TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

M.M. Mahbubul Syeed

On the Socio-Technical Dependencies in Free/Libre/Open Source Software Projects

Julkaisu 1300 • Publication 1300

Tampere 2015

Tampereen teknillinen yliopisto. Julkaisu 1300
Tampere University of Technology. Publication 1300

M.M. Mahbubul Syeed

On the Socio-Technical Dependencies in Free/Libre/Open Source Software Projects

Thesis for the degree of Doctor of Science in Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB109, at Tampere University of Technology, on the 29th of May 2015, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2015

ISBN 978-952-15-3525-3 (printed)
ISBN 978-952-15-3533-8 (PDF)
ISSN 1459-2045

Abstract

During the course of the past two decades, Open Source Software (OSS) development model has lead to a number of projects which have produced software that rivals and in some cases even exceeds the scale and quality of the traditional software projects. Among others, Eclipse, Apache, Linux, and BSD operating system are representative examples of such success stories.

However, OSS project like traditional in-house projects, often pose the potential for enormous problems, whose effects run the gamut from immense cumulative delay through complete breakdown and failure. This situation is evident, as OSS development is a socio-technical endeavor and is non-trivial. Such development occurs within an intensively collaborative process, in which technical prowess must go hand in hand with the efficient coordination and management of a large number of social, inter-personal interactions across the development organization. Furthermore, those social and technical dimensions are not orthogonal. It has been recognized that the structure of a software product and the layout of the development organization working on that product correlate.

Therefore this thesis argue that a comprehensive understanding on the sustainable evolution of OSS projects can be gained through the examination of the mutual influence of social and technical dimensions in OSS development. Thus, the goal of this thesis is the verification and reasoning of the following proposition,

"The evolution of the Open Source Software (OSS) project is constrained by the non-orthogonal evolution of Social and Technical dimensions (often termed as Socio-Technical dependency) of such projects".

In concrete terms, this thesis investigates and measures empirically the extent to which the two dimensions of OSS projects, social and technical, approximate and influence each other during the evolution of the projects. Perceived insight is then used to build proposals that would provide empirical basis to frame theory around the affirmed proposition.

Moving towards this goal, this thesis proposes models, methods, frameworks and tool supports to measure, assess, and reason the socio-technical dependency within OSS project context. The starting point is to propose a data model to mimic the social and technical dimensions and their inter-relationships. This model is instantiated through the repository data of OSS projects that represent each of these dimensions. Then, methods and a mathematical model are proposed to derive dependency between the two dimensions, and to utilize them in measuring socio-technical dependency quantitatively. These proposals are then put into practice within distinct OSS project contexts to empirically measure and investigate socio-technical dependency. Along the process, frameworks, architectural design and corresponding tool implementations are provided to automate the analysis and visualization of such dependency.

Reported results suggest that high degree of socio-technical congruence can be considered as the implicit underlying principle for building team collaboration and coordination within the developer community of long lived OSS projects. Even being highly distributed community of developers, and mostly using passive communication channels, OSS communities are tied together by maintaining task dependent communication. Such communication is often ad-hoc, adaptive and situated as it cope with rapid and continuous changes in the underlying software.

Additionally, collaboration among projects are significantly influenced by the resembling properties among the projects. Resembling properties (e.g., project domain, size, and programming language) often form a favorable ground, thus creating a stimuli for developers to participate in those projects.

Keywords: Open Source Software (OSS), Free/Libre/OSS projects, Socio-Technical dependency, Socio-Technical congruence, Conway's law, Resemblance, Forking.

Preface

Research work of this thesis has been carried out during the years 2011-2014 as an individual research work in the Department of Pervasive Computing, Tampere University of Technology, Finland. This thesis is financially supported by Tampere Doctoral Programme in Information Sciences and Engineering (TISE) during the period 2011-2013, and by TUT graduate school funding for the year 2014. I would also like to acknowledge *Tivit/Digile's Cloud Software* and *Need4Speed* projects, *Nokia* foundation, *Ulla Tuomisen* and *Tuula and Yrjö Neuvo* säätiö for their financial support at various stages of my Ph.D studies.

First and foremost, I would like to thank Allah the Almighty, to bless me with the strength, capacity, mercy and protection in all aspects of my life.

I would like to express my sincere and deepest gratitude to my academic supervisor, Associate Professor Imed Hammouda for his invaluable guidance, continuous support, encouragement and infinite tolerance throughout the course of this work. Sincere gratitude to Professor Tommi Mikkonen for creating an enthusiastic work atmosphere and constructive feedback in getting the thesis done.

I would like to thank Professor Kari Smolander and Dr. Andrea Capiluppi for their valuable and constructive feedback and competent judgement on the manuscript.

I would like to take this opportunity to offer my sincere gratitude to one of the best human beings I have ever mate, the late Professor Tarja Systä. Her vision and guidance would always be with me.

I would like to express my gratitude to Ulla Siltaloppi and Hilkka Losoi for their kind help with practical matters and friendly support.

Special thanks to all my friends in Finland due to whom life in abroad becomes homely. I am very obliged to families Asaduzzaman and Bhuiyan for offering numerous enjoyable moments. I would like to express my deepest gratitude to my younger brothers, Shumon, Tareq, Khyrul, Mishu, Mehran, Sunny and Safwan, who often be the inspiration of my work.

I wish to express my deepest gratitude to my beloved wife Fatema, my sweet daughter duo – Suha and Samah for their endless support, encouragement and love due to which each step of life become a pave to success.

Finally, the person to whom my existence belongs, Abu Jafar Md. Syeed, my father, friend and mentor, this work is dedicated to you.

Tampere, May, 2015
M.M. Mahbubul Syeed

Contents

List of Figures

List of Tables

List of Included Publications

[I] M.M. Syeed, I. Hammouda, and T. Systä. Evolution of Open Source Software Projects: A Systematic Literature Review. *Journal of Software, vol. 8, no. 11, 2013*, pages 2815–2829. November, 2013.

[II] M.M. Syeed, T. Aaltonen, I. Hammouda, and T. Systä. Tool Assisted Analysis of Open Source Projects: A Multi-faceted Challenge. *International Journal of Open Source Software and Processes, vol. 3, no. 2, 2011*, pages 43–78, April-June, 2011.

[III] M.M. Syeed, and I. Hammouda. Socio-technical Congruence in OSS Projects: Exploring Conway's Law in FreeBSD. In *Proceedings of the 9th IFIP WG 2.13 International Conference of Open Source Systems (OSS'2013)*, pages 109–126. Springer, June, 2013.

[IV] M.M. Syeed, and I. Hammouda. Socio-Technical Dependencies in Forked OSS Projects: Evidence from BSD Family. *Journal of Software, vol. 9, no. 11, 2014*, pages 2895–2909. November, 2014.

[V] M.M. Syeed, K. Marius Hansen, I. Hammouda, K. Manikas. Socio-Technical Congruence in the Ruby Ecosystem. In *Proceedings of the 10th International Symposium on Open Collaboration (OpenSym)*, pages 2. ACM, August, 2014.

[VI] M.M. Syeed, and I. Hammouda. Who Contributes to What? Exploring Hidden Relationships Between FLOSS Projects. In *Proceedings of the 10th IFIP WG 2.13 International Conference of Open Source Systems (OSS'2014)*, pages 21–30. Springer, May, 2014.

[VII] M.M. Syeed, I. Hammouda, and C. Berko. Exploring Socio–Technical Dependencies in Open Source Software Projects — Towards an Automated Data-driven Approach. In *Proceedings of 17th the Academic MindTrek Conference*, pages 273–280. ACM, September, 2013.

[VIII] M.M. Syeed. Binoculars: Comprehending Open Source Projects through graphs. In *Proceedings of 8th IFIP WG 2.13 International Conference of Open Source Systems (OSS'2012)*, pages 350–355. Springer, September, 2012.

Chapter 1

Introduction

During the past two decades, Open Source Software (OSS) development has become a powerful mechanism for developing and distributing IT applications. A number of IT pioneers are now putting serious interest and investment in favour of OSS movement which supports OSS to gain substantial market credibility and legitimacy [73].

What makes OSS projects inimitable to that of the traditional in-house software development can be traced back in the seminal work of Raymond, "The Cathedral and the Bazaar" [74]. In this book, the traditional form of software development model, which is primarily barred within a closed group of people planning a cathedral, is placed in a contrast to the bazaar model of OSS development. Within this model, a loosely-knit community of developers and developer-turned-users commonly strewn all over the world, come together to develop the software, without the desire of monetary compensation. Developers participating in these projects join or leave voluntarily, not being restricted by organizational rules and regulations. Most often such model of development capitalizes on rigorous peer review and parallel debugging that lead to innovation and rapid advance during the evolution of software products [55]. Produced software is often distributed freely under a chosen license agreement, such as GNU General Public License [55] to guide the right to access, modify and redistribute the code.

To facilitate open participation and collaboration, open source projects exploit a wide range of components, often coming with a large number of versions reflecting their development and evolution history. Such components can be classified broadly into two categories: technical artifacts, and the social artifacts. Technical artifacts often consist of code management tools, such as version control systems (e.g., CVS; SVN, Git), issue management systems (e.g., Jira, Bugzilla), and commit records. In contrast social artifacts are often composed with community communication channels, which is mostly composed of cheap medium over Internet. For instance, emails, forums, wiki's, chatting are commonly used [I]. These artifacts (both technical and social) offer the means for adequate management and development of the product, and to build community collaboration through communication which ranges from pure technical discussion to merely social conversation.

Software projects of such distributed nature often pose the potential for enormous problems, whose effects run the gamut from immense cumulative delay through complete breakdown and failure [39]. Such situation is evident, as software development is a socio-technical endeavor [92]. Any non-trivial software development occurs within an intensively collaborative process, in which technical prowess must go hand in hand with the efficient coordination and management of a large number of social, inter-personal interactions across the development organization. Furthermore, those social and technical dimensions are not orthogonal. It has been recognized that the structure of a software product and the layout of the development organization working on that product correlate [92].

However, despite of the high potential to be exposed to such development problems which often jeopardies a project's success, OSS development model has lead to an adequate number of successful projects. These projects produced software that rival and in some cases even exceeds the scale and quality of the traditional software projects [6].

Therefore, in gaining a comprehensive understanding on the sustainable evolution of OSS projects, the mutual influence of social and technical aspects in OSS development must be evaluated.

One way to investigate this issue is to measure if there is a good fit (or congruence) between the coordination structure mandated by the technical work units (tasks) and the actual social organization (as expressed for example by the communication paths observed among its members). Such dependency between the technical and the social dimensions are often called the Socio-technical Dependency (see Section 2.2).

This thesis offers a thorough investigation on the socio-technical dependency within the context of OSS projects through the utilization of the data maintained in OSS repositories. These repositories represent both the technical and social evolution of the project.

1.1 Motivation

Software development, hence also an OSS project, is a complex engineering activity due to its inherent multi-dimensional perspectives [9]. It is not about developing the software only. In reality, it involves interactions among people, processes, and tools during the development of a complete software product. In practice, software development is performed by teams, possibly geographically dispersed, consisting of a number of individuals ranging from tens to thousands [27]. In this setup, organizations often organize themselves around their products' architectures; the main components of their product define the organization's key subtasks [45] and become the source of most relevant information pertinent to the task dependencies that define coordination need among individuals and teams [25].

Due to dynamic nature of software development, changes in task dependencies are obvious, which in turn jeopardies the appropriateness of the information flows and perturb the organization's ability to coordinate effectively. Modularity, the principal concept of software engineering, cannot alone reduce or maintain the coordination needs [7], because the theory of modularity relies on the assumption that there exists a simple and obvious relationship between the product modularization and the task (or work) modularization [4] [69]. Hence, by reducing the technical interdependencies among the modules of a system, the modularization theories argue, work interdependencies are reduced, thereby reducing the need for communication among work groups [16]. Unfortunately such assumption has flaws when applied in the context of software development. For instance, software modularization approaches use only a subset of the technical dependencies (e.g., syntactic relationships [31]), leaving out other potentially relevant ones [16]. However, the tie between product structure and task structure is no longer as simple as previously assumed, and that diminishes over time [16].

Additionally, injecting minimal communication between teams, collocated or distributed with increasing software modularity, is detrimental to the success of the project. Product development literature argues that information hiding, which leads to minimal communication between teams, causes variability in the evolution of the projects, frequently resulting in integration problems [95]. Putting it in software development context, information hiding led development teams to be unaware of other teams' work resulting in coordination problems [90], which is even more severe for geographically distributed software development projects [39].

Even though those findings do not overrule the necessity of software modularization, such modularization must be supplemented with effective coordination mechanisms to allow developers to deal correctly with the dynamic conditions that are not captured in the architectural specification [16]. Even for a highly modular architecture, effective coordination is drastically affected by minor changes in product architecture, which cause substantial changes in task dependencies [41]. Such changes often introduce new dependencies, modify the existing ones, or even eliminate some. Therefore, coordination and collaboration in such product development organizations must be calibrated according to the dynamic changes in dependencies within the product architecture.

This calibration for aligning the collaboration of the community to that of the technical structure of the software are termed as socio-technical dependency in software projects. Here, software, processes, tasks, and technology shape the technical dimension, and the developer community forms the social dimension of the project. In Section 2.2 a formal definition of these dimensions are given.

Recent research suggests that efficacious socio-technical alignment is auspicious for organizations, because high degree of socio-technical dependency within a project results faster completion of modification requests [17] with higher build success [57] and product quality [10]. In contrast, socio-technical gaps are noted problematic for distributed software development [28]. For instance, low socio-technical dependency is subjected to lower productivity with increased number of code changes [18] [28] and negative performance level [88] within the organizations. Complementary to these results, it is often cited that high degree of socio-technical dependency is a natural consequence and a desired property for collaborative development activities [11].

Being motivated by these facts, this research sets its goal to examine the notion of socio-technical dependency within the paradigm of OSS software projects. OSS is an interesting research topic due to its thriving success in many domains at a global scale. With community driven development model and open collaboration, such projects are in a clear contrast to that of in-house projects. However, until now, a very little is known on the driving forces that lead such projects towards successful evolution, specially from socio-technical perspective [I]. Thus, it is beneficial to explore whether successful OSS projects exhibits socio-technical properties which might explain the success and quality issue of such projects. In turn, such results would encourage in-house projects to adapt and mesh OSS trend and practices to the traditional practices in order to foster team communication and to minimize collaboration gaps.

A study of this kind requires access to the complete data set representing the evolution of both the technical and the social dimensions. In this regard, OSS projects often maintain the data in public repositories and make them available through Internet. Therefore, given suitable data mining tools, these data can be used in empirical research covering most of the aspects involved in OSS development.

In a nutshell, this thesis is intuitively motivated as follows: development of software artifacts that depend on each other is likely to require coordination between the people responsible for those artifacts. Conversely, lack of coordination across responsible people of dependent artifacts may be a telltale of development problems.

1.2 Research Focus

This research is centered round the following proposition,

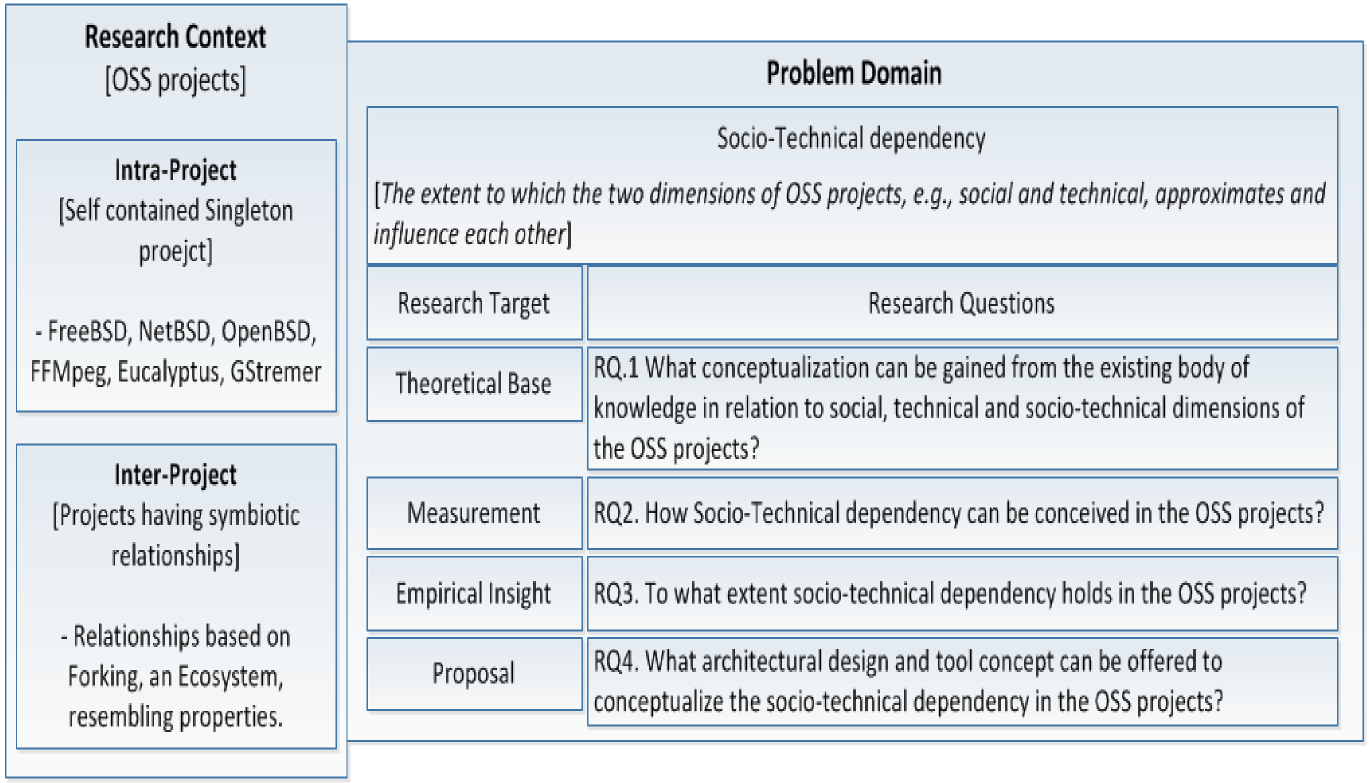
"The evolution of Open Source Software (OSS) projects is constrained by the non-orthogonal evolution of the Social and the Technical dimensions (often termed as socio-technical dependency) of such projects".

Therefore, the purpose of this thesis is to investigate and measure the extent to which the two dimensions of OSS projects, social and technical, approximate and influence each other during the evolution of the projects. In doing so a set of empirical investigation on the socio-technical dependency for a wide range of OSS project setup is carried out. Perceived insight are then used to build proposals that should provide empirical basis to frame theory around the proposition stated above. In order to build foundation for the empirical investigation, a concrete conceptualization of the socio-technical dependency is first conceived by defining each of the project dimensions (social, technical, socio-technical) and the related concepts. These definitions are then used to propose models, methods, framework and tool support that are used for piloting the empirical analysis.

During this process of investigation, special attention is paid to resolve issues that often pose threat to the research results and tool performance. This includes reproducibility and inter-comparability for the research results, whereas for the tools, issues like reusability, extensibility, automation with unified data representation are ensured.

Figure 1.1 draws the landscape of this research by defining the research context, problem domain, research focus, and corresponding research questions.

As per this figure, the context of this research is the OSS projects. It should be noted that OSS projects are not always standalone projects, thus dependency and collaboration often exists beyond the boundary of a project. Therefore, this thesis categorizes OSS projects in two abstraction levels based on certain relationship criteria, e.g., intra-project and inter-project category. Here, Intra-project defines a self-contained, singleton project that has its own project setup and management,



Research Context
[OSS projects]

Intra-Project
[Self contained Singleton proejct]

- FreeBSD, NetBSD, OpenBSD, FFMpeg, Eucalyptus, GStremer

Inter-Project
[Projects having symbiotic relationships]

- Relationships based on Forking, an Ecosystem, resembling properties.

Problem Domain

Socio-Technical dependency

[*The extent to which the two dimensions of OSS projects, e.g., social and technical, approximates and influence each other*]

Research Target	Research Questions
Theoretical Base	RQ.1 What conceptualization can be gained from the existing body of knowledge in relation to social, technical and socio-technical dimensions of the OSS projects?
Measurement	RQ2. How Socio-Technical dependency can be conceived in the OSS projects?
Empirical Insight	RQ3. To what extent socio-technical dependency holds in the OSS projects?
Proposal	RQ4. What architectural design and tool concept can be offered to conceptualize the socio-technical dependency in the OSS projects?

Figure 1.1: The research context and the problem domain

that is, a community of developers and own project management systems. For instance, FreeBSD, NetBSD, OpenBSD, FFMpeg are the examples of such singleton project. In contrast, Inter-project defines a set of projects that exhibits a given degree of symbiotic relationship. For this study, three such relationships are conceived, e.g., Forking (see Section 2.2.7), Resembling projects (see Section 2.2.8), and an Ecosystem (see Section 2.2.9).

The reason of defining this composition of OSS projects as the research context is to gain more holistic view on the topic by examining socio-technical dependency for each given context.

Therefore, socio-technical dependency in general constitutes the problem domain. Within this domain, four research targets and corresponding research questions are defined to fine tune the research focus.

The first research target offers a **Theoretical baseline** for piloting this study (see Figure 1.1). This target is formulated as follows,

RQ.1 What conceptualization can be gained from the existing body of knowledge in relation to social, technical, and socio-technical dimensions of the OSS projects?

Here the target is to build a theoretical foundation based upon the state-of-the-art research in order to define, (a) the motive and therefore the need for this study, (b) the three dimensions of OSS projects (social, technical and socio-technical) through systematic classification of the concerned literature, and finally, (c) the current progress and thereby set the target for further progress.

The next research target is the **Measurement** of the socio-technical dependency. Beyond just understanding how architectural dependencies impose task dependencies at organizational level and influence the project overall (as discussed in Section 1.1), practitioners need to be able to determine how well equipped a given organization is to carry out the design and implementation of a system with a particular architecture [42]. However, with the exception of informal and ad hoc attempts to accommodate the development organization, there is not yet any formal proposal of model and methods to assess the fit of the architecture and the organization proactively [I] [42]. This assessment is especially important to do at the early stage in the project, and also at discrete points where changes are made to the architecture, so that appropriate and timely adjustments can be made. Thereby, following research question originated,

RQ.2 How Socio-Technical dependency can be conceived in OSS projects?

The target here is to propose a quantitative measure of socio-technical dependency which is often defined as socio-technical congruence (see Section 2.2.5). Socio-technical congruence, as per the definition, verifies if there is a good fit between the coordination structure mandated by the technical work units (tasks), and the actual social organization (as expressed for example by the communication paths observed among its members) [92]. Therefore, it is a solid indicator of how well a development team is organized in carrying out a software development project.

This thesis proposes a mathematical model and associated methods to measure socio-technical congruence from the data maintained in OSS project repositories. As the model and the methods operate on OSS project data, a generic

OSS data model is a prerequisite. This data model should mimic the social, and technical dimensions, and offer means to explore their inter-relationships (i.e., socio-technical). Additionally, a single model to fit all would unify data modeling, analysis and presentation.

Finally, a framework is proposed that could integrate the data model, and implement the methods and the mathematical model to achieve fully automated analysis of socio-technical dependency.

In addition, a detailed, coherent, and systematic methodological approach is adopted to conduct the empirical researches for this thesis. This approach describes among others, the data sources, data acquisition, cleaning and representation in detail. This ensures the reproducibility of the reported results, and sets a benchmark against which comparative studies can be carried out.

The next target is to gain **Empirical insight** by putting the proposed approach in practice to empirically measure, evaluate and reason for the socio-technical dependency. Therefore, the following research question emerges,

RQ.3 To what extent socio-technical dependency holds in the OSS projects?

The aim here is to accumulate the empirical results comprising distinct OSS project context (see Figure 1.1) to build proposals, trends and patterns for socio-technical dependency in such projects.

In order to perform the empirical studies, six OSS projects, one OSS ecosystem and a third party OSS repository are used as case studies. The six OSS projects are, FreeBSD[1], NetBSD[2], OpenBSD[3], FFMpeg[4], Eucaliptus[5], and GStreamer[6]. The selected ecosystem is the Ruby Gems Ecosystem[7], and the third party data provider service is the OHLOH (recently changed to OpenHub)[8] repository.

Finally, the target is to formulate the **Proposals** for architectural design, corresponding tool implementation and thereby distill lessons learned during the tool-building experience as a practitioners guide. Following research question formalize this research target:

RQ.4 What architectural design and tool concepts can be offered to conceptualize the socio-technical dependency in the OSS projects?

The composite focus of this research question can be break down into following set of tasks,

(a) Define architectural design based on the OSS data model and the framework that would support implementation of the methods and the mathematical model for the analysis and visualization of the socio-technical dependency. Ad-

[1]http://www.freebsd.org/
[2]http://www.netbsd.org/
[3]http://www.openbsd.org/
[4]https://www.ffmpeg.org/
[5]https://www.eucalyptus.com/
[6]http://gstreamer.freedesktop.org/
[7]https://rubygems.org/
[8]https://www.openhub.net/

ditionally, such design should incorporate sound design principles to support rapid and distributed development and extension of the tools.

(b) Implement tools based on the proposed architecture to support the intended analysis through interactive visualization and exploration.

(c) Demonstrate the fit of the design and corresponding tools by deploying them in the empirical analysis.

(d) Distill recommendations and guidelines based on the lessons learned during the process of designing, implementation and evaluation of the tools.

Figure 1.2 further demonstrates the relationships among these four research targets.

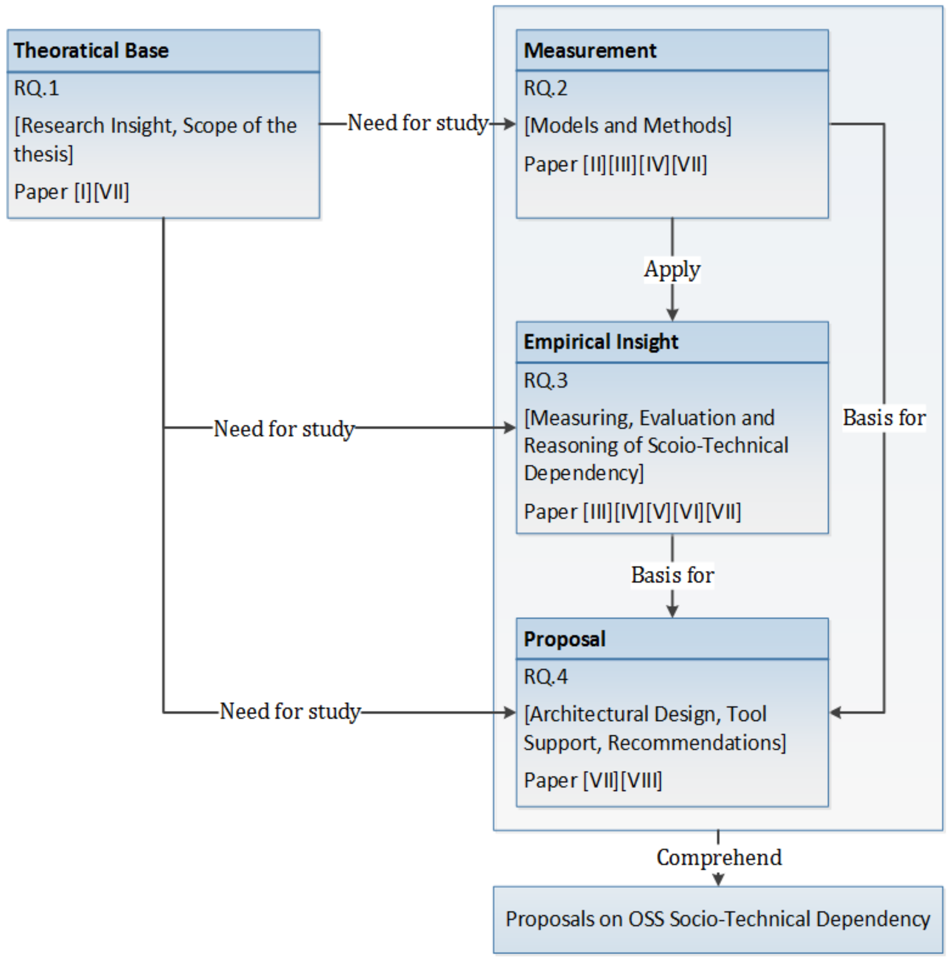

Theoratical Base
RQ.1
[Research Insight, Scope of the thesis]
Paper [I][VII]
Need for study
Measurement
RQ.2
[Models and Methods]
Paper [II][III][IV][VII]
Apply
Empirical Insight
RQ.3
[Measuring, Evaluation and Reasoning of Scoio-Technical Dependency]
Paper [III][IV][V][VI][VII]
Need for study
Basis for
Basis for
Proposal
RQ.4
[Architectural Design, Tool Support, Recommendations]
Paper [VII][VIII]
Need for study
Comprehend
Proposals on OSS Socio-Technical Dependency

Figure 1.2: An overview of the main research questions (RQs), how they relate to each other, the included papers that address each question.

The first research question (RQ1) under the target Theoretical Base is asked in order to gain insight into the challenges and practices within the problem domain. Therefore, it defines the problem domain in concrete terms, and forms the basis to formulate and investigate other research targets (as shown by the

Need for study link in Figure 1.2). Additionally, it offers a set of future research directions apart from defining the basis of this thesis.

The second research question (RQ.2) under the target Measurement is asked to define the models and methods to quantitatively measure socio-technical dependency.

The third research question (RQ.3) is formulated to perform the empirical investigation by utilizing the proposed models and methods. Therefore, an Apply relationship link is drawn between the targets Measurement and the Empirical Insight (see Figure 1.2).

Then, in the Proposal target, architectural design and tool implementations are offered to put the proposed models, methods and framework into practice. This target thus has a Basis for relationship with the prior two targets (see Figure 1.2).

Finally, the proposals on the socio-technical dependency is offered through the accumulation and comprehension of the observed results.

1.3 Contribution

This thesis includes eight publications, consisting of three journals and five conference publications. The author of this thesis is the main author of all of the eight included papers. This means that the responsibility for running the research, performing corresponding analysis and conducting most of the writing process has been assumed by this author with close cooperation of his supervisors.

The main contributions of this thesis are listed in the following:

✓ The thesis offers a systematic evaluation of the state-of-the-art research on the problem domain. This not only plots the basis for this study, but also contributes in forecasting prominent research directions on the topic.

✓ The thesis offers OSS data model to mimic the social and technical dimensions and their inter-relationships by instantiating the OSS project repositories.

✓ The thesis offers a generic set of graph theocratic methods on top of the OSS data model to explore and examine the social, technical and socio-technical dimensions of OSS projects.

✓ The thesis proposes a mathematical model to measure the socio-technical dependency.

✓ The thesis offers empirical investigation by deploying the models and methods to measure socio-technical dependency in OSS projects. To get a holistic view, a number of OSS projects are explored. This evaluation leads to build proposals on socio-technical dependency within the defined context of OSS projects.

✓ The thesis proposes framework to fully automate the data driven analysis of OSS projects.

- ✓ The thesis offers architectural design concepts based on the proposed data model and the framework that would support rapid and distributed development of analytical tools.
- ✓ The thesis offers semi-automated and automated tools that support data driven analysis and interactive exploration of socio-technical dependencies in OSS projects. These tools seamlessly integrate the proposed models, methods, and framework to address issues, such as, reusability of the data, tool extensibility, automation in analysis and visualization with unified data representation.
- ✓ The thesis offers recommendations and guidelines for the design and implementation of the tools for OSS data analysis and visualization.
- ✓ Finally, a detailed systematic methodological approach is adopted to conduct each empirical research in this thesis. This approach documents each critical step within the process to support reproducibility of the reported results and to perform comparative analysis.

1.4 Outline of The Thesis

This thesis is organized as follows: Chapter 2 defines the core concepts related to the context and the problem domain of this thesis. Related works along this line are also discussed. Chapter 3 is devoted to present, in detail, the research methodologies used to answer the research questions, and bridging them to each included paper to which they are applied. In Chapter 4, the main results and contributions of this thesis are summarized per reported study. Finally, the synthesis of the reported results are presented in Chapter 5. This includes explicit listing of contributions per research question, presenting the proposals on the socio-technical dependency, a discussion on the overall implications of this research, and presenting the concluding words.

Chapter 2

Background

This chapter defines the core concepts that constitute the research focus of this thesis. Related works on the topics are also presented. This chapter is organized as follows, Section 2.1 binds the concepts in relation to the research focus discussed in Section 1.2, the definitions of the concepts and the proposed mathematical model to measure socio-technical dependency are then presented in Section 2.2.

2.1 Theoretical Framework

The purpose of this thesis is to investigate and measure the extent to which the two dimensions of OSS projects, social and technical, approximate and influence each other during the evolution of the projects. Dependency between these two dimensions are often conceived as socio-technical dependency in related literature. The exact interpretation of these dimensions are given in Section 2.2.1, Section 2.2.2, and Section 2.2.3 to concretely define the problem domain (presented in Figure 1.1) of this thesis.

Within the *Measurement* research context (presented in Figure 1.2), the quantitative measurement of socio-technical dependency is done through the conceptualization of Conway's law and socio-technical congruence. Because these notions are used as the conceptual framework to define the mathematical model for measuring socio-technical dependency. Therefore, these two notions are defined in Section 2.2.4 and Section 2.2.5, respectively.

Then, the proposed mathematical model that quantitatively measure socio-technical congruence is presented in Section 2.2.6. Related definition of architecture (e.g., explicit and implicit architecture of the software) and developer coordination network (e.g., explicit and implicit coordination network) are also presented.

Within the *Empirical insight* research context, this mathematical model is used as a means to measure socio-technical congruence both at intra-project and at inter-project level. Recall that at intra-project level, congruence is examined for three distinct symbiotic relationships among OSS projects, e,g., forking, resemblance and ecosystem. Therefore, these project relationships are defined in Section 2.2.7, Section 2.2.8 and Section 2.2.9, respectively.

Finally, in Section 2.2.10 the background works motivating the proposal of the framework and the tool supports (within the *Proposal* research target) are presented.

2.2 Relevant Concepts

This section draws together the concepts sturdily tied with OSS projects and this thesis.

2.2.1 “Social” Dimension of OSS Projects

In Merriam-webster dictionary the term “Social” is defined as *Tending to form cooperative and interdependent relationships with others* [1].

OSS development is a perceptible example of the collaborative community-based model [84]. The community forms the social dimension of OSS projects which comprises developers and users [2]. Based on the level of participation, the role of the community members can be classified within the range of project leaders (maintainers) and core members (contributors) to active and passive users [98] [96] [102]. Project leaders are the ones who usually initiate the project and make the major development decisions. Core members make significant contributions to a project over time. Depending on task allocation, core members can further be subdivided into creators (leaders), communicators (managers), and collaborators [34]. Active users are the part of the community who report bugs, but do not fix them. Finally, passive users are all remaining users who just use the system.

Members of an OSS community are not subjected to any strict organizational rules, regulation or structure. Rather, they voluntarily join and contribute to the project. These people belong to discrete geographical locations having significant difference in background, time zone, language and cultural distances. Thus socialization within the community to contour collaboration, and cooperation are primarily done using simple communication media like email, wiki, and chat [101]. It is worth noticing that the medium of communication varies from project to project. For instance, FreeBSD, NetBSD, and OpenBSD projects maintain only mailing list archives to discuss the development and use of the system, to report bugs, and to commit patches for bug fixes and new features. In contrast, projects like FFMpeg, and Eucalyptus maintain a separate bug reporting system for tracking fault reporting and resolution. Due to such variation of communication channels, this thesis considers collecting data from all the relevant sources specific to a case study project that links to any form of communication, contribution and collaboration. Table 3.1 summarizes these data sources. Communication and collaboration among the developers seen by these communication media are used to define the social dimension, an example of which can be found in Section 2.2.6.

These data sources are frequently used by the researchers in understanding social dynamics of OSS projects [37]. Such efforts include, but are not limited to understanding the nature of collaboration in OSS development project [14] [96] [66]; examining the community structure, and practices [68] [46] [20]; un-

[1]http://www.merriam-webster.com/dictionary/social

derstanding the motivation, participation, and performance of the community members [75], and identifying success factors [37] [40] [93].

In this thesis the social dimension comprises the community of developers and their communication traces as seen by the communication medias used by the projects.

2.2.2 "Technical" Dimension of OSS Projects

In Merriam-webster dictionary, the term "Technical" is defined as *Relating to the practical use of machines or science in industry, medicine, etc*[2].

The technical dimension of an OSS project comprises the software codebase, and the activities concerning its development and maintenance. In other words, developers in such project exercise their software engineering knowledge and expertise in a real-life project setup to produce an industry scale software. Thus, the definition of the term technical fits in the context of OSS projects.

OSS projects often use a number of repositories and corresponding data management tools to organize the technical artifacts. Frequently used repositories include, source code repositories, bug/issue reports, commit records, and change logs. In managing changes of these repositories over time, following data management tools are used: CVS/SVN/GitHub for source code version management, Bugzilla/Zira/GitHub for bug/issue reporting and tracking. Many projects use categorical email archives to maintain commit records and change logs. For instance, FreeBSD, NetBSD, and OpenBSD use CVS and SVN to version control their code base. Additionally, NetBSD, and OpenBSD use categorical email archives for tracking commit records, change logs, and bug reports, whereas FreeBSD uses Bugzilla for bug/issue tracking and resolving. Data extracted from each of these sources are used to define the technical dimension, an example of which can be found in Section 2.2.6.

In this thesis the technical dimension consists of the source code for each stable release of a project, associated commit records, change logs, and issues that explicitly reveals the architectural dependencies, and the trace of contributions made by the developers during the evolution of the project.

Research within the technical domain has contributed heavily to understanding the complexity of the software development through exploration of the sources listed above. Focus of such research can be broadly classified along two axis, namely intra-project analysis and inter-project analysis.

Intra-project studies can be classified further into macro and micro level studies. Macro level studies explores how the different versions of the codebase evolve in the case of forked projects [77] for instance. In contrast, micro level studies mostly study the evolution of the product itself. Such research can be carried out for instance, to understand the software growth in size (e.g., lines of code, commits, comments, commit total) [76] [38] [13] [36], complexity (e.g., cyclomatic complexity, halstaed complexity), and modularity [80] [91] [86] of the software. Other studies can focus on the quality perspective of the software [19] [59] [64], or predict the evolution, maintainability (e.g., refactoring), defect density of the code base [97] [67] [51].

Inter-project studies, on the other hand, mostly explore the relationships

[2]http://www.merriam-webster.com/dictionary/technical

and dependencies among related projects. Such research includes for instance, studying update propagation patterns among different OSS projects [70].

2.2.3 "Socio-Technical" Dependency in OSS Projects

As discussed in Sections 2.2.1 and 2.2.2, software development endeavors involve two fundamental elements: a technical and a social component. The technical component consists of the product to develop, the processes, the tasks, and the technology employed in the development effort. The social component consists of the organization and the individuals involved in the development process, their attitudes and behaviors. In other words, a software development project can be thought of as a socio-technical system where the two components, the technical and the social, need to be aligned in order to have a successful project [16]. In literature such alignment is termed as Socio-Technical dependency. This dependency is prescribed as a natural consequence and desired property for a collaborative development environment [11]. Thus, an in-depth understanding on the socio-technical dependency within the context of OSS projects would enable to develop methods and taxonomy to better understand such development process [66].

2.2.4 Conway's Law

Conway's Law states that organizations which design systems are constrained to produce systems which are copies of the communication structures of these organizations [21]. That is, the software product architecture should reflect the organizational structure of its development team [96] [67]. In [51], Conway's Law is considered homomorphic and thus claimed to be true in reverse as well. This implies that the communication patterns among the developers should reflect the architectural dependency in the developed software. In literature, this law has been taken as a ground for investigating socio-technical dependencies in OSS projects [72], and advice that such investigation should be carried out at task level to device better team coordination and efficient resource management [43].

However, the question, *Does Conway's law matter in the modern era of software development?* remains central to the research community. As reported in [42], Conway's law still matters in software development as product quality is strongly affected by the organizational structure. Also, with the advent of global software engineering where development teams are distributed across the world, the impact of organizational structure on Conway's law and its implications on quality is significant [57].

This research takes this law as one of the basis for defining and conceptualizing Socio-technical congruence, and to derive the mathematical model to measure such congruence.

2.2.5 Socio-Technical Congruence

A key concern in studying socio-technical dependency is how to examine the relationship between the two dimensions. One way to do this is to measure Socio-Technical congruence, which is based on the Conway's law and the organizational theory. Both these theories offer a conceptual framework to define Socio-Technical congruence. As a side note, organizational theory identifies the fit or match between a particular organizational design and the organization's ability to carry out a task [12]. Current research in this track focuses on two factors: temporal dependencies among tasks that are assigned to organizational groups and the formal organizational structure as a means of communication and coordination [15] [69].

Capitalizing on these two theories, this thesis conceived the definition of socio-technical congruence as "the match between the coordination requirements established by the dependencies among tasks and the actual coordination activities carried out by the developers". In other words, the concept of congruence has two components: the coordination needs determined by the technical dependencies within the technical dimension, and the actual coordination activities that took place among the developers within the social dimension. The mathematical model proposed and evaluated in this thesis is presented in Section 2.2.6.

2.2.6 Measuring Socio-Technical Congruence

This section defines in brief the exact instantiation of the concepts presented earlier to measure socio-technical congruence. This approach is used in publications [III], [IV], [V] and [VIII] that are included in this thesis.

Explicit Architecture
The explicit architecture of a software presents the relationships among components of the software (e.g., code files, modules or packages) based on the actual design and implementation. In this thesis, functional dependency, attribute referencing and header file inclusion dependency at code file level are used to derive the Explicit Architecture of a software product. The produced architecture corresponds to the technical dimension of the project. This study derives the explicit architecture from the source code of each stable release of the following OSS projects: FreeBSD, NetBSD, OpenBSD, FFMpeg, and Eucalyptus. Figure 2.1(a) shows a hypothetical explicit architecture in which code files F1 and F2 exhibits a relationship with edge weight of 2, and code files F1 and f3 have a relationship with weight 1. The weight in this architecture designates the types of relationships identified for a pair of code files.

For the publication [III] and [IV], the explicit architecture was abstracted to get the package level explicit architecture. In this architecture, package P1 and P2 have a relationship if code file F1 in package P1 has a dependency relationship with a file F2 in package P2 or vice-versa. The relation weight between packages P1 and P2 reflects the total number of such relation found among the code files in P1 and P2.

Explicit Coordination Network
The explicit coordination network is a social network in which two developers have a relation if they have direct communication history as recorded in the mailing archives representing both the social and technical interactions among the developers. Thus, this network reveals the actual communication and collaboration that took place among the community members at any given point of time. In this thesis, developers coordination network is derived based on mailing archives that corresponds to both technical (e.g., development, commits, release planning, bug reporting and resolution) and social (e.g., general email

conversations) interactions. Figure 2.1(b) shows a hypothetical coordination network in which developers D1 and D2 have a relation with weight 15, whereas developers D2 and D3 have a relation with weight 20. Each relation weight in this network shows the number of interactions that have taken palace between a pair of developers. For this study, the explicit coordination network defines the social dimension of an OSS project.

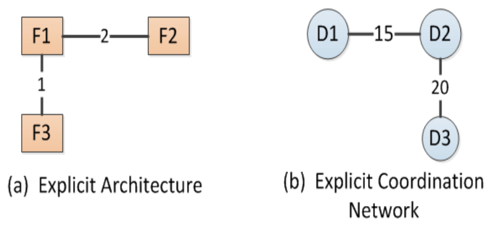


Figure 2.1: Explicit Architecture and Coordination Network

Implicit Architecture
The implicit architecture defines an architecture of the software where any two components (e.g., packages or code files) are related if there are developers who have either (a) contributed to both components, or (b) have direct communication at organizational level (e.g., a one-to-one email conversation). For instance, consider Figure 2.2(a). According to this figure, developer D1 has contributed to code files F1 and F2, developer D2 has contributed to code file F2, and developer D3 has contributed to code file F3. Also, note that developers D1, D2 and D3 have communication as shown by the edges between them. Thus, according to the definition, code files F1 and F2, and and F2 and F3 are linked to each other in the Implicit Architecture as shown in Figure 2.2(b). In this figure, edge weight between two code files defines the number of instances in which both the code files receive common contribution (according to the definition).

In the context of this thesis, an implicit architecture reveals the *should be* dependency among the code files based on the interaction history of their contributing developers at organizational level. Thus, a pairwise comparison of relations between implicit and explicit architectures would effectively reveal the level of congruence within that community.

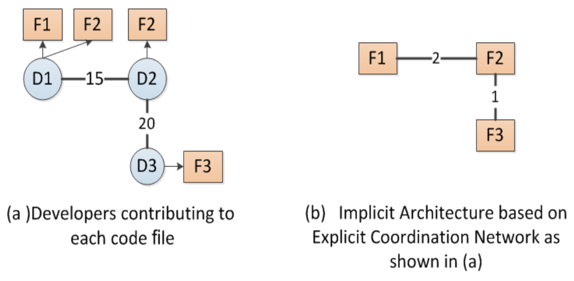

F1
F2
F2
D1
15
D2
20
D3
F3
(a)Developers contributing to each code file
F1
2
F2
1
F3
(b) Implicit Architecture based on Explicit Coordination Network as shown in (a)

Figure 2.2: Implicit Architecture derived from Explicit Coordination Network

Implicit Coordination Network
The implicit coordination network is a developer relation network in which two developers have a relation if they have contributed either (a) to a common code file or (b) to the code files that have direct relations in the Explicit Architecture. In other words, this network reveals the coordination patterns that should be devised by the technical dependency in the code base. For instance, consider that developer D1 has contributed to code files F1 and F2, developer D2 has contributed to code file F2, and developer D3 has contributed to code file F3 (as shown in Figure 2.3(a)). Hence, according to the definition, developers D1 and D2, and D1 and D3 have link with each other in implicit coordination network as shown in 2.3(b). In this figure, edge weight between two developers defines the number of instances in which both of them have common contribution (according to the definition).

In the context of this thesis, an implicit coordination network reveals the actual communication and coordination need among the developers based on their contribution or task responsibility. Thus, a pairwise comparison of relations between an implicit coordination network to that of the explicit one would effectively reveal the level of congruence within that community.

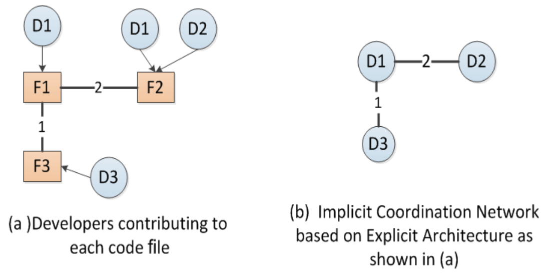

D1
D1
D2
F1
2
F2
1
F3
D3
(a)Developers contributing to each code file
D1
2
D2
1
D3
(b) Implicit Coordination Network based on Explicit Architecture as shown in (a)

Figure 2.3: Implicit Coordination Network derived from Explicit Architecture

Measuring Socio-Technical Congruence
Congruence was measured following the similarity measure presented in equation 2.1. This measure is analogous to fit / congruence measure used in organizational theory method [18] and to the Conway's Law.

$$Congruence = \frac{|Ref_A \bigcap Analogous_A|}{|Ref_A|} \times 100 \qquad (2.1)$$

In the above equation, Ref_A is the reference architecture or network (either explicit or implicit), and $Analogous_A$ is the analogous architecture or network (either explicit or implicit) with which congruence will be measured.

This equation measures congruence between the two given architectures or networks with respect to the reference one, Ref_A. Therefore, the numerator of equation 2.1 identifies the commonalities between the two given architectures, then divided by the size of the reference architecture and expressed as a percentage.

As an example, consider the explicit architecture presented in Figure 2.1(a) and the implicit architecture shown in Figure 2.2(b). A congruence measure between the two using equation 2.1, would be as follows,

Congruence = $\frac{|[E_{F1-F2}, E_{F1-F3}] \bigcap [E_{F1-F2}, E_{F2-F3}]|}{|[E_{F1-F2}, E_{F1-F3}]|} \times 100 = \frac{1}{2} \times 100 = 50\%$

Here, the explicit architecture is taken as Ref_A and the implicit architecture is considered as $Analogous_A$; result reveals 50% congruence between the two given architectures.

2.2.7 Forking

In the context of open source development, forking occurs when a part of a development community (or a third party not related to the original project) starts a completely independent line of development based on the source code of the original project [77]. To be considered as a fork, a project should have the following characteristics:

- A new project name.
- A branch of the software.
- A parallel infrastructure (web site, version control system, mailing lists, etc.).
- A new developer community.

Based on this definition, the following set of relationships are proposed within a pair of forked projects [IV]: (a) parent-child, in which one project is forked from the other, (b) siblings, if two projects are forked from the same parent project, and (c) lineages, for all decedent relationships in which (a) and (b) do not hold. For example, in Figure 3, NetBSD and OpenBSD have a parent-child relationship, FreeBSD and NetBSD are sibling projects, whereas FreeBSD and OpenBSD are the lineages of 386BSD. The Figure 2.4 shows these relationships among the BSD projects with rough time line of their forking.

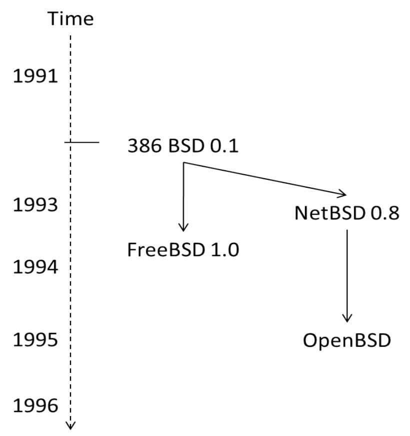


Figure 2.4: Rough time line of the forked BSD projects.

2.2.8 Resemblance

According to the Merriam webster dictionary the term *resemblance*[3] refers to *the state of looking or being like someone or something else*. In other words, resemblance refers to some distinctive properties that makes one person or thing like another.

In the context of this thesis, the given definition of resemblance is adopted to measure how similar two OSS projects are if they are both being contributed to by the same developer(s). The set of properties selected for defining resemblance / similarity between projects are programming language, project size (in LOC), license, project domain and rating. These properties are often cited by popular forges (e.g., SourceForge[4]) for categorizing FLOSS projects. The hypothesis here is that if OSS developers contribute to several projects, simultaneously or at different times, then there might have resemblance among those projects.

This study will be significant from diverse perspectives. For instance, companies may identify who influences and controls the evolution of a specific project of interest [1], explore the social structure of FLOSS development [24], or simply study what motivates people to join open source communities [8].

2.2.9 Software Ecosystem

A software ecosystem in general is the interaction of a set of actors on top of a common technological platform that results in a number of software solutions or services [62]. In literature, different perspectives are taken while defining a software ecosystem. This thesis adopts pure technical perspective. Two widely adopted definitions within this perspective are presented bellow,

According to Messerschmitt and Szyperski,

A software ecosystem refers to a collection of software products that has some given degree of symbiotic relationships [63].

According to Lungu et al.,

A software ecosystem is a collection of software projects which are developed and evolve together in the same environment [60].

This thesis views an ecosystem as a collection of software projects that evolves in the same environment and have some given degree of symbiotic relationships.

2.2.10 Tooling

There exists research works that acts as a driving force in proposing the framework, and tool support in answering RQ.4 under research target *Proposals*. For instance, the proposal for the framework targeting towards gaining full fledged automation of OSS data analysis is motivated by the evolution of quality models [35]. There are three generations of quality models that are distinct from each other according to the degree of automation they offer in exploring and analyzing data sources. For the first generation approaches, the tasks of data gathering and evaluation are mostly manual. For instance, first generation quality models for OSS (e.g., OpenBRR, QSOS) falls within this category [35]. The

[3]http://www.merriam-webster.com/dictionary/resemblance
[4]www.sourceforge.net

second generation approaches offer semi-automated process. A representative example would be the second generation quality models (e.g., QualOSS) [35]. Following the trend towards the automation of OSS data analysis approaches, it can be argued that eventually the third generation approaches would endeavor fully automated methods and techniques [35], which motivates in proposing the framework [VII].

Tool implementation to comprehend OSS dynamics offer several alternatives, each of which addresses specific challenges. For instance, CodeSaw [33] provides a time series representation of social interaction data in juxtaposed displays. Tesseract [81] explores the multi-perspective relationships in a project for a user-selected time period (i.e., the evolution), and represents them via four juxtaposed displays. In [5], FASTDash was proposed as an interactive conflict management tool which provides a spatial representation of the shared code base by highlighting team members current activity. CollabVS [26] addresses this issue at editing time, and provides a visual representation of conflicting code and a communication mechanism. Palantir [82] performs similar task by graphically displaying the shared workspace to the developers with the information of what others are doing, and calculating the severity of such activities. Also Augur [30] provides a line oriented view of the source code with colors for each pixel line indicating the location of the modification work and how recently it was conducted. In [89], Ariadne utilizes call-graph approach to visualize social dependency of the developers due to code sharing. Similarly, Expertise Browser [65] determines developers expertise from previous contributions. Yet none of these tools effectively explores graph theoretic methods though the unified modeling of OSS repository data for analysis and visualization. However, such approach is appropriate for OSS projects as graph structures are most suitable for analyzing data that exhibits inherent relationships. This need sets the favorable ground for the design and implementation of the Binoculars [VIII] and Pomaz tool [VII].

Chapter 3

Research Approach

In the quest to answer the research questions (presented in Section 1.2), it is essential to utilize certain research methodologies. Because the research methodology provides the link between research questions and the data used to answer them. Thus, it is indispensable to select a methodology that will provide the necessary data to answer the stated research questions. This chapter gives a synopsis of the methodological approaches that are used in this thesis.

The research presented in this thesis employs an empirical research method [100]. As dictated in [100] and [79], empirical research offers the opportunity to verify complex human behavior and their interactions with technology in a complex real life setup through observations and experiences. This method is also one of the highly practiced research approach in software engineering [79].

As this research primarily focused on understanding the mutual dependency between the social and technical dimensions within the real world setup, and belongs to software engineering domain, this thesis utilizes empirical research methodology.

This chapter is organized as follows, the overall landscape of the empirical research methods is presented in Section 3.1, and the methods applied in the context of this thesis are described in Section 3.2. Data collection and analysis approaches are highlighted in Section 3.3 and Section 3.4, respectively. Finally, the adoption of the research approaches per included publication is summarized in Section 3.5.

3.1 Basic Research Approach

Empirical research can be classified in accordance to two distinct paths: *fixed* and *flexible* [78] research design. The fixed research design is also known as *quantitative research*, whereas the flexible research design is often termed as *qualitative research*.

The fixed research design (or quantitative research) is a highly pre-specified research design, having a strong grab on what to do and how to do in advance. In this research approach a conceptual framework or theory needs to be developed before getting into the main part of the research study. Consequently, all data supporting the study need to be collected before starting to analyze it. Quantitative research is generally used to derive relations between variables, for instance, quantifying a relations or comparing two or more groups, for the purpose of explaining, predicting, or controlling the phenomena. Statistical methods are commonly used to establish such relations and confirm hypotheses, with generalized findings.

The flexible research design (or qualitative research), on the other hand, builds on the assumption that reality cannot be divided into discrete measurable variables. Hence, flexible research allows changing parameters of the design based on new information revealing during the course of the study. This includes modification of the research questions or data sources for example. The motive of qualitative research is to seek better understanding and explaining complex situations in their natural setting, where issues of the real world are described. Data used for qualitative research can be both qualitative and quantitative. Within this research paradigm, observations and inductive reasoning are used to build theory from the ground up.

The nature of this thesis is qualitative, even though quantitative data have been collected and analyzed. The contributions of this thesis are the empirical investigation of socio-technical dependency within the context of OSS projects, and offering novel models, methods, framework and tool support to comprehend the implication of such dependency. As such goals lead to gaining an in-depth understanding on the topic, qualitative research is more suitable than research in the breadth (quantitative research design).

3.2 Research Methods

There are several research methods for conducting empirical research in the domain of software engineering. Commonly used research methods include, but are not limited to, case studies [79], surveys [29], systematic literature review [52], constructive research and experiments [71] [100], and action research [99]. The research methods selected for this thesis were systematic literature review, case studies, and constructive research.

3.2.1 Systematic Literature Review (SLR)

Evidence-based Software Engineering (EBSE) [53] relies on aggregating the best available evidence to address engineering questions posed by researchers. A recommended methodology for such study is the Systematic Literature Review (SLR) [53]. An SLR is meant to identify, evaluate, and synthesize all available researches relevant to a particular research question, or phenomenon [53]. Research study contributing to a SLR is termed as *primary study*, whereas the SLR itself is called a *secondary study*.

An SLR can be exploited to serve a wide range of research purposes. For instance, it can be used for identifying gaps in the current research and develop a framework / background for appropriately positioning new research activities. Alternatively, it can be used to identify the extent to which current empirical evidences support / contradict theoretical hypotheses [52] [53].

The principal rationale in undertaking an SLR is to ensure that at every step of the process that it does not patronage any predefined research hypothesis, and report research that supports it [52]. Therefore, an SLR must be undertaken in accordance with a predefined review strategy / protocol [52].

A review protocol illustrates the method that will be used to carry out an SLR, and must be defined and peer reviewed before commencing the actual review [52]. The protocol should define the following elements [52] [53]: rationale and the research questions to be investigated; search strategy for identifying primary studies (e.g., search terms, forums relevant to the study); inclusion or exclusion criterion for the primary study; and finally, strategy for data extraction and synthesis.

In this thesis the SLR is conducted to meet the first research target, the *Theoretical baseline* (see Figure 1.2). The motive here is to gain a deeper insight on what constitutes the study of the social, technical and socio-technical dimensions of OSS projects, to identify gaps in current research, and thus to form the ground for initiating this research. Therefore, as research method, an SLR is the logical choice for this part of the thesis.

To carry out the SLR [I], a review protocol was adopted following the guidelines for conducting SLR in software engineering [52] [23].

3.2.2 Case Study

A case study is an in-depth investigation of a phenomenon in its real-life context [103] focusing on a specific case. The cases are objects of the real world which are studied in their natural setting, i.e., real software organizations, software projects, a product, a group of people, or an individual. Case studies are typically flexible design studies and well suited for software engineering research [79].

Software engineering research is a multidisciplinary discipline. On one hand, it studies activities like development, operation, and maintenance of the software and related artifacts [48]; on the other hand, it deals with individuals, groups and organizations that participate in those activities. Additionally, social and geo-political conditions that affect such process are also studied. Therefore, case study research can be effectively used to address many research questions in software engineering [79]. Accordingly, OSS projects, encompassing the multidisciplinary perspectives and the complexity of software engineering, are well suited for case study research.

However, realization of a case study method is specific to the research focus, as different approaches within case study research are targeted to serve different purposes [79]. For this thesis, positivist case study method is adopted [54]. A positivist case study searches evidence for formal propositions, measures variables, tests hypotheses, and draws inferences from a sample to a stated population [54].

In the context of this thesis, case study research is used to serve the research targets *Measurement*, and *Empirical insight*. Note that the focus of Measurement research target is to propose models (i.e., OSS data model and mathematical model) and associated methods. The data model mimics both the social and technical dimensions of OSS projects and supports exploring their inter-relationships. Whereas, the proposed mathematical model measures socio-technical dependency by mobilizing the methods on the data model. Then, under the Empirical insight target, the proposals are tested by measuring socio-technical congruence for a number of compositions of OSS projects (defined in Section 1.2). As the research targets contemplate the purpose of a positivist case study research, thus this approach is selected for this part of the thesis.

3.2.3 Constructive Research Approach (CRA)

A constructive research approach (CRA) can be defined as "a methodology that creates innovative constructions to solve real world problems and thus contributes to the field of study where it is applied" [61] [71]. This research approach is widely used within the paradigm of software engineering and computer science [50].

According to [61] [71], an idealized model of CRA should encompass the following building blocks: (a) find a practically relevant (real-life) problem which requires solution; (b) obtain an understanding on the topic and on the problem; (c) design on innovative artifact that is intended to solve the original problem; (d) implement and demonstrate that the solution works; (e) make a theoretical contribution through carefully linking the solution to existing theoretical knowledge; and (f) examine the scope of applicability.

In practice, the steps do not follow each other in a simple sequence; rather the process is iterative and sometimes also recursive.

In this thesis CRA is used to address the *Proposal* research target (as shown in Figure 1.2). Recall that the focus here is to present architectural design and tool implementations that put the proposed models, methods and framework into practice. The implemented tools are then tested with real life OSS project data. This verification demonstrates the suitability of the proposals and the tools in understanding the socio-technical dependency within the context of OSS projects. Therefore, CRA as a research method seems a good fit for this part of the thesis.

3.3 Data Collection Method

The collected data in empirical studies can be quantitative (e.g., numbers and classes) or qualitative (e.g., words, descriptions, and pictures). For this thesis, mostly qualitative research methods are applied, which tend to be based on qualitative data, however quantitative data have also been collected.

There are a number of data collection methods to be choose from, and the choice should dependent on the information sought after [147]. In this work, two data collection methods are followed. These methods are most commonly used for the selected research methods, and are highly accepted by the concerned research community [78] [37] [93]. These are described next.

3.3.1 Systematic Literature Search Strategy

Input to an SLR are the primary studies on the topic of interest. Searching and selection of the primary studies must be carried out according to the guidelines stated in the review protocol. This is required to reduce selection bias [52]. The study selection criteria is set to sort out the primary studies that are highly relevant to the research questions [52].

In order to formulate selection criteria for primary studies, a series of steps are needed to be executed. This includes: (a) defining the qualifying criteria (both inclusion and exclusion) for the primary studies. These criteria should be defined in relation to the research questions; (b) defining the search strategy that should list the relevant journals and conferences, search keywords and search strings for individual digital library. Search strategy might include only automated keyword search or a manual search, or both; and finally, (c) review and consult the search results with the domain experts to reduce selection bias [52] [I].

Meticulous interpretation and implementation of these steps are discussed in publication [I].

3.3.2 Archival data

An archive, according to Oxford dictionary[1], refers to a collection of historical documents or records providing information about a place, institution, or group of people. In the realm of software development organizations, archival data records refer to, for instance, meeting minutes, documents from different development phases, organizational charts, communication and development history log, versions of the code releases and previously collected measurements in an organization [78]. For efficient management and organization of archival data, configuration management tools are needed. Such tools offer collection, storage, and tracking of changes over different versions of a document.

Advantages of using archival data is many. They include for instance, taping into extensive data sets that are often drawn from large representative samples. These data is recorded and maintained, thus having a clear chain of modification history. Such volume of historical data is well beyond the capacity of individuals to process (or even produce) and often cannot be retrieved via any other source. Additionally, with advancement in the data management and revision control systems, data can to be stored with good documentation, including full code-books describing the variables and codes that have been used, and easily accessible recording methods [78]. However, completeness and quality are the two major concerns with archival data [79]. Archives offer only the data that is recorded, which is beyond the capacity of researchers to assess for completeness and quality. This might raise limiting factor in using archival data for research, unless necessary measures are taken to mitigate them.

In this thesis, archival data is used to conduct case study and constructive research, because, OSS projects often consist of a wide range of repositories, coming with a large number of versions reflecting their development and evolution history [II]. While it is a challenge to acquire tacit knowledge from developers and users of such projects due to their distributed nature, OSS communities often produce a rich set of repositories as a byproduct of their development activities. In addition to the source code and other software artifacts, there are repositories that contain the development history, such as bug reports, mailing lists, and revision history logs [II]. These repositories are maintained through data management and revision control systems [II].

OSS projects often maintain their project data using either of the following means: use own data management systems and make them publically available through internet, or use third party hosting facilities to host and maintain project data. These sources often contain a plethora of information on both the underlying software and the associated communication and development process [22] [3]. In literature [56] a great emphasis is given to leveraging these repositories for deriving technical dependencies as well as developers' coordi-

[1]http://www.oxforddictionaries.com/definition/english/archive?searchDictCode=all

nation patterns. The repository data is often longitudinal, allowing for analysis along the whole project evolution phases. These data sources are highly accepted and utilized medium for empirical studies on OSS projects [32] [37] [93]. In Section 2.2.1 and Section 2.2.2 a discussion on the available data sources is presented in relation to the case study projects. Additionally, data from the Ruby Gems Ecosystem and the OHLOH repository are also collected. Table 3.1 offers a summary listing on these used data sources, and link to included publications in which further details can be found.

3.4 Data Analysis Methods

Data analysis for quantitative and qualitative data is conducted in different ways. Since the research methods selected for this thesis are qualitative in nature, qualitative data analysis methods are used. The following data analysis methods were used.

3.4.1 Data Synthesis for Systematic Literature Review

Data synthesis in an SLR involves collating and summarizing the data collected from primary studies [52]. Activities comprising data synthesis should be specified in the review protocol. Extracted information (intervention, population, context, sample sizes, outcomes, study quality) need to be tabulated. This tabulation of information should be consistent with the review questions, and highlight similarities and difference between study outcomes. Synthesis can be both quantitative and qualitative (or descriptive). However, it is sometimes possible to complement a descriptive synthesis with a quantitative summary. For this thesis, the later approach is followed.

3.4.2 Content Analysis

The focus of content analysis is to gather information and generate findings. The gathered information (content) can be any documented information and different categories containing content are constructed for analysis.

A content analysis can either be *inductive* or *deductive* analysis [29]. In inductive analysis, dominant themes of the collected data is looked for and identified. By using inductive reasoning and experience the researcher reviews the data for unifying ideas. On the other hand, in deductive analysis, the researcher preselects the themes and sub-themes, preferably in the form of research questions. The researcher then walks through the collected data and records every support relevant to the research questions. This analysis of the data can be supported by computer aided data analysis programs [78].

For the purpose of this thesis, deductive content analysis approach has been used.

3.5 Research Classification

The results in this thesis have been reached through the use of the presented research design and methods. Table 3.2 illustrates the link between each publi-

Table 3.1: OSS data sources used in this thesis

Source Type	Data Sources	Purpose	Collected From	Purpose in the thesis	Ref. Paper
Project's internal sources	Version Control System (CVS, SVN)	Maintain and track changes of the source code over the releases.	FreeBSD, NetBSD, OpenBSD, FFMpeg, Eucalyptus.	The source code for each stable release of the selected projects was downloaded, parsed and analyzed.	[III] [IV] [II] [VIII] [VIII]
	Email Archives (developer)	To trace task-oriented communication and coordination activities of the developers during project evolution.	FreeBSD, NetBSD, OpenBSD, FFMpeg, Eucalyptus, GStremer.	The communication records of each developer has been tracked (both technical and non-technical).	[III] [IV] [II] [VIII] [VIII]
	CVS / SVN Commits	To trace contributions from each developer. In some projects, commit records are made available thorough specific email archives. e.g., BSD projects.	FreeBSD, NetBSD, OpenBSD, FFMpeg.	Commits records for each developer is collected.	[III] [IV] [II] [VIII] [VIII]
	Bug reports (e.g., Bugzilla, Zira)	To trace reported bugs and their resolution. Again, for some projects, these reports are made available thorough specific email archives. e.g., BSD projects.	FreeBSD, NetBSD, OpenBSD, FFMpeg, Eucalyptus.	Bug reports submitted and resolved by each community member (either developer or user) is recorded.	[III] [IV] [II] [VIII] [VIII]
	Email Archives (users)	To trace user communication and contribution history.	FFMpeg, Eucalyptus.	For each user of the software, email communication history is collected.	[II] [VIII]
	Gem Specification, Git issues and pull requests	To trace Gems dependency, and developer's contribution and collaboration.	Ruby Gems Ecosystem.	Specification for each Gem (equivalent of a Project) in the ecosystem is collected. A specification contains metadata that include the gem name, dependencies to other gems, and URIs for the gem. Issues and pull requests submitted and resolved by the developers of a Gem have been collected.	[V]
External sources (e.g., OHLOH, Source-Forge)	OHLOH	OHLOH is a free, public directory of OSS projects and the respective contributors. It collects and maintains development information of over 400 thousand FLOSS projects, and provide analysis of both the codes history and ongoing updates, and attributing those to specific contributors. It can also generate reports on the composition and activity of project code bases. These data can be accessed and downloaded through a set of open API which handles URL requests and responses.	OHLOH	Collected project data (e.g., domain, size, license, language) and developer contribution record (e.g., kudo rank, total contribution, contributed project list, number of commits).	[VI] [VII]

cation to that of the research design, research methods, data collection methods, and data analysis methods.

Publication [I] is a review publication. The primary focus of this study is to classify and structure the state of the art literature in the context of OSS evolution and assess the coverage and, find gaps in the current research. For this study, systematic literature review method is used. However, each of the other studies is accompanied with related literature review either as a starting point for designing the study or as a validation step to compare the outcome against existing knowledge. The difference lies in that for those studies a less systematic process was applied in searching and analyzing related literature than for publication [I].

Case study research is the primary