering research, even though the need for more rigorous experimentation has been identified multiple times.

The research platform whose principles we outlined forms part of a long term research effort aiming to improve the quality of software engineering studies. In the following sections, after presenting the requirements for the envisaged platform and the existing work in the field of tool support for software engineering research, we report on the core component, the data processing and analysis tool.

3 Requirements

Platforms (Gawer and Cusumano 2002), as opposed to concrete tool implementations, have distinct requirements that govern their design and use. In the case of the platform we propose, the most important requirement is that it must provide the foundation for conducting arbitrary studies by integrating and abstracting various *data sources* in an extensible, yet concise intermediate representation for metadata. Currently, OSS projects use a variety of tools to manage their project repositories. Consequently, analysis tools that researchers use must be re-written for each repository management tool. Moreover, **semantic data integration** requires project-specific information (Robles and Gonzalez-Barahona 2005; Canfora and Cerulo 2006; Ratzinger et al 2008; Goeminne and Mens 2011). For example, when matching developer identities across data sources, both project-specific naming rules, generic name abbreviation patterns and company-specific username generation patterns might apply (Goeminne and Mens 2011). Therefore, in order to enable cross-project examination and comparisons, an analysis tool must reconcile the semantic differences between diverse systems managing the same data.

The size of the data to be processed imposes **scaling** challenges. Currently, long-lived projects, such as GNU Emacs and FreeBSD, have repositories that contain more than 20 years of history, and their data sizes exceed the gigabyte mark. On a larger scale, Mockus (2009) reported on a gathered data set exceeding a terabyte in size. In order to cope with the vast data sizes a research platform aiming to integrate data from many projects must be scalable through multiprocessing and distributed processing. Fortunately, many analysis methods in software engineering research are trivially parallelizable, as they usually apply a tool on independent states of a project's data and aggregate selected results. However, some analysis methods must be applied serially; in such cases, parallelism can be achieved by running the analysis method on multiple projects concurrently. A large-scale analysis platform must provide the appropriate process abstractions that will enable efficient scheduling of analysis tasks on available processors, dependencies among tasks to preserve the correct order of execution, automatic resource management and clustering capabilities that will enable load distribution on a cluster of local or cloud-based computers.

Scalable processing and data integration form the basis on which software analysis platforms should be built; the platform should then offer useful **generic services** to researchers, so that they can easily implement their analysis tools. Such services can include

– programmatic access to both raw and intermediate representations of platform data,

– extensible data processing workflows that allow developers to inject their custom analysis code,
– automation of the experimentation process through workflows,
– source code parsers to analyze syntactic and semantic properties of the source code.

Analytics platforms must also assist with **data exploration**. Generating the data through efficient and scalable mechanisms may be beneficial for research, but creates problems in data analysis efficiency (Buse and Zimmermann 2012). Platforms should have support for visualizing datasets, either through extensible visualization frameworks or through APIs that provide the analysis data in easy to digest formats for external tools.

Finally, platforms need to be **open to the community** of other researchers. For that, platforms must be fully documented, while they should be distributed as open source software, in order to enable independent replication and inspection of the data processing algorithm implementations. Datasets should be distributed along with research platforms, in the platform's native data format.

4 Existing Approaches

Two areas of research on software engineering tools are related to our work: platforms for mining software repositories and large scale software engineering research facilities. We focus our description on platforms. Platforms, as opposed to tools or combinations of tools in the form of scripts, have at least the following distinctive characteristics.

– They provide well-documented open interfaces.
– They offer extensible data representations to accommodate varying requirements.
– They automate tool chain invocations and regulate access to data and other platform subsystems.
– They anticipate future developments by featuring well defined extension points for the workflows they support.

We therefore do not include in our overview tools that perform data transformations or constitute proof of concept implementations of algorithms.

4.1 MSR Platforms

Researchers working with empirical data understood early on that standalone product data measurements do not suffice. A variety of tools have been developed to automate the process of extracting and processing data from software process support systems. The Release History Database (RHDB) (Fischer et al 2003) was the first to combine data from more than one data sources, namely from bug tracking and SCM systems. The RHDB tool uses the CVS log to extract and store to a database information about the files and versions that changed. The CVSAnaly tool (Robles et al 2004), converts information from SCM repositories to a relational format. CVSAnaly works in three steps; it first parses the CVS log, then it cleans the

data and extracts semantic information from it, and finally it produces statistical data about the project. A similar tool is SoftChange (German and Hindle 2005). SoftChange extracts data from additional datasources (change log files) and infers facts from the source code once the data extraction is finalised. The tools presented here do not constitute platforms in the definition we provide above; however, they were early efforts in the field to use intermediate data formats, which could then be reused.

The Hackystat tool (Johnson et al 2005) was the first effort that considered 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 developers' toolchains. The sensors monitor the developers' use of tools and update a centralized server. Since Hackystat was designed as a progress monitoring console rather than a dedicated analysis tool, it is not suitable for post-mortem analyses, like the ones performed on OSS repositories. However, it provides valuable information about the software process while it is developed.

The Hipikat tool (Cubranic et al 2005) was designed to act as an automated store of project memory. It provides artifact-based search for project related artifacts. For instance, it combines structural relationships with relationships found by measures of textual similarity applied on source code commits. It can therefore retrieve context related data based on previous project behavior, which it then presents to developers in the form of recommendations within their development environment.

The Kenyon tool (Bevan et al 2005), is a platform that pre-processes data from various types of SCM repositories in a unified schema and then exports the database to other tools, which are invoked automatically. The Kenyon metadata database is specifically tuned for studying source code instability factors. Kenyon was the first platform to abstract repository and source code data in the form of runtime models, against which analysis tool results are stored. Alitheia Core's data model is more comprehensive, including models for email and bug database analysis, while it can be extended by plug-ins. The Sourcerer project (Linstead et al 2009) built a software analysis platform for extracting facts from Java programs that can then be used for indexing and searching over large collections of projects.

A tool similar to Alitheia Core is Evolizer (Gall et al 2009). Evolizer is a platform that enables software evolution analysis within the Eclipse IDE. It leverages Eclipse's Java parsers to provide in-depth static analysis information for Java programs. Except from source code, modules exist to import metadata from software repositories (SCM and BTS) into a relational database. Alitheia Core improves over Evolizer by providing a generic metadata schema that abstracts diverse repository management systems, by enabling the analysis of mailing lists and by being designed to run without user intervention on many projects. However, it lacks Evolizer's data linking facilities.

Tesseract (Sarma et al 2009) is an interactive framework for exploring socio-technical relationships in software development. At the heart of Tesseract lies a tool that analyzes and links data from Subversion and Bugzilla repositories, as well as emails. Its intermediate format can support several projects in the same database, while its analysis is mostly directed to recovering links between communication and source code artifacts.

Moose (Nierstrasz et al 2005) is a meta-platform for software analysis. At its core, Moose supports the specification of models representing various aspects of

the software development process. Researchers can re-use existing models, such as FAMIX (Demeyer et al 1999), for describing the static structure of object oriented systems), or use Moose's meta-modelling facilities to create new models for custom analyses. The services offered by Moose include parsers for various languages (at various levels of maturity), data visualizations and importers for various data formats. Moose differs from Alitheia Core in that it promotes abstraction and meta-facilities over pre-defined processing workflows and data formats. In effect, Moose is a superset of Alitheia Core (in fact, of all presented repository mining platforms). Moose's open ended platform architecture spawned a number of tools and has seen significant use in the Smalltalk language community.

Through the Hismo extension (Gîrba and Ducasse 2006), Moose gains the ability to model and process software histories. Hismo provides a representation of a project's revision log, in each entry of which FAMIX models can be attached. From an abstract point of view, Hismo's and Alitheia Core's history and code representation models are very similar: they both model project history as a stream of consecutive events (originating in revisions in the project's SCM tool) and allow the attachment of code models to each event. However, Alitheia Core's modeling is more fine-grained: apart from the revisions, the file structure and the code structure are also individually versioned. This permits Alitheia Core to store metadata of a project's history more efficiently and also to optimize tool invocations based on the exact data that changed from revision to revision.

Churrasco (D'Ambros and Lanza 2010) is an example of the MSR tools that spawned out of the Moose community. Churrasco was designed to support collaborative software evolution analysis. It can process data from SVN and Bugzilla repositories, link them and store them in a metadata database. It also supports the generation of FAMIX models for the latest version of the analyzed project. Churrasco automates the acquisition of data from OSS project repositories through its web interface, while providing interesting visualizations and collaborative annotation of the data exploration process. The Small Project Observatory (Lungu et al 2010) attempts to analyse software ecosystems rather than simple projects. Similarly to Alitheia Core, it can store several projects in the same database, while it has been shown to scale to the order of hundreds of projects. In addition to Alitheia Core, all the above mentioned platforms offer visualizations of ecosystem evolution and individual developer activity. Both however lack support for pluggable analysis tools and associating measurements with artifact versions.

All tools presented above share a few underlying design principles. First, they process raw data and extract metadata, which are stored in a relational schema and then processed further. In addition, the newest of those tools can incorporate and reference metadata from multiple repositories. Apart from SCM data, they can also process and link together other types of data such as source code and bugs. A more detailed analysis of the similarities and the differences among the tools is presented in Section 6.1.

4.2 Scaling Research

The need for large-scale research facilities has been identified by many authors (Zelkowitz and Wallace 1997; Sjøberg et al 2005; Zannier et al 2006). Current practice focuses on making specific tools scale (Livieri et al 2007; Shang et al 2011) or providing

pre-processed datasets (Howison et al 2006; Herraiz et al 2009; Linstead et al 2009; Tempero et al 2010). In addition, initial work has provided requirements for e-Science platforms and workflows (Howison et al 2008; Gasser and Scacchi 2008) for large-scale quantitative software engineering research.

On the tools front, Livieri et al (2007) modified the CCfinder code cloning identification tool to work on a cluster of machines. The throughput gained enabled them to process very large projects in a fraction of the time required in the case of single machines. Shang et al (2010), attempted to combine large-scale, distributed processing paradigms, specifically MapReduce (Dean and Ghemawat 2004), with existing tools in order to facilitate research with large software repositories. This effort was later expanded (Shang et al 2011), where the authors explore the use of Pig language and associated tools over Hadoop as a runtime for a standalone tool (J-REX) on a large collection of data. They found that only inherently data-parallel analysis algorithms can efficiently exploit the MapReduce paradigm, while porting existing tools to Hadoop requires significant amounts of tool modifications and glue code.

Along similar lines Boa (Dyer et al 2012) leverages the Hadoop implementation of the MapReduce framework to allow the execution of software repository queries on a cluster. We already outlined the difficulties Shang et al encountered in expressing repository analysis tasks in a form that lends itself to efficient execution on a Hadoop cluster. Boa's developers propose to solve this problem by providing a domain-specific language, modelled after Sawzall (Pike et al 2005), whose constructs are powerful enough to express queries on software artifacts and can at the same time be efficiently executed on the cluster.

Similarly to other big data fields, large-scale software engineering research can be facilitated by appropriate datasets.

In the FLOSSmole project Howison et al (2006) collected metadata from OSS projects hosted on the Sourceforge site and offered both the dataset and an online query interface. The dataset consists of metadata about the software packages, such as numbers of downloads or the employed programming language. The FLOSSMetrics project (Herraiz et al 2009) provides a dataset consisting of source code size, structure and evolution metrics from several OSS projects. The Sourcerer project (Linstead et al 2009), apart from the analysis tools, provides a pre-processed dataset of code structure metrics from thousands of Java projects. Finally, the Qualitas corpus project (Tempero et al 2010), provides a curated collection of different versions of Java projects, in source code form.

In the recent years, the Promise repository has become a prime source of data for software engineering studies. The repository started as a container for the datasets published in the homonymous working conference, but its use has since expanded. It currently contains more than 100 datasets of various sizes, the majority of which lie in the areas of defect and effort prediction.

5 The Alitheia Core Analysis Platform

To fulfill the requirements presented in Section 3, we designed and implemented Alitheia Core, an integrated platform for analyzing data from software repositories. Alitheia Core integrates data and process datasets, a relational database for

metadata and tool results, parsers for several repository formats, and tool abstractions that can be extended to provide custom analysis methods. In this section, we present the architecture and implementation of the Alitheia Core platform.

5.1 Data

Raw data Alitheia Core works with data originating from three classes of external systems, namely SCM systems, BTS and mailing lists. It requires raw data to be accessible on the host filesystem; data retrieval from the project hosting sites is delegated to external tools and mirroring processes. Alitheia Core expects the data for each project to be stored in a directory, which is organized as follows:

SCM data Alitheia Core supports two distinct types of SCM data sources: centralized systems (Subversion) and decentralized systems (Git). To mirror the repositories, standard tools such as svnmirror for Subversion and git clone in the case of Git can be used.

Mailing List data Alitheia Core can process data from mailing list archives, using the maildir format (Bernstein 2000) for storing email messages, in a directory per mailing list configuration. Emails can be retrieved from Mailman archives from mbox mailboxes, or by subscribing to the a mailing list and using fetchmail rules to route them to the appropriate mailing list subdirectory.

Bug Data Currently, there are two widely used systems in OSS, namely Bugzilla and Jira. Both offer roughly the same functionality to end users, and both can be extended with project-specific fields. Alitheia Core currently supports Bugzilla; automated scripts can download and maintain a mirror of bugs from a Bugzilla installation that supports the remote procedure call protocol. Bugs are downloaded as XML files, one file per bug.

Metadata Alitheia Core uses a relational database management system (RDBMS) to store metadata about the processed projects (see Figure 1). The role of the metadata is not to create replicas of the raw data in the database, but to provide a set of entities against which analysis tools work, while also enabling efficient storage and fast retrieval of the project state at any point of the project's lifetime. Moreover, the storage schema is designed to include the minimum amount of data required in order to abstract the differences between raw data of various systems.

An indicative example of the metadata representation's parsimony is that of processing SVN revision information to record changes to project files through commits. SVN keeps track of changes to files as a sequence of four allowed operations (add, mofidy, delete and copy) along with the textual difference that was created by the operation on the file. Alitheia Core maps those operations to corresponding entries in the ProjectFile table. Therefore, for each new commit, Alitheia Core will create an entry in the ProjectVersion table along with as many entries in the ProjectFile table as the number of file changes reported by the commit. The same approach is also followed when analyzing the source code to extract changes in classes and methods; Alitheia Core will only record a new version of a method if the commit affected code in the method's body, instead of recording all methods as changed due to a new version of the file.

The role of the metadata schema is pivotal in disengaging analysis tool implementation from raw data formats. Using the metadata schema, analysis tools can

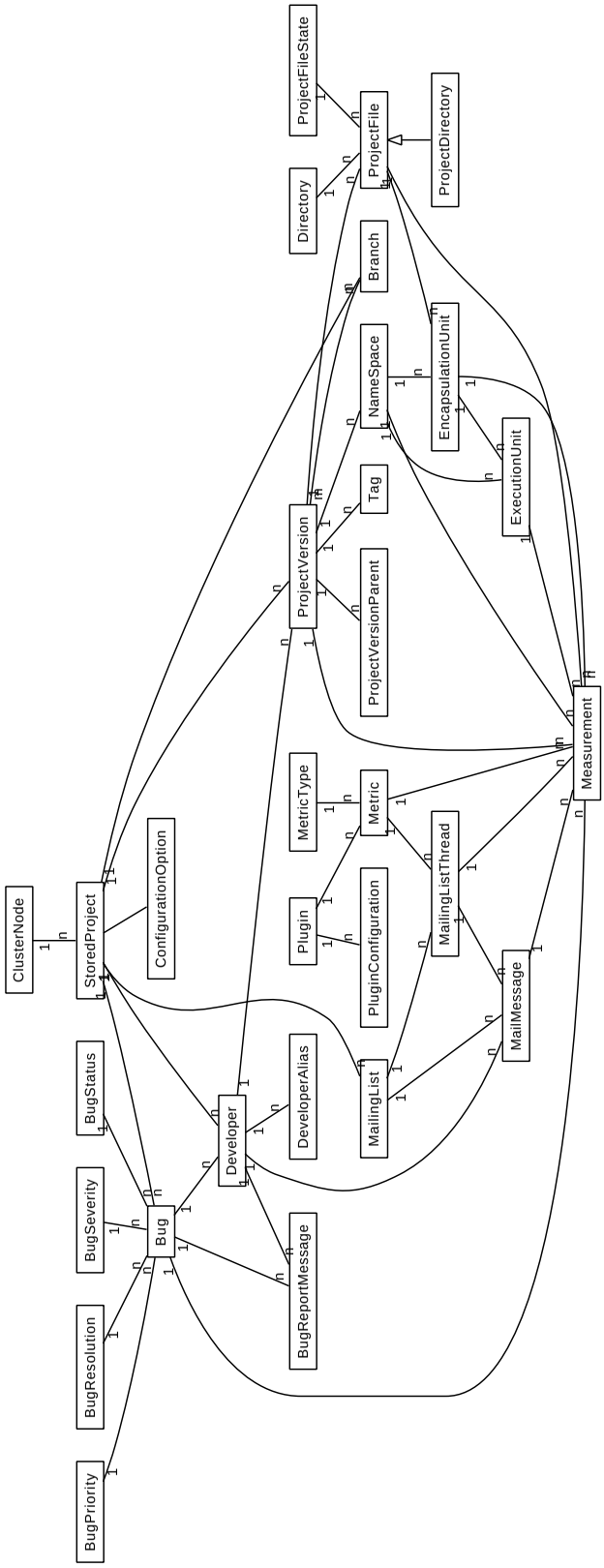


Fig. 1 Simplified view of the Alitheia Core metadata schema

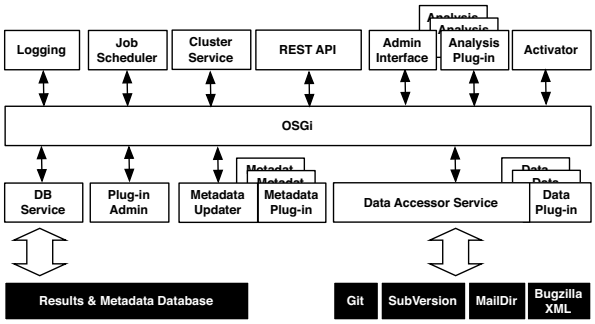


Fig. 2 The Alitheia Core tool architecture

process version control system logs, project files, mailing list threads and other entities without resorting to the raw data stores. This allows analysis tools to be written faster, to be compact and, more importantly, to be broadly applicable. The intermediate data approach may not be novel (Kenyon, Hipikat, Evolizer, Sourcerer and Moose use it at various granularity levels), however this is the first time that an intermediate schema can integrate data from many projects and all major raw data formats, including both distributed and centralized SCMs, in a space conserving manner.

5.2 Architecture

Alitheia Core is modeled as a set of co-operating services, all attached on a shared bus (see Fig. 2). Each service performs a well defined set of operations and is accessible to the rest of the system through a service interface. The OSGi plug-in model is used as the basis for Alitheia Core; consequently all service plug-ins are modeled as OSGi bundles. OSGi provides Alitheia Core plug-ins with service discovery and service versioning functionalities while also functioning as a lightweight application server to host the plug-ins at runtime. The basic services are the following:

DB Service The DB 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 entities used throughout the system to model project resources. It uses object relational mapping (ORM) to eliminate the barrier between runtime types and stored data and has integrated transaction management facilities. ORM facilitates plug-in implementation by transparently converting simple queries to method calls and hiding complex queries inside model navigation methods.

Data Accessors Data accessors provide a two level abstraction of the data managed by Alitheia Core. The low-level part of the accessors stack exposes an abstracted view of the raw data formats; this is done by modeling abstract classes of systems and providing drivers that convert the underlying data to the common representation. Its function is reminiscent of the virtual file system layer found in many operating systems (Spinellis 2007). The raw data accessor is extensible via plug-ins.

The upper layer encapsulates common non-destructive actions done on data stores. It combines raw data with metadata, while optimizing common access patterns and managing resources. Examples of such optimizations include in memory checkout reconstructions (with pointers to raw data), project timeline views across all types of data for each project and multiple concurrent on disk checkouts.

Job Scheduler One of the important functions Alitheia Core performs is the splitting of the processing load on multiple CPUs. Although one might be tempted to run all analysis workflow steps run in parallel, in practice algorithmic or data constraints may hinder the scheduler's ability to parallelize the work. The job scheduler service maintains a task queue and a configurable size worker thread pool and assigns tasks from the task queue to idle workers. Tasks in Alitheia Core are co-operative; at any point, a task can yield its execution to allow space for dependent tasks to execute. The scheduler will automatically resume the yielded task when dependencies have finished execution. This allows for MapReduce-like functionality to be implemented in memory.

Activator The activator service initiates analysis plug-in calculations. More details can be found in Section 5.3.

Cluster Service The cluster service regulates the operation of Alitheia Core on a cluster of machines. Currently, it supports project-level distributed processing by means of associating projects with specific cluster nodes.

Interfaces Alitheia Core has two types of interfaces; a simple web-based interface allows the addition of projects and the initiation of data analysis operations. In addition, a REST API provides read-only access to the data stored in the Alitheia Core database. More details can be found in Section 5.5.

5.3 The plug-in mechanism

To cope with the availability of a large number of data formats and data analysis methods, Alitheia Core was designed from the ground up to be extensible. The Alitheia core engine can be extended by plug-ins that perform various types of data analyses and by plug-ins that provide accessor mechanisms to new data formats.

Data accessor plug-ins provide Alitheia Core with interfaces to external systems (more specifically, their data stores). Alitheia Core models abstract classes of such systems, based on their function and their operations. Currently, the modeled systems include version control, bug tracking and mailing list management systems; specifying new ones is a matter of extending the accessor service with abstract definitions of behavior and data. Moreover, to cater for the semantic differences between various implementations of classes of services, for example between centralized and distributed version control systems, the metadata updaters are extensible, too. Usually, data plug-ins include both new data accessors and new metadata updaters for the processed data format. The list of data and metadata updaters currently in Alitheia Core is presented in Table 1.

Analysis plug-ins process raw data or metadata and extract measurements or other bits of information, referred to as metrics. Each analysis plug-in in Alitheia Core is associated with a set of activation types. An activation type indicates that the analysis must be performed in response to a change to the corresponding project asset; this is the name of the database entity that models the asset.

Table 1 Data and metadata processing plug-ins in Alitheia Core

Algorithm	Phase	Input	Output	Description
svnmap	MI	Subversion repository	ProjectVersion, ProjectFile, Developer, DeveloperAlias	Parses Subversion revisions to a relational format.
gitmap	MI	Git repository	ProjectVersion, ProjectFile, Developer, DeveloperAlias	Parses Git revisions and stores metadata, linking them to raw revisions.
bugmap	MI	BugzillaXML data	Bug, BugReport, Developer, DeveloperAlias	Stores selected bug and developer metadata
mailmap	MI	maildir archives	MailingList, MailMessage, Developer, DeveloperAlias	Extracts email metadata and developer information from email messages organized in mailing lists.
idresolv	MRI	Developer, DeveloperAlias	Developer, DeveloperAlias	Uses heuristics to consolidate developer identities across data sources.
threadresolv	MRI	MailMessage	MailThread	Parses mail messages and reorganizes them into threads.
modresolv	MRI	ProjectFile	ProjectFile	Identifies directories as source code modules based on whether they contain source code files.
javaparse	P	ProjectFile	Namespace, ExecutionUnit, EncapsulationUnit	Parses Java source code and extracts information about changed packages, classes and methods.
pythonparse	P	ProjectFile	Namespace, ExecutionUnit, EncapsulationUnit	Parses Python source code and extracts information about changed classes, methods and modules.

MI Metadata Import, MRI Metadata Relationship Inference, P Parsing

Therefore the metric is activated each time a new entry is added to the database table (for example, a new entry in the ProjectFile table will automatically trigger the project size metric to calculate a result). Alitheia Core can automatically calculate the set of entities per activation type for which a specific analysis plug-in has not been run for and consequently can schedule jobs for invoking the analysis plug-in on those entities. An analysis plug-in can define several metrics, which are identified by a unique short name (mnemonic). 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 metadata entry the metric was calculated against.

5.4 Data Processing Workflow

When Alitheia Core processes a new project it typically performs the following actions.

Project data mirroring The project data is mirrored locally. Alitheia Core does not provide any support for retrieving or mirroring project data, but it is distributed with scripts that a system administrator can install to setup mirroring.

Metadata import The appropriate raw data processor ("updater" in Alitheia Core) syncs the data in the project mirror to the metadata in the Alitheia Core database.

Source code parsing The configured source code parser analyzes the project's source code and extracts information about changed namespaces, modules and functions. Multiple source code parsers can exist per project.

Metadata relationship inference At this stage, analysis tools can extract relationships between metadata entities. Examples include linking of bug reports to source code commits and resolution of developer identities. Metadata inference tools might modify metadata in place or store the extracted relationships in new locations.

Analysis plug-in invocation Analysis plug-ins extract interesting facts about either the raw data or the metadata and store them into the database.

All plug-ins in the Alitheia Core workflow can have dependencies to other plug-ins or to whole workflow phases. Alitheia Core uses this information to topologically sort the plug-in executions before scheduling them on the workqueue. If no dependencies exist, processing occurs in parallel for plug-ins within the same phase. Alitheia Core also provides mechanisms for parallelizing the work within the context of a workflow step. Consider the bug to commits linking scenario, where N bugs need to be checked against M commits for references. The job scheduling mechanism can be used to spawn M jobs each of which analyzes all N bugs retrieved by the linking process during initialization.

All steps of the workflow are *idempotent* by convention (in case of analysis plug-ins, by design), meaning that results are only calculated once and subsequent invocations of tools will not change them. Idempotence allows tool invocations to be stopped at any stage of processing and be restarted without requiring to recalculate the analysis state up to the stop point. The work scheduler relies on idempotence properties of analysis plug-ins to query the database schema for missing results and only schedule jobs for missing results. This means that if plug-ins must re-run for an already calculated result, then the this result must be cleared externally. On the other hand, plug-ins can be non-idempotent; apart from efficient job scheduling, no other practical consequences arise.

Alitheia Core's workflow is not atypical with respect to the workflows defined by other tools. However, Alitheia Core's workflow is extensible via plug-ins, while all steps are automatically parallelizable, provided that dependencies have been satisfied.

5.5 The REST API

The REST API allows accessing the stored metadata and analysis tool results in a hierarchical manner over the HTTP protocol. The Alitheia Core metadata model (see Figure 1) provides the basic entities which are then mapped onto Uniform Resource Locators (URLs). Clients can retrieve lists of resources, or navigate among resource relationships by calling the appropriate URLs. Results can be returned in either XML or JSON data formats. An indicative list of calls supported by the REST API is presented in Table 2.

The REST API enables external clients to use the data without knowledge of the intricacies of the Alitheia Core plug-in mechanism. Fig. 3 presents an example of an AJAX client made possible by the REST API. Using standard Javascript libraries (jquery, jqplot) and HTML5, the client implements a capable metrics browser for

Table 2 Example REST API calls and their result.

Call	Result
`/api/project`	Get list of all projects
`/api/project/2`	Get information on project 2
`/api/project/2/versions`	Get list of versions for project 2
`/api/project/2/version/14/files`	Get list of project files for version 14
`/api/project/2/version/14/files/changed`	Get list of project files that changed in version 14
`/api/metrics`	Get list of installed metrics
`/api/metrics/by-id/8/result/1,3,5`	Get result of application of metric 8 on entities identified as 1, 3 and 5. As metrics know the type of entity they store results against, they interpret the provided entity identifiers as keys to the appropriate result table.

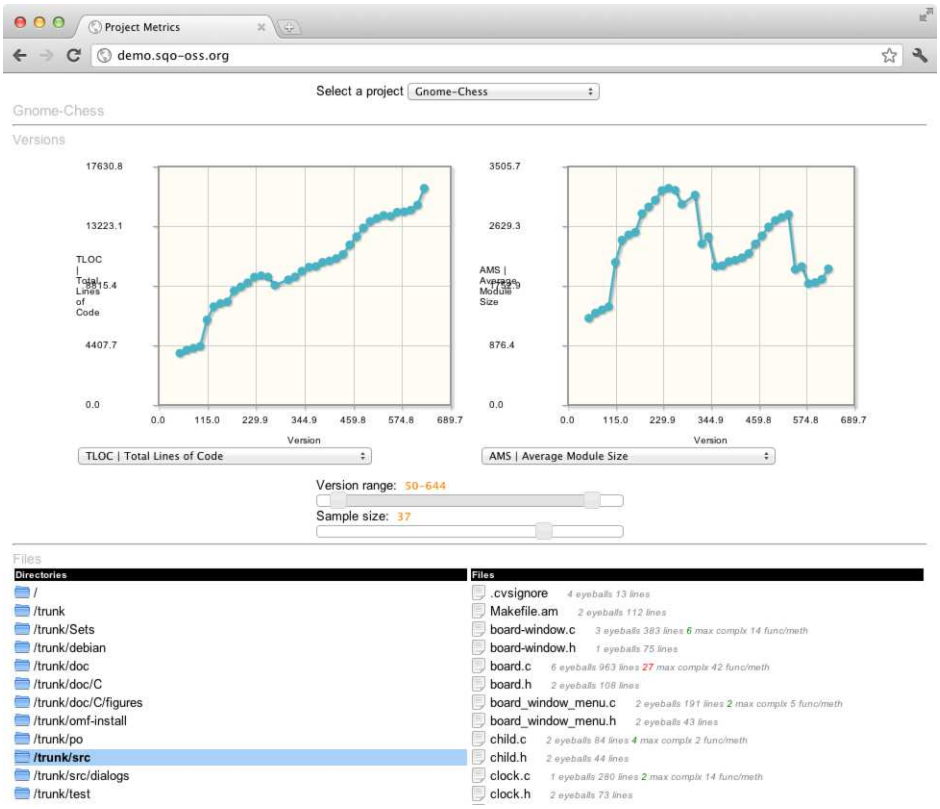


Fig. 3 A client to the Alitheia Core REST API, displaying file and version bound metrics for a small OSS project.

the data offered through our servers in 500 lines. Other types of clients that can be based on the REST API include clients that synchronize data between different Alitheia Core installations, IDE clients that annotate source code based on the results of analysis tool runs, or visualization clients.

5.6 Implementation

Alitheia Core posed a significant implementation challenge, with performance getting from the outset a high priority. Although no special data structures have been developed for Alitheia Core, we took extreme care to use the appropriate containers that would allow for lock free multiprocessing. Also, we took advantage of the multiversion concurrency control (MVCC) capabilities (Bernstein and Goodman 1983) present in modern databases to enable contention free execution at all performance critical paths of Alitheia Core, leaving the details of data synchronization to the database. After initial implementation, Alitheia Core performance has been carefully profiled and several changes were made to the data schema and data semantics to make it scalable. Examples of such refinements include the following.

- The incremental approach to data storage, as described in Section 5.1. Similarly to other existing systems, such as Evolizer, the initial implementation stored the full project tree in the database, for each project version. This optimization conserves more than 90% storage space in case of large projects while not being slower to query.
- The augmentation of the RDBMS schema with in memory data structures that facilitate data analysis. For example, it is often useful for plug-ins to browse the project's file tree or filter specific files (e.g. all Java files or all documentation files) before retrieving those from the repository. In-memory checkouts and filtering work by reconstructing the project's filesystem structure in memory, using data from the RDBMS only. For projects with about 10,000 revisions, both operations complete in less than a second, thereby providing very fast access to the required data.
- The fact that workflow and metric plug-ins can add their custom entities to the schema. This allows for efficient joins between data formats that the plug-ins understand and generic entities that Alitheia Core offers. The initial implementation expected plug-ins to store all custom information in a common key-value table for all plug-ins, leading to suboptimal query performance and more complex query behaviour, because joins had to be performed in the plug-in code.

The result is a system whose performance characteristics are mainly limited by external factors, namely the availability of computing resources, the performance of the employed RDBMS and the algorithmic complexity of the experimental process.

The complexity of Alitheia Core proved to require a substantial amount of implementation effort. Alitheia Core is under development for more than three years and currently consists of more than 28 thousand lines of executable code (KLOC). From those, 5.4 KLOC are devoted to the data model and operations on it, 2 KLOC are test cases and 1.5 KLOC are framework overhead (namely, OSGi activators). Metrics, data plug-ins and parsers sum up to 10 KLOC. The Alitheia Core design went under two major revisions that had a significant impact on reducing the number of lines in the system: during the first, we replaced the web services stack then employed to retrieve data from the core and the corresponding frontend with a functionally equivalent lightweight combination of a REST API with a Javascript-based web front end. This reduced total code footprint by more than 40%. In the second revision, we replaced boilerplate analysis plug-in setup code

with Java annotations and an annotation processor class. This reduced code by more than 10% and simplified the analysis tool implementation.

Alitheia Core benefits from the extensive use of existing mature OSS components: Equinox as the OSGi platform, Hibernate for ORM, log4j for logging, Resteasy for the REST API, Jetty as the embedded web server and several libraries for accessing the raw data sources. From the total 21MB of the Alitheia Core binary distribution, only 0.8MB represents code developed for Alitheia Core, a metric indicative of the value that third party components bring to Alitheia Core.

6 Evaluation

In this section, we provide evidence on Alitheia Core's ability to answer the main challenges researchers face, as presented in Section 2 and to address the requirements set forth in Section 3. Initially, we compare Alitheia Core's features to other existing tools presented in the literature. We also evaluate the platform's performance and demonstrate its scalability using data from real world projects. Finally, we present the execution of an example case study using Alitheia Core.

In the descriptions below, we frequently refer to our project data mirror. The mirror contains data from more than 700 OSS projects, from various OSS project repositories, such as SourceForge.net, and the Gnome ecosystem along with large self-hosted projects, such as KDE and FreeBSD. Not all available data is mirrored for all projects, but for most projects, the mirror contains up to date versions of the source code repository, archived and relatively current versions of mailing lists and, for more than half of the projects, archived versions of their bug report databases, up to early 2009. Also, the mirror does not currently contain any data from Git repositories, because when it was assembled, Alitheia Core lacked Git support. The data size of the project mirror is currently 290GB. An overview of the distribution of the sizes for all projects in the mirror can be seen in Fig. 4.

6.1 Comparison with Existing Platforms

In Table 3, we present a feature comparison of tools that were identified as MSR platforms in Section 4. The comparison is split into six categories, which correspond to the high level platform requirements we identified in Section 3. Specifically, we examine the services each platform offers to extension writers, the data analysis facilities supported by each tool and the interfaces offered. We also examine peripheral features, such as the scaling mechanisms each platform offers and its interfaces.

On the data sources front, all platforms offer support for acquiring data from a source code version control system. Typically, this support includes the file tree versioning paradigm offered by Subversion and CVS. The content management paradigm exposed by Git is a radical departure (Bird et al 2009) from file trees and requires significant changes to the data model each tool supports. Alitheia Core's data schema has been recently extended to support Git. Most tools also offer bug data import, exploiting the ubiquity (in the open source world) of the Bugzilla data schema. Only Hipikat and Alitheia Core support importing of emails, albeit each for a different purpose; Alitheia Core's mail support is generic and aims to act as

Table 3 Platforms used in empirical software engineering research and their features.

Platform	Hipikat	Kenyon	Moose with Hismo	Evolizer	SPO	Churra-sco	Alitheia Core
Data Sources							
Centralized SCM	CVS	CVS, SVN, Clearcase	CVS	CVS, SVN	SVN	SVN	SVN
Distributed SCM	—	—	—	—	—	—	GIT
BTS	Bugzilla	—	—	Bugzilla	Bugzilla	Bugzilla	Bugzilla
E-mails	✓	—	—	—	—	—	✓
Others	text files	—		—	—	—	—
Data acquisition support	—	—	—	—	—	✓	—
Services							
Pluggable workflows	—	✓	—	—	—	—	✓
Analysis plug-ins	—	✓	✓	△	—	—	✓
External tool wrappers	—	✓	—	—	—	—	✓
Data plug-ins	—	✓	✓	—	—	—	✓
Source code parsers	Java	—	SmallTalk, Java, C++	Java	—	△	Java, Python
Automated tool invocations	—	✓	—	—	—	△	✓
Data Analysis							
Intermediate representation	✓	✓	✓	✓	✓	✓	✓
Link BTS issues to commits	✓	—	—	✓	—	✓	—
Link emails to commits	✓	—	—	—	—	—	—
Link emails to BTS issues	✓	—	—	—	—	—	—
Developer identity resolution	?	—	—	✓	?	?	✓
Fine-grained change analysis	—	—	—	✓	—	—	—
Source code meta-model / versioned?	—	—	FAMIX/✓	FAMIX/—	—	FAMIX/?	Custom AST/✓
Scaling							
Multi-project support	—	✓	—	—	✓	✓	✓
Computational model	—	Concurrent processes	—	?	?	?	Scheduler, Jobs
Multicore support	?	✓	—	—	?	✓	✓
Cluster Support	—	△	—	—	?	?	△
Interfaces							
Administration Interface	—	—	✓	—	—	?	✓
REST API	—	—	—	—	—	—	✓
Analysis results GUI	IDE integration	—	✓	—	web-based	web-based	web-based
Visualizations	—	—	✓	✓	✓	✓	—
Availability							
Source code	—	—	—	—	—	—	✓
Executable code	—	—	✓	✓	—	—	✓
Datasets	—	—	—	—	—	—	✓
Documentation	△	—	✓	△	—	△	✓
Demo installation	—	—	—	—	—	✓	✓
Data current at: 17/10/2012							
Source	Cubranic et al (2005)	Bevan et al (2005)	Nierstrasz et al (2005)	Gall et al (2009)	Lungu et al (2010)	D'Ambros and Lanza (2010)	

— No support, △ limited or partial support, ✓ full support, ? No information available

a source of process information, while Hipikat indexes email contents in its search engine to provide developers with context oriented help. Finally, despite most tools work with OSS data, only Churrasco includes support for retrieving them from the original repository, thereby potentially simplifying analysis; all others expect a local data mirror.

Moving to services and data analysis support, a common theme is the support of intermediate representations at various granularity levels. Some platforms can also process source code and extract its structure to either the Moose-oriented FAMIX metamodel or to a custom AST in the case of Alitheia Core. The advantage of the FAMIX metamodel is that the representation can encapsulate several object-oriented languages in the same intermediate format; it is however open to investigation whether it can also support functional or mixed functional/object-oriented languages (e.g. Scala) whose use has been increasing. A notable mention is Evolizer, whose change distilling capabilities (Fluri et al 2007) can help researchers identify changes on the AST level. Alitheia Core is does not offer as comprehensive support as other platforms for linking data among data sources, even though no part of its architecture precludes the implementation of such analysis.

An important part of all software platforms is their support for external tools. Alitheia Core, being designed from the ground up as an integration platform, offers the most comprehensive feature set on this front. It can wrap existing tools (see next section for an example), automatically parallelize their invocations using its job based abstraction and store their results in its internal database, using simple result wrappers. It also enables external tools to access its internal database remotely, using a REST API. All other platforms rely on tools designed specifically for them.

Summarizing, Alitheia Core offers stronger platform features than competing tools, but it lacks the built-in analysis algorithms other platforms offer. Specifically, Alitheia Core provides comprehensive data source support, including beta-quality support for distributed SCM systems, an extensible analysis workflow, while being fully open and documented. All source code related resources in Alitheia Core have attached versions stemming from commits and metrics can be attached to them. Alitheia Core also offers a computational model, which it exploits to automatically parallelize the submitted workloads. The open-ended architecture of Alitheia Core allows missing data integrations to be implemented as plug-ins to the analysis workflow, and depending on their nature, to use the versioning capabilities of the data model. Alitheia Core is lacking on the results visualization front, even though the REST API can be used as a rich data source for visualization clients.

6.2 Performance and Scaling

Import Performance To illustrate the ability of Alitheia Core to handle large datasets, we conducted an analysis with five OSS projects of increasing sizes: JConvert, JodaTime, Gnome-VFS, Evolution and FreeBSD. We run the analysis on our reference hardware platform, the specifications of which are listed in Table 4. The examined projects were selected from our project data mirror based on their cumulative data sizes and their placement in the size distribution chart (see Fig. 4), to test the performance properties of Alitheia Core. We imported the projects starting from smaller (JConvert) to bigger (FreeBSD). To simulate realistic work-

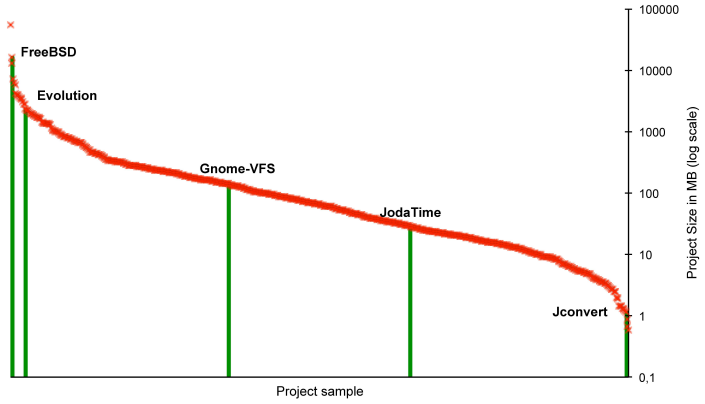


Fig. 4 Project size distribution for projects in our project data mirrors. Marked are the projects selected for benchmarking.

Table 4 Hardware and software configuration for the machine used for measuring performance

Item	Description
Computer	Custom-made 4U rack-mounted server
CPU	4 × Dual-Core Opteron, 2.4GHz
RAM	16 GB 400 MHz with ECC
Raw data storage	6 × 500GB, SATA II, 7.2k RPM, hardware RAID 10
Database storage	4 × 300GB, SATA II, 10k RPM, hardware RAID 10
Operating System	Debian Linux stable 5.0, kernel 2.6.26-1-amd64
Database	MySQL 5.1.53 64-bit
Database Configuration	InnoDB engine, 7GB buffer pool, no binary logging, read committed transaction isolation
Alitheia Core Configuration	16 worker threads, 3GB heap

Table 5 Key size metrics and time required to process data from various OSS projects

Project	JConvert	JodaTime	Gnome-VFS	Evolution	FreeBSD
Size					
Num. Revisions(MB)	220 (2.3)	1,570 (27)	5,551 (207)	25,248 (1,305)	180,332 (5,868)
Num. Emails(MB)	—	—	3,102 (33)	79,722 (1,256)	1,879,058 (13,364)
Num. Bugs(MB)	—	—	2,167 (18)	60,624 (739)	—
Total data size (MB)	2.3	27	258	3,300	19,232
Processing Results					
Total File Revisions	624	11,042	25,119	169,223	782,940
Num. Bug Comments	—	—	13,301	226,357	—
Uniq Developer Ids	1	10	2,217	35,092	89,309
Performance					
Total time to import	0:00:08	0:02:20	0:20:40	2:02:12	22:48:32
Time to import SCM	0:00:08	0:02:20	0:20:40	2:02:12	22:48:32
Time to import emails	—	—	0:00:18	0:09:20	2:37:18
Time to import BTS	—	—	0:00:30	0:47:06	—
Revisions/s (MB/s)	27 (0.28)	11.2 (0.22)	4.44 (0.17)	3.46 (0.17)	2.20 (0.08)
Emails/s (MB/s)	—	—	172.2 (1.8)	142.3 (2.2)	199.2 (1.4)
Bugs/s (MB/s)	—	—	72.3 (0.6)	21.4 (0.26)	—
MB/s	0.28	0.19	0.20	0.45	0.25
GB/hour	1.08	0.69	0.70	1.62	0.89

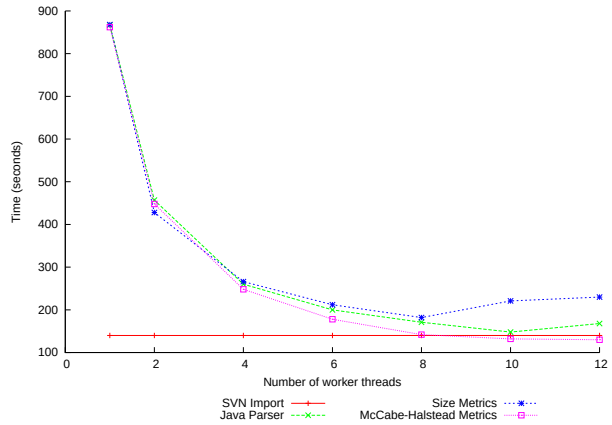
load conditions, we did not clean the database after each project was processed; should we have cleaned the database, the results would have been better as virtual memory pressure exerted by the database to the operating system would have been minimized (at least in our MySQL-based setup). As a consequence, if the report order was reversed, the results would have been slightly better for bigger projects. Finally, we only evaluate the second stage of the data processing workflow; specifically, we run the plug-ins svnmap, bugmap and mailmap (see Table 1).

The results of these experiments can be seen in Table 5. From the projects we analyzed, both Evolution and (especially) FreeBSD lie at the top of the size scale in the OSS project ecosystem. Specifically, 93% of the projects in our mirror have a data footprint smaller than Evolution. The time to complete the metadata import is in all cases dominated by the time to process SCM entries. This result stems from the fact that the Subversion SCM system is used by all projects in our dataset. Subversion uses an event sourcing architecture that stores revision differences incrementally, and therefore it is not straightforward to process them afterwards in parallel. On that front, the Git SCM design represents good opportunities for optimization, as branches, which are very common when developing with Git, can be processed in parallel, until a merge point is reached.

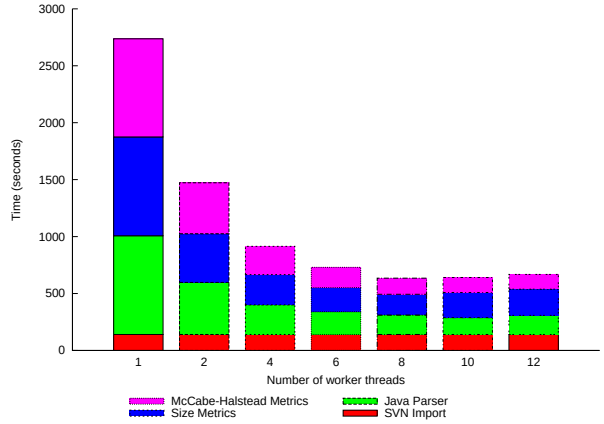
On the other hand, emails and bug reports are independent from each other, so Alitheia Core can process those concurrently. Combined with the fact that processing emails and bugs is a relatively simple operation as no change history is maintained for either entity, concurrent processing makes a big difference in the throughput rate: in the case of Evolution, emails can be processed more than 10 times faster than data from SCM repository. This difference in throughput also explains the inconsistencies in the scaling of the overall data processing rate. If the time needed to process data in parallel (email and BTS issue processing) is close to the time required to process SCM data then the overall throughput will be correspondingly higher.

The overall results show that the data import performance in Alitheia Core degrades gracefully as data sizes increase. The best performance, in terms of data processed per unit of time, is demonstrated on medium to large datasets. This happens because those projects have very large email and bug datasets, which can be processed in parallel, so the overall throughput increases.

Scaling To evaluate the scaling capability of Alitheia Core, we conducted an experiment where we varied the number of worker threads, while processing the same project using the same pipeline. Specifically, we selected the JodaTime project for which we processed 1,570 Subversion revisions (11,042 file changes), then used the Java parser to extract method-level change information and finally run the project size and code structure (McCabe and Halstead) metrics on. Apart from the Subversion import phase, all other tools can be run in parallel. The runtime behaviour of all employed analysis tools contains both several I/O sessions, to read raw data from the project's repository and to query the database, and a computation operation to calculate the metric results. This means that since analysis tool executions are mapped onto worker threads, the operating system should be able to interleave execution I/O intensive parts with CPU-bound parts during all metric runs. Alitheia Core was run on our reference host (see Table 4). To ensure that Alitheia Core will not compete from resources with the database server, we run the database server on another machine (2 CPUs, 2 GB RAM, high performance



(a) Time per processing phase



(b) Contribution of each processing phase to the total time

Fig. 5 Time to execute various processing phases versus the number of worker threads used. The test machine has 8 physical cores.

I/O). The chosen configuration was deliberately underpowered to demonstrate the effect of external tools on the performance of Alitheia Core.

The results can be seen in Figure 5. Overall, by increasing the number of cores, the total time required for analyzing a medium repository decreased by 76% at a scaling ratio of 4.3 times. The individual tools scaling behaviors varied according to their dependence on the external database. For example, the size metrics plug-in which calculates metrics such as number of files per version by means of aggregation queries, saturated the database server at 8 worker threads. On the other hand, the Java parser tool, which only used the database to store final results improved its performance up to 11 concurrent threads. Nevertheless, all tools benefit from the automatic workload parallelization of Alitheia Core's design, effectively decreasing the required execution time by 72%–82%.

An interesting finding of the analysis is that the size metrics plug-in executes slower than the Java parser plug-in. While this may contradict common sense,

Table 6 Comparing Alitheia Core against simple invocations of external tools.

Project	JConvert	JodaTime	Gnome-VFS	Evolution	FreeBSD
sloccount					
Alitheia Core	0:00:06	0:03:02	0:09:35	5:16:24	72:35:45
sloccount	0:03:32	0:33:12	3:05:32	59:16:12	>400 hours
Speedup	94%	90%	84%	89%	> 80%
FindBugs					
Alitheia Core	0:28:12	0:14:15	—	—	—
FindBugs script	1:07:33	00:55:51	—	—	—
Speedup	63%	72%	—	—	—

given that parsing is considered a non-trivial task, it is actually a result of implementation choices of the corresponding plug-ins and optimizations enabled by Alitheia Core. Specifically, the Java parser has been implemented using a hand-tuned Java language grammar using the ANTLR parser generator (Parr and Quong 1995); while the constructed abstract syntax tree is accurate enough to calculate most known metrics, it skips costly steps such as type resolution. Moreover, using Alitheia Core facilities, the Java parser plug-in can select the Java files that have changed in every specific revision. This leads to a very small number of files being parsed very fast; on average each file takes less than 100ms to be parsed on our reference server, while an average revision is processed in less than a second. On the other hand, the size metric plug-in uses a complicated database query (including a 5-table join) to calculate aggregate project size counts per revision; this leads to significant load on our MySQL database. Using a database that handles complicated joins more efficiently or more capable database server hardware could improve the size metric plug-in results.

Use of External Tools To demonstrate how the incremental tool invocation and automatic workload parallelization approach Alitheia Core takes pays off, we compared Alitheia Core against a simple script that loops over all revisions of a provided repository and invokes an external analysis tool. We selected two tools to compare Alitheia Core against, namely sloccount (Wheeler 2010) and FindBugs (Hovemeyer and Pugh 2004). The sloccount tool processes a source code tree and calculates the number of executable statements for many known programming languages. On the other hand, FindBugs processes Java bytecode and identifies potential bugs by comparing it against a predefined list of know bug patterns. Both tools require a project checkout; in addition, FindBugs requires each project revision to be built first. Fortunately, the two Java projects we examined contained ant and maven build scripts, so each revision could be build using the respective build tool. To invoke the tools, we developed a script that performed an initial project checkout at revision 0, run either sloccount or FindBugs, and then updated to the next available revision. In the case of FindBugs, it also invoked maven or ant first. If the build failed (due to compilation failures or missing dependencies), the version was skipped from analysis through FindBugs. In all cases, the measurements where taken from the trunk directory.

Within Alitheia Core, we used the project size metrics plug-in to compare against sloccount. Both the Alitheia Core plug-in and sloccount already calculated roughly the same metrics, namely the number of executable lines, the number of comments, the number of files and their aggregations per project version. In the

case of FindBugs, we created a new analysis tool plug-in that checked out (using the Alitheia Core facilities) a revision, run maven or ant, and if the build was successful, it then run FindBugs, parsed the results and stored them in the database. The plug-in was implemented in 500 lines of code, the majority of which where devoted to starting, monitoring and reading the output of external tools. Both experiments where run on our reference host, using the configuration presented in Table 4. For this experiment, Alitheia Core was configured with four worker threads.

The results can be seen in Table 6. Independently of the size of the project it was tested with, Alitheia Core finished the provided task in less that 80% (sloccount) and 63% (FindBugs) of the time it took for a competing simple shell script to run. This outcome is the result of optimizations at various levels of the workflow. In the case of the line counting experiment, file contents are never written to disk; instead, when a plug-in requires access to the contents of a file in a specific project revision, Alitheia Core will retrieve it directly from the repository into a memory buffer. Even though, as described above, the aggregation of size metric results per version is done in a suboptimal manner, the fact that no files need to be read from disk leads to a significant performance improvement. Indeed, for revision 1000 of the Jodatime project, sloccount requires two seconds to calculate its results. The corresponding query executes in less than one second on our database, while four such queries can be run in parallel without affecting database performance.[1]

In the case of FindBugs, an additional optimization originates from the fact that the plug-in can query the database to verify whether a specific version contains a Maven/Ant build file before checking it out on the disk. This enables Alitheia Core to quickly skim through the earlier versions of Jodatime, where an older, incompatible version of the Maven tool was used.

Assessment The results indicate that Alitheia Core can handle very large projects on modest, by today's standards, servers. At an average rate of 1GB/hour and with an average project size of 500MB, one can import metadata from about 500 projects in 10 days on a single machine, a process that only needs to be performed once. Various types of analysis can then be performed on or with those metadata and the performance will mostly depend on the number of total available processors. Due to the automatic distribution of the workload on multiple processors, any analysis will take on average significantly less time on Alitheia Core than by performing the same experiments using simple shell scripts, irrespective of whether the analysis tools were developed specifically for Alitheia Core or not. In turn, this indicates that it may be beneficial for tools to be written for or ported to Alitheia Core. As the scaling experiment shows, in its current form, Alitheia Core's performance and scaling is mainly restricted by the performance of the database used.

In the following example, we also present evidence of Alitheia Core working on multi-project datasets, using a small cluster of four machines.

[1] Recall that Alitheia Core was configured with four worker threads, thereby leaving 4 idle cores to MySQL.

6.3 Case Study Example: Development Teams and Maintainability

Software maintenance is a crucial part of the software life cycle. It is widely reported that maintenance operations can take up to 70% of the total costs of the software (Boehm 1987). Consequently, maintainability is one of the top-level criteria in the ISO software quality model (ISO/IEC 2004) and its assessment has been the topic of study of numerous works (Oman and Hagemeister 1994; Deursen and Klint 1998; Muthanna et al 2000; Heitlager et al 2007).

As software development is primarily a human oriented task, one would think that development teams play a significant role in affecting maintainability. Intuitively, one might expect that teams staffed with highly skilled individuals will produce higher quality and, consequently, more maintainable code. A question that arises is whether peer-pressure among colleagues can affect maintainability. For example, Raymond (2001) has argued that the OSS development model can tame complexity through shared increased source code awareness by developers and end users. Does this improvement reflect itself in software maintainability? Is the size of the team working on the same project, and the consequent applied peer pressure, an enabling factor for producing maintainable software? The counterargument based on the non-linear scaling of the communication overhead in large projects (Brooks 1975) is also valid, especially in the eyes of practitioners. One would argue that the more people work on a module, the worse its maintainability will be, as miscommunicated ad-hoc modifications and spurious coding practices would prevail over organized, clean, structured code.

The objective of this case study is to present how Alitheia Core can simplify large-scale research through the examination of the potential correlation between team size and software maintainability. For that, we formulate the following two simple hypotheses:

H1 Team size correlates with maintainability at the project level.
H2 Team size correlates with maintainability at the source module level.

6.3.1 Method of study

To evaluate maintainability, we use the Maintainability Index (MI) metric (Oman and Hagemeister 1994; Welker and Oman 1995), a composite metric that attempts to quantify maintainability by combining several low level measurements such as Halstead's Volume (Halstead 1977), McCabe's Extended Cyclomatic Complexity (McCabe 1976), the number of lines of code and number of lines of comments. The metric is a result of statistical analysis on metric data from large C and Pascal language projects. The suggested method of calculation is on source code directories (Oman and Hagemeister 1994) even though it can also work at the project level. As with any composite metric, the MI metric has faced criticism (Luijten et al 2010), while other similar polynomial (Muthanna et al 2000) or hierarchical (Heitlager et al 2007) models have emerged. Therefore, we only consider the MI metric adequate for the purpose of demonstrating the functionality of Alitheia Core.

To implement the MI metric, we used Alitheia Core's compositional approach to building analysis tools. Consequently, we broke down the implementation into four plug-ins, each of which produces measurements that can be stored and retrieved

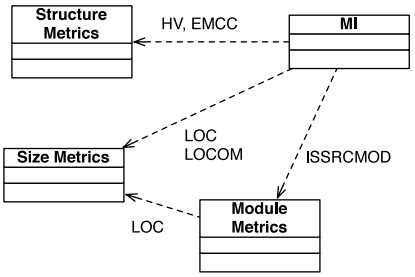


Fig. 6 The Maintainability Index metric plug-in dependencies on other metric plug-ins.

independently (see Fig. 6). The structure metrics plug-in implements the full set of McCabe and Halstead metrics for C and Java, at the individual source file level. At the time the study was done, Alitheia Core lacked proper parser functionality and therefore metrics were implemented using ad-hoc parsers, based on regular expressions. The size metrics plug-in calculates various size metrics, such as the number of executable statements (or lines of code — LOC) and the number of lines of comments (LOCOM), at both the source file and project-wide level. The module metrics plug-in uses the results of the file level size metrics to calculate module sizes. As C lacks direct namespace declarations, we considered a module to be a directory with source code files. The module metrics plug-in also marks directories as source directories if they contain any source files (sets the ISSRCMOD metric to one). If one or more source code files contained in a module change, then a new module revision (and its measurement) is added to the result set.

All that is left for the MI metric to do for each project version is to retrieve the source code directories by querying the module metrics plug-in, and for each one of them to retrieve the metric formula's four measurements required by querying the corresponding plug-ins. Alitheia Core offers abstractions that hide the complexity of querying metric results and browsing version files and directories behind simple method calls. These help reduce the code required for implementing the MI metric plug-in to less than 250 source lines of code. This number includes the calculation of the metric per module and its aggregation per version. As a comparison, command line tools used to calculate MI in two other studies (Samoladas et al 2004; (Spinellis 2006, Section 7.1.1)) are more than 1500 lines long.

To calculate the number of developers per resource, we also implemented a new plug-in that queries the database for the number of developers that were active in a time window of one, three and six months. As active, we considered all developers that committed to the SCM repository at least once in each corresponding time window. We chose to look at specific window sizes, because, in common with other studies (Anvik et al 2006), looking for developers active within a small time window provides a realistic measure for the churning developers contributing to an open source project.

We run the described metrics on a dataset comprising of 142 projects, whose primary language (in terms of lines of code in their latest revision) is C. For projects containing mixed C and other languages source files, the metrics were only applied on the C portion of the code base. Prior to running the metrics, the full history of the projects was imported into the database. This comprised a total of 588,241 project versions and 7,516,327 file versions, including 607,390 module versions. We

Fig. 7 Query used to extract the results for the maintainability index and number of developers ("eyeballs") metrics per module revision for the project GNOME-VFS.

```
select MI.MODMI, EYEBALL.MODEYEBALL
from
(   select pfm.PROJECT_FILE_ID as ID, pfm.RESULT as MODMI
    from   STORED_PROJECT sp, PROJECT_VERSION pv, PROJECT_FILE pf,
           PROJECT_FILE_MEASUREMENT pfm, METRIC m
    where  pfm.METRIC_ID=m.METRIC_ID
           and pf.PROJECT_FILE_ID=pfm.PROJECT_FILE_ID
           and pv.PROJECT_VERSION_ID=pf.PROJECT_VERSION_ID
           and pv.STORED_PROJECT_ID=sp.PROJECT_ID
           and m.MNEMONIC="MODMI" and sp.PROJECT_NAME="Gnome-VFS"
) as MI,
(   select pfm.PROJECT_FILE_ID as ID, pfm.RESULT as MODEYEBALL
    from   STORED_PROJECT sp, PROJECT_VERSION pv, PROJECT_FILE pf,
           PROJECT_FILE_MEASUREMENT pfm, METRIC m
    where  pfm.METRIC_ID=m.METRIC_ID
           and pf.PROJECT_FILE_ID=pfm.PROJECT_FILE_ID
           and pv.PROJECT_VERSION_ID=pf.PROJECT_VERSION_ID
           and pv.STORED_PROJECT_ID=sp.PROJECT_ID
           and m.MNEMONIC="MODEYBALL"
           and exists (
               select pfm1.PROJECT_FILE_ID
               from PROJECT_FILE_MEASUREMENT pfm1, METRIC m1
               where  pfm1.METRIC_ID=m1.METRIC_ID
                      and m1.MNEMONIC="ISSRCMOD"
                      and pfm.PROJECT_FILE_ID=pfm1.PROJECT_FILE_ID
                      and pfm1.RESULT="1"
               )
           and sp.PROJECT_NAME="Gnome-VFS"
) as EYEBALL
where
    MI.ID = EYEBALL.ID
```

run the analysis on a four machine cluster containing in total 12 2GHz class CPUs, 16 1GHz class CPUs and a combined total of 34 GB of memory. The database was run on our reference server, but shared resources with an instance of Alitheia Core running on the same host. The total working set comprised of about 33 million jobs. Running the size and structural metrics plug-ins proceeded at an average rate of 90 jobs per second. Similarly, the MI plug-in, whose performance mostly relies on database querying speed and executes simple index based queries, calculated results at the rate of more than 120 jobs per second.

6.3.2 Results

To validate our hypotheses, we extracted and correlated measurements from the MI and developer team size plug-ins and analyzed them statistically. As metrics are stored against module or project versions, it is straightforward to extract them with an SQL query, such as the one shown in Figure 7. As Alitheia Core does not currently employ any mechanisms for statistical analysis, the results were extracted from the database into text files.

To analyse the results, we used the Maximal Information Coefficient (MIC) metric from the Maximal Information-based Nonparametric Exploration (MINE) suite

(Reshef et al 2011). This newly proposed bivariate correlation suite of metrics has two important properties: it works well on discovering associations other than linear, and it does not make any assumptions on the distribution underlying the data. Similarly to the most well known tests, MIC reports the strength of correlation between two variables as a number in a range from 0 (no correlation) to 1 (strong correlation). The significance of the correlation is not reported by the MIC algorithm, but can be calculated using matrices; as a rule of thumb, with more than 350 data points, any MIC result greater than 0.28 has $p < 0.01$, while any MIC result less than 0.21 has $p > 0.05$. As MIC can capture relationships that are functionally different from linear, its score can be very different from a Pearson or Spearman correlation test on the same sample; in fact, its equitability properties can be exploited to compare relationships of the similar strength across different functional forms. Apart from MIC, which effectively measures the strength of the correlation, the MINE suite reports the derivation from monotonicity of the correlation (MAS), the degree to which the dataset appears to be sampled from a continuous function (MEV) and the complexity of the association (MCN). MIC has been shown to work well on empirical data from software repositories, especially when sample sizes grow (Posnett et al 2012).

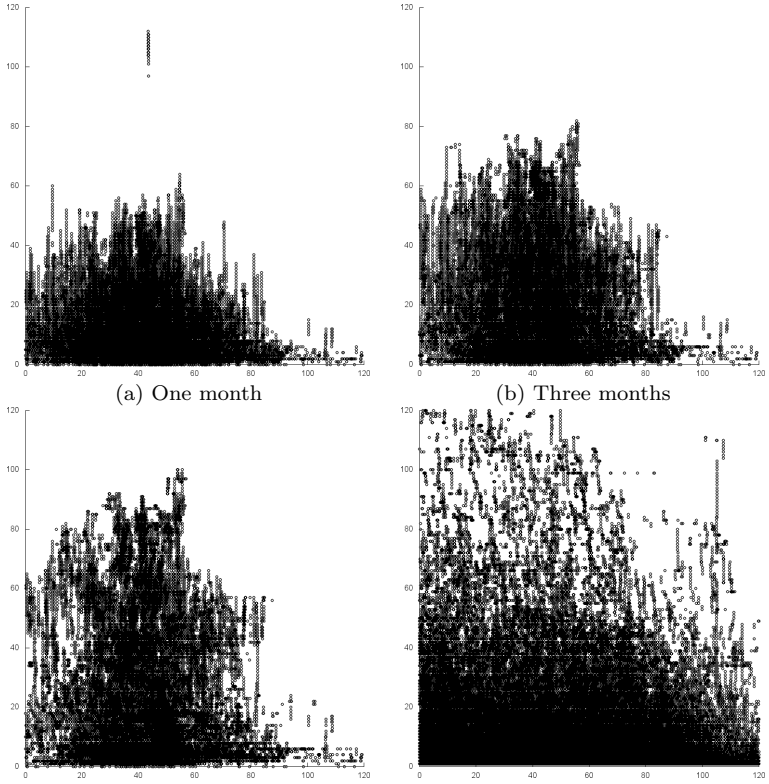
At the project level, we examined the correlation against time windows of one, three and six months within which we determined developer activity by checking if a developer committed a change to the project's SCM. At the module level, we calculated the size of the team for the specific revision of the module as the number of all developers that committed changes in files residing in the specific module. In both cases, we filtered out projects with a single developer. We also filtered out projects with less than 350 data points to obtain statistically significant results. We did not do any further processing of the data (i.e. filtering out developers that committed only few times).

The results of the statistical analysis can be seen in Table 7. Overall and across all projects, we did not find any significant correlations, either at the module or at the project level, so both our hypotheses must be rejected. This result suggests that the number of people working on an open source project does not seem to affect an important aspect of software quality, maintainability, as measured by the MI metric. Consequently, maintainability may be driven by other factors, such as developer competence or processes that encourage adherence to project development guidelines.

6.3.3 Replicating with Other Platforms

To provide a comparative evaluation of Alitheia Core in a real world repository mining scenario, we attempted to replicate the case study described above with an external platform. From the platforms examined in Table 3, only Evolizer and Moose were available online. Our goal was to implement the same metrics as extensions to all platforms, without modifying the platform's internals. To understand how the platforms work, we browsed each platform's source code and used the available online documentation. We describe our experiences for each tool below.

To examine version histories of projects in C, Moose requires the Hismo (SCM import) and inFusion (C parser) tools to be installed. Hismo can attach FAMIX models to the history models it creates. Moreover, basic source code metrics, such as the ones we use, come with FAMIX models. However, extracting models from



(a) One month (b) Three months

(c) Six months (d) Per module

	MIC	MAS	MEV	MCN
Project Level				
1 month	0.35431	0.07141	0.35431	8.189824
3 months	0.36088	0.08216	0.36088	11.40301
6 months	0.36713	0.08847	0.36705	10.42511
Module level	0.33618	0.25779	0.33618	11.60547

Table 7 Correlation analysis between number of developers (horizontal axis) and maintainability index score (vertical axis). All results are at $p < 0.01$.

inFusion and attaching them to Hismo is a manual process, that has to be carried out per each project revision. Also, Hismo does not model history for project files nor does it save developer related information per file revision.

Evolizer does not support attaching arbitrary metrics to file or project versions in its database. Metric implementations in Evolizer rely on FAMIX models, and Evolizer is only able to compute such models for the latest project version and only for Java projects. Moreover, metrics in Evolizer are only evaluated on the fly. This means that in order to save them or link them to specific entities, Evolizer must be modified to use a database similarly to Alitheia Core. Moreover, Evolizer does not feature an automated, workflow-based analysis process. Specifically, due to its deep integration with Eclipse, it requires projects to be checked out within the IDE before being submitted for analysis. Finally, Evolizer could not save more than one project in a particular database.

7 Applicability and Limitations

In the following section, we discuss aspects that affect Alitheia Core's applicability on the problem domain it was designed for.

7.1 Advantages over Existing Tools

By combining metadata abstractions with transparent multiprocessing, Alitheia Core offers a wealth of services to researchers working with data from software repositories.

- Alitheia Core simplifies the task of implementing custom analysis plug-ins or integrating external tools by requiring plug-ins to implement just a single method for each supported data type. The process of creating the necessary files and integrating plug-ins to the runtime is automated through Alitheia Core's build system, while the extensibility points feature stable interfaces and are fully documented.
- Researchers can insert custom processing phases to the data processing workflow. This feature enables custom analysis algorithms to resolve and store links between diverse types of data either across projects or across repositories before the analysis tools are invoked. Using Alitheia Core, a researcher can devise algorithms that, for example, link bug reports to project versions based on the appearance of bug report numbers in a commit number or identify and mark commit operations based on the type of refactoring they induced. Algorithms can be run incrementally on projects, without requiring recalculation of dependent results or later phases of the analysis.
- Alitheia Core allows cross repository and cross project analysis. As data from multiple projects are imported into a single metadata schema, data retrieval operations can be performed either progammatically or by means of SQL statements.
- Alitheia Core features fast analysis turnaround. The time required to execute large case studies is roughly proportional to the size of the analyzed repository, as Table 5 shows. Scaling for tools developed for Alitheia Core is only affected by external subsystem performance, while plug-ins driving external tools can offer a significant performance increase. We argue that fast turnaround and analysis automation is an enabling factor for experimentation, because researchers can develop their analysis algorithms on large datasets and promptly derive feedback from their input.

7.2 Suitability

In developing Alitheia Core, the focus was to enable researchers to leverage data from multiple OSS repositories in a common intermediate format, while also permitting distribution of processing load on multiple processors. As a result, Alitheia Core could be a suitable tool in the following research cases.

Scaling Research Large-scale research is increasingly being deemed a requirement for producing credible studies. Within the MSR research community, Alitheia

Core can be most useful for medium to large-scale quantitative studies, which might benefit from the data integration and workload distribution possibilities provided by the tool. Alitheia Core has been demonstrated in this and other works (Gousios and Spinellis 2009; Kalliamvakou et al 2009) to be able to handle large volumes of data, albeit with externally imposed limits to the volumes it can process.

Reproducibility The issue of reproducibility of research results has received increasing attention during the last years in the field of empirical software engineering (Shull et al 2008; González-Barahona and Robles 2012). Alitheia Core's shared data and code repositories and standardized formats it provides might be a first step toward this goal.

Driving External Tools It is common for researchers to use external command-line tools to process data from repositories. As we have shown in the FindBugs experiment, Alitheia Core can wrap external tools and offer a significant performance increase while making it possible to store the results for future reference. In principle, any tool with idempotent execution can be used efficiently with Alitheia Core.

Software Process Monitoring Even though Alitheia Core was developed in a research context, it can also be used as the basis of an online team performance monitoring and software quality analysis process. External analysis and visualization tools can use Alitheia Core's REST API to retrieve data, while Alitheia Core's processing engine can be connected to the team's repositories and can also be extended to custom data types. Such a setup could potentially achieve the benefits of project telemetry (Johnson et al 2005), without requiring changes to the developer's tooling. In previous work (Kalliamvakou et al 2009), we have shown that it is possible to use software repositories to extract team performance indicators, while in this work we presented an example of measuring maintainability over very large code bases.

On the other hand, Alitheia Core may not be suitable for all kinds of analyses involving software repositories. Some cases where Alitheia Core might not be a suitable tool are listed below.

Experiments Research with software repositories is usually done post-hoc. A researcher can thus not vary parameters of interest in a controlled way to study cause-effects, i.e. conduct experiments. Although large-scale case studies can reduce sampling bias, researchers can find a large number of correlations without having contextual information to investigate whether the correlations represent cause-effects or are spurious. This is a known limitation of the field, and also one affecting research being done with Alitheia Core.

Qualitative Studies While qualitative studies are certainly possible within the field of empirical software engineering (Seaman 1999), they are not very common (Glass et al 2002; Sjøberg et al 2007). Alitheia Core does not include support for conducting qualitative studies, for it does not provide the appropriate qualitative study abstractions, such as questionnaires or interview tabulation.

Data Analysis Alitheia Core does not include any mechanisms for statistical data processing. While it is possible to include a statistics or data mining tool as a late stage analysis plug-in, the default implementation does not yet offer such functionality, possibly making the results analysis process more involved.

Moreover, Alitheia Core does not include any provisions for visualizing data, delegating the task to external tools.

7.3 Data Integrity

As Alitheia Core integrates data from various sources, processes them and produces metadata, an issue that emerges is that of data integrity; how can we ensure that the data Alitheia Core produces is accurate? The question has not been explored in depth by the repository mining community, but is critical for trusting the results of empirical studies.

In data processing applications, data integrity should be scrutinized at locations where data transformations occur. In the context of Alitheia Core, most data transformations occur in three cases: i) when importing raw data to the metadata database, ii) when inferring relationships between the metadata and iii) when metrics are run. In the first case, as the metadata is merely a stripped down version of the raw data,[2] comparing the two representations can be fully automated through a test suite, and this test suite can be both exhaustive and very accurate. The later two cases apply more complex transformations, such as combinations of selections, aggregations or joins, and consequently their outcome is in principle not traceable to the original data. While automated unit tests can be helpful in isolating problems during development, they cannot guarantee the transformation correctness, and therefore must be augmented by human inspection. We believe that the inspection can become a community effort through a web site that enables researchers to rate the data processing accuracy.

During the development of Alitheia Core, automatic validation of the processed data was, regretfully, not a major concern. However, having learned from our painstaking experiences with manual validation, we are now following an automated approach in the development of newer subsystems. For example, for the development of the Git plug-in, we have created a test harness suite that automatically imports to the Alitheia Core database and compares to the source (both revision and file tree entries) any given Git repository. This approach not only allowed us to find bugs in our code early on, but also revealed a serious bug in the Git repository access library we use.

7.4 Privacy

While the participation of developers to OSS projects does not come with privacy guarantees, the mass processing and the interconnection of data from multiple projects has begun to raise eyebrows in the repository mining community (Robles and Gonzalez-Barahona 2005; Godfrey et al 2009). In common to most other repository analysis systems, Alitheia Core does not take any measures to protect the developers' identities. However, given the so-called "social coding" and search facilities provided by modern repository hosting sites, like GitHub, we feel that targeted compromises of developer privacy are considerably more likely to occur

[2] More formally, the transformation is an injective one to one function with domain set the raw data and target set the metadata.

through such sites than through platforms that aggregate repository data, such as Alitheia Core.

8 Lessons Learned

The main lessons we learned from developing and applying Alitheia Core concern the value of large-scale empirical research and the many issues that arise from trying to perform it on large, real world datasets. Researchers are already aware of the former (Zelkowitz and Wallace 1997; Sjøberg et al 2005; Zannier et al 2006); in this section, we present what we learned on the latter.

8.1 Efficient and Robust Data Processing

The most important lesson we learned is that when dealing with large volumes of diverse data, naïve and brute force approaches to data analysis do not work. Before processing large quantities of data, a researcher must fully understand their format and, especially, their interactions. This knowledge is required for designing efficient storage schemata, extracting parallelism from common operations on data or for computing the execution order of calculations. Knowledge of the data can be acquired by building prototypes or by studying other analysis tools. Moreover, even in cases where data formats are standardized, processing algorithms will almost certainly need to handle special cases. As an example, one of the problems we have dealt with on that front, concerned email message processing. The format for email messages is documented in the RFC-5322 (Resnick 2008) *de facto* standard. However, a number of email clients, interpret the standard liberally when formatting several headers (notably, the Date and References headers). Such issues complicate implementation and hinder application of generic data processing algorithms, and must be expected from researchers.

8.2 Scaling

Another important lesson that we learned is that scaling is the result of good system design, not optimization. For example, in early prototype versions of Alitheia Core, components used uncooperative threads for implementing parallelism. The problems with this approach were that the machine became easily saturated due to the threads competing for resources and the fact that errors could not be isolated and correlated with the input data. Higher-level abstractions, such as jobs, workers and task queues, allowed us not only to take full advantage of machine resources but also to distribute the load to multiple machines.

On the other hand, researchers aiming to scale their analysis algorithms to thousands of projects should be aware that relying on relational databases will be a roadblock rather than an enabling solution. At a medium scale (up to 200 medium size projects), a relational database would indeed help a researcher import, extract and manipulate data at an adequate rate. However, the database layer in a tiered architecture is usually the most difficult and most expensive resource to scale, and its performance profile is typically a black box for the developer. In

our experience, the key to database performance is to limit the amount of data manipulation the database has to do. For example, complex aggregation queries can be replaced by range queries and data filtering at the requesting node site. Even so, the load exerted for a cluster of machines can easily saturate even the most powerful database server. Distributed database clusters or data warehousing might be possible solutions (Pavlo et al 2009) on that front.

Looking forward, we believe that, in common to other big data processing systems, new generation software repository analysis systems might gain by being designed on top of distributed, schema-less data stores. Such data stores can automatically distribute storage, retrieval and data filtering operations on multiple nodes on a cluster, effectively distributing the processing load at the expense of atomicity, consistency, isolation and durability (ACID) guarantees (Bunch et al 2010). While ACID properties are critical to ensure consistency in concurrent read-write transaction processing workloads (Pavlo et al 2009), most data analytics tasks depend on read-only performance, where schema-less data stores excel (Thusoo et al 2010). Work has already started by others in evaluating software repository analysis with schema-less databases (Shang et al 2011).

8.3 Failures

A high volume application should be designed to isolate the effects of errors and recover from them, not to anticipate and correct them. It is usually not possible to estimate all possible system states when designing the system, and moreover it is not economical (in terms of development time or system performance) to develop workarounds within the system. If an unexpected error occurs, the system must be able to stop processing before errors reach persistent data stores and to resume processing from known good states. Alitheia Core has been very successful on that front, as a result of combining the thread pool processing pattern, transactions by default on all database operations and extensive, multi-level logging. The thread pool pattern allows us to split the workload into small pieces, each associated with a uniquely identifiable data chunk (a project version, an email message, etc.). If processing fails, the event along with the failing data entity is recorded. This allows the researcher to examine the precise conditions, such as data format errors and missing pre-requisites, that caused the failure. Processing failures do not affect the database, because the use of transactions ensures that the database can be rolled back to a consistent state.

9 Conclusions and Future Work

As with any empirical science, research in software engineering aims to understand the behavior of existing systems in order to extract models, theories and best practices that can be generalized and applied to similar occasions. Currently, most research efforts in software engineering are using data from OSS projects, on a relatively limited scale. This fact leads to conclusions that at best cannot be generalized and at worst may be wrong. Furthermore, the lack of standardized experimentation toolkits renders cross validation of the published research results almost impossible.

Alitheia Core was designed from the ground up to enable researchers to design and conduct large-scale quantitative research. The platform can readily process hundreds of projects, with data storage system performance being currently the only obstacle toward scaling it in the order of thousands. In addition to validating our approach, our work on Alitheia Core indicates that, given the appropriate data abstractions, large-scale experimentation is a practical proposition. Alitheia Core forms part of an ongoing research effort to improve empirical software engineering research studies through standardization on experimentation platforms and datasets. Alitheia Core as a tool can be improved on a number of fronts: one obvious improvement is the provision of workflow plug-ins for intra-repository data linking. The results querying interface is spartan in its current form, which, in our view, limits the potential of Alitheia Core as a data analysis tool. What is needed is a graphical user interface that will allow for hierarchical graphical exploration of the data and enable the construction of interesting data visualizations. Also, Alitheia Core's metadata and results storage subsystem has begun showing its limits; currently, we are investigating alternative ways to remove our reliance on SQL databases, without breaking the data abstractions we have built. Finally, the move of many open source projects to new data-rich repositories, such as Github and Bitbucket, presents unique opportunities in standardizing the data acquisition and processing process, an important step to achieving scale. On this front, we are working toward making Alitheia Core compatible with the exposed data formats.

Acknowledgments and Availability

The authors would like to thank the anonymous reviewers for their many excellent suggestions for improving this paper and Alitheia Core. Mirko Böhm and Vassileios Karakoidas have contributed code and ideas over the years that helped improve Alitheia Core. Panos Louridas, in addition to several contributions, implemented the Java and Python parsers. Martin Pinzger made Evolizer available for evaluation.

The full source code for Alitheia Core is available at: `http://www.sqo-oss.org`. A snapshot of the data used to conduct the case study presented in this work can be found at: `http://istlab.dmst.aueb.gr/~george/sw/`. The raw project repository mirrors in the format Alitheia Core expects can be found at: `http://ikaria.dmst.aueb.gr/repositories`.

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.

References

Alpern B, Attanasio CR, Burton JJ (2000) The Jalapeño virtual machine. IBM Systems Journal 39(1)

Anvik J, Hiew L, Murphy GC (2006) Who should fix this bug? In: Proceedings of the 28th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '06, pp 361–370, DOI 10.1145/1134285.1134336

Basili V (1996) The role of experimentation in software engineering: past, current, and future. In: Proceedings of the 18th International Conference on Software Engineering, pp 442–449, DOI 10.1109/ICSE.1996.493439

Bernstein DJ (2000) Using maildir format. Online, http://cr.yp.to/proto/maildir.html

Bernstein PA, Goodman N (1983) Multiversion concurrency control—theory and algorithms. ACM Trans Database Syst 8:465–483

Bevan J, Whitehead EJ Jr, Kim S, Godfrey M (2005) Facilitating software evolution research with Kenyon. In: ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, ACM, New York, NY, USA, pp 177–186

Bird C, Rigby PC, Barr ET, Hamilton DJ, German DM, Devanbu P (2009) The promises and perils of mining Git. In: Godfrey MW, Whitehead J (eds) MSR '09: Proceedings of the 6th IEEE Intl. Working Conference on Mining Software Repositories, Vancouver, Canada, pp 1–10

Boehm B (1987) Industrial software metrics top ten list. IEEE Software 4(5):84–85

Brooks FP (1975) The Mythical Man Month. Addison-Wesley, Reading, MA

Bunch C, Chohan N, Krintz C, Chohan J, Kupferman J, Lakhina P, Li Y, Nomura Y (2010) An evaluation of distributed datastores using the AppScale cloud platform. IEEE International Conference on Cloud Computing 0:305–312

Buse RP, Zimmermann T (2012) Information needs for software development analytics. In: Proceedings of the 34th International Conference on Software Engineering

Canfora G, Cerulo L (2006) Fine grained indexing of software repositories to support impact analysis. In: MSR '06: Proceedings of the 2006 international workshop on Mining software repositories, ACM, New York, NY, USA, pp 105–111

Cubranic D, Murphy G, Singer J, Booth K (2005) Hipikat: a project memory for software development. Software Engineering, IEEE Transactions on 31(6):446–465, DOI 10.1109/TSE.2005.71

D'Ambros M, Lanza M (2010) Distributed and collaborative software evolution analysis with Churrasco. Science of Computer Programming 75(4):276–287

Dean J, Ghemawat S (2004) MapReduce: Simplified data processing on large clusters. In: Proceedings of the 6th Symposium on Operating Systems Design and Implementation, pp 137–150

Deligiannis S, Shepperd M, Webster S, Roumeliotis M (2002) A review of experimental investigations into object-oriented technology. Empirical Software Engineering 7(3):193–231, DOI 10.1023/A:1016392131540

Demeyer S, Tichelaar S, Tichelaar E, Steyaert P (1999) FAMIX 2.0: The FAMOOS information exchange model. Tech. rep., University of Bern

Deursen AV, Klint P (1998) Little languages: little maintenance? Journal of Software Maintenance: Research and Practice 10(2):75–92

Dyer R, Nguyen H, Rajan H, Nguyen T (2012) Boa: Analyzing ultra-large-scale code corpus. In: Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity, ACM, New York, NY, USA, SPLASH '12, pp 87–88, DOI 10.1145/2384716.2384752

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

Fluri B, Wursch M, PInzger M, Gall H (2007) Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Transactions on Software Engineering 33(11):725–743

Gall H, Fluri B, Pinzger M (2009) Change analysis with Evolizer and ChangeDistiller. IEEE Software 26(1):26 – 33

Gasser L, Scacchi W (2008) Open Source Development, Communities and Quality, IFIP International Federation for Information Processing, vol 275, Springer Boston, chap Towards a Global Research Infrastructure for Multidisciplinary Study of Free/Open Source Software Development, pp 143–158

Gawer A, Cusumano M (2002) Platform Leadership. Harvard Business School Press

German DM, Hindle A (2005) Measuring fine-grained change in software: Towards modification-aware change metrics. IEEE International Symposium on Software Metrics 0:28, DOI http://doi.ieeecomputersociety.org/10.1109/METRICS.2005.32

Gîrba T, Ducasse S (2006) Modeling history to analyze software evolution. Journal of Software Maintenance and Evolution: Research and Practice 18(3):207–236

Glass RL, Vessey I, Ramesh V (2002) Research in software engineering: an analysis of the literature. Information and Software Technology 44(8):491–506, DOI DOI:10.1016/S0950-5849(02)00049-6

Godfrey MW, Hassan AE, Herbsleb J, Murphy GC, Robillard M, Devanbu P, Mockus A, Perry DE, Notkin D (2009) Future of mining software archives: A roundtable. IEEE Software 26(1):67–70, DOI http://doi.ieeecomputersociety.org/10.1109/MS.2009.10

Goeminne M, Mens T (2011) A comparison of identity merge algorithms for software repositories. Science of Computer Programming

González-Barahona JM, Robles G (2012) On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. Empirical Software Engineering 17(1–2):75–89

Gousios G, Spinellis D (2009) A platform for software engineering research. In: Godfrey MW, Whitehead J (eds) MSR '09: Proceedings of the 6th Working Conference on Mining Software Repositories, IEEE, pp 31–40

Halstead M (1977) Elements of software science. Elsevier Publishing Company

Heitlager I, Kuipers T, Visser J (2007) A practical model for measuring maintainability. In: Proceedings of the 6th Int. Conf. on the Quality of Information and Communications Technology, IEEE Computer Society, pp 30–39

Herraiz I, Izquierdo-Cortazar D, Rivas-Hernandez F, Gonzalez-Barahona J, Robles G, Duenas-Dominguez S, Garcia-Campos C, Gato J, Tovar L (2009) FLOSSMetrics: Free/libre/open source software metrics. In: CSMR '09. 13th European Conference on Software Maintenance and Reengineering, pp 281–284

Hovemeyer D, Pugh W (2004) Finding bugs is easy. SIGPLAN Not 39(12):92–106

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

Howison J, Wiggins A, Crowston K (2008) Open Source Development, Communities and Quality, IFIP International Federation for Information Processing, vol 275, Springer Boston, chap eResearch Workflows for Studying Free and Open Source Software Development, pp 405–411

ISO/IEC (2004) 9126:2004 Software engineering – Product quality – Quality model. Tech. rep., International Organization for Standardization, Geneva, Switzerland

Johnson P, Kou H, Paulding M, Zhang Q, Kagawa A, Yamashita T (2005) Improving software development management through software project telemetry. Software, IEEE 22(4):76–85, DOI 10.1109/MS.2005.95

Kalliamvakou E, Gousios G, Spinellis D, Pouloudi N (2009) Measuring developer contribution from software repository data. In: Poulymenakou A, Pouloudi N, Pramatari K (eds) MCIS 2009: 4th Mediterranean Conference on Information Systems, pp 600–611

Lattner C, Adve V (2004) LLVM: A compilation framework for lifelong program analysis and transformation. IEEE/ACM International Symposium on Code Generation and Optimization 0:75

Lin Y, Huai-Min W, Gang Y, Dian-Xi S, Xiang L (2010) Mining and analyzing behavioral characteristic of developers in open source software. Chinese Journal of Computers 10:1909–1918

Linstead E, Bajracharya S, Ngo T, Rigor P, Lopes C, Baldi P (2009) Sourcerer: mining and searching internet-scale software repositories. Data Mining and Knowledge Discovery 18:300–336, 10.1007/s10618-008-0118-x

Livieri S, Higo Y, Matushita M, Inoue K (2007) Very large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCfinder. In: Proceedings of the 29th international conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, ICSE '07, pp 106–115

Luijten B, Visser J, Zaidman A (2010) Faster defect resolution with higher technical quality of software. In: Proc. of the 4th International Workshop on Software Quality and Maintainability (SQM'10)

Lungu M, Lanza M, Gîrba T, Robbes R (2010) The Small Project Observatory: Visualizing software ecosystems. Science of Computer Programming, Elsevier 75(4):264–275, DOI 10.1016/j.scico.2009.09.004

McCabe T (1976) A complexity measure. IEEE Transactions on Software Engineering pp 308–320

Mitropoulos D, Gousios G, Spinellis D (2012) Measuring the occurrence of security-related bugs through software evolution. In: Proceedings of the 16th Pan-Hellenic Conference on Informatics, to appear

Mockus A (2009) Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In: Godfrey MW, Whitehead J (eds) MSR '09: Proceedings of the 6th IEEE Intl. Working Conference on Mining Software Repositories, pp 11–20

Mockus A, Fielding RT, Herbsleb JD (2002) Two case studies of open source software development: Apache and Mozilla. ACM Trans Softw Eng Methodol 11(3):309–346, DOI 10.1145/567793.567795

Muthanna S, Kontogiannis K, Ponnambalam K, Stacey B (2000) A maintainability model for industrial software systems using design level metrics. In: Proceedings of the seventh Working Conference on Reverse Engineering, Brisbane, AU, pp 248–256

Nierstrasz O, Ducasse S, Gîrba T (2005) The story of Moose: an agile reengineering environment. In: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, ACM, New York, NY, USA, ESEC/FSE-13, pp 1–10

Oman P, Hagemeister J (1994) Construction and testing of polynomials predicting software maintainability. Journal of Systems and Software 24(251–266)

Parr T, Quong R (1995) ANTLR: A predicated-LL (k) parser generator. Software: Practice and Experience 25(7):789–810

Pavlo A, Paulson E, Rasin A, Abadi DJ, DeWitt DJ, Madden S, Stonebraker M (2009) A comparison of approaches to large-scale data analysis. In: Proceedings of the 35th SIGMOD international conference on Management of data, ACM, New York, NY, USA, SIGMOD '09, pp 165–178

Perry DE, Porter AA, Votta LG (2000) Empirical studies of software engineering: a roadmap. In: ICSE '00: Proceedings of the Conference on The Future of Software Engineering, ACM, New York, NY, USA, pp 345–355

Pike R, Dorward S, Griesemer R, Quinlan S (2005) Interpreting the data: Parallel analysis with Sawzall. Scientific Programming 13(4):277–298

Posnett D, Devanbu P, Filkov V (2012) MIC check: A correlation tactic for ESE data. In: MSR'12: Proceedings of the 9th Working Conference on Mining Software Repositories, Zurich, CH

Ratzinger J, Sigmund T, Gall HC (2008) On the relation of refactorings and software defect prediction. In: MSR '08: Proceedings of the 2008 international working conference on Mining software repositories, ACM, New York, NY, USA, pp 35–38, DOI 10.1145/1370750.1370759

Raymond ES (2001) The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O' Reilly and Associates, Sebastopol, CA

Reshef DN, Reshef YA, Finucane HK, Grossman SR, McVean G, Turnbaugh PJ, Lander ES, Mitzenmacher M, Sabeti PC (2011) Detecting novel associations in large data sets. Science 334(6062):1518–1524

Resnick P (2008) Internet message format. RFC 5322, Internet Engineering Task Force

Robles G (2005) Empirical software engineering research on libre software: Data sources, methodologies and results. PhD thesis, Universidad Rey Juan Carlos, Madrid

Robles G, Gonzalez-Barahona JM (2005) Developer identification methods for integrated data from various sources. In: MSR '05: Proceedings of the 2005 international workshop on Mining software repositories, ACM, New York, NY, USA, pp 1–5, DOI 10.1145/1083142.1083162

Robles G, Koch S, Gonzalez-Barahona JM (2004) Remote analysis and measurement of libre software systems by means of the CVSAnalY tool. In: Proceedings of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS), Edinburg, Scotland, UK

Samoladas I, Stamelos I, Angelis L, Oikonomou A (2004) Open source software development should strive for even greater code maintainability. Commun ACM 47(10):83–87

Sarma A, Maccherone L, Wagstrom P, Herbsleb J (2009) Tesseract: Interactive visual exploration of socio-technical relationships in software development. In: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, Washington,

DC, USA, ICSE '09, pp 23–33
Seaman C (1999) Qualitative methods in empirical studies of software engineering. Software Engineering, IEEE Transactions on 25(4):557–572
Shang W, Adams B, Hassan AE (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, ACM, New York, NY, USA, ASE '10, pp 275–284
Shang W, Adams B, Hassan AE (2011) Using Pig as a data preparation language for large-scale mining software repositories studies: An experience report. Journal of Systems and Software In Press, Corrected Proof
Shaw M (2003) Writing good software engineering research papers. In: Proceedings of the 25th International Conference on Software Engineering, 2003., pp 726–736, DOI 10.1109/ICSE.2003.1201262
Shull F, Carver J, Vegas S, Juristo N (2008) The role of replications in empirical software engineering. Empirical Software Engineering 13:211–218, 10.1007/s10664-008-9060-1
Sjøberg DI, Hannay J, Hansen O, Kampenes V, Karahasanovic A, Liborg NK, Rekdal A (2005) A survey of controlled experiments in software engineering. IEEE Transactions on Software Engineering 31(9):733–753
Sjøberg DI, Dyba T, Jorgensen M (2007) The future of empirical methods in software engineering research. In: FOSE '07: 2007 Future of Software Engineering, IEEE Computer Society, Washington, DC, USA, pp 358–378, DOI http://dx.doi.org/10.1109/FOSE.2007.30
Šmite D, Wohlin C, Gorschek T, Feldt R (2010) Empirical evidence in global software engineering: a systematic review. Empirical Software Engineering 15:91–118
Spinellis D (2006) Code Quality: The Open Source Perspective. Addison-Wesley, Boston, MA
Spinellis D (2007) Another level of indirection. In: Oram A, Wilson G (eds) Beautiful Code: Leading Programmers Explain How They Think, O'Reilly and Associates, Sebastopol, CA, chap 17, pp 279–291
Spinellis D (2008) A tale of four kernels. In: ICSE '08: Proceedings of the 30th international conference on Software engineering, ACM, New York, NY, USA, pp 381–390, DOI 10.1145/1368088.1368140
Spinellis D, Szyperski C (2004) How is open source affecting software development? IEEE Software 21(1):28–33, DOI 10.1109/MS.2004.1259204, guest Editors' Introduction: Developing with Open Source Software
Tempero E, Anslow C, Dietrich J, Han T, Li J, Lumpe M, Melton H, Noble J (2010) Qualitas corpus: A curated collection of Java code for empirical studies. In: 2010 Asia Pacific Software Engineering Conference (APSEC2010), pp 336–345
Thusoo A, Shao Z, Anthony S, Borthakur D, Jain N, Sen Sarma J, Murthy R, Liu H (2010) Data warehousing and analytics infrastructure at Facebook. In: Proceedings of the 2010 international conference on Management of data, ACM, New York, NY, USA, SIGMOD '10, pp 1013–1020
Tichy WF, Lukowicz P, Prechelt L, Heinz EA (1995) Experimental evaluation in computer science: A quantitative study. Journal of Systems and Software 28(1):9–18, DOI DOI: 10.1016/0164-1212(94)00111-Y
Welker K, Oman P (1995) Software maintainability metrics models in practice. Journal of Defence Software Engineering 8(19–23)
Wheeler DA (2010) Linux kernel 2.6: It's worth more! Online
Witten IH, Frank E (2005) Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann
Wohlin C, Wesslen A (2000) Experimentation in software engineering — An introduction. Kluwer Academic Publishers
Zannier C, Melnik G, Maurer F (2006) On the success of empirical studies in the international conference on software engineering. In: ICSE '06: Proceedings of the 28th international conference on Software engineering, ACM, New York, NY, USA, pp 341–350, DOI 10.1145/1134285.1134333
Zelkowitz MV, Wallace D (1997) Experimental validation in software engineering. Information and Software Technology 39(11):735–743, evaluation and Assessment in Software Engineering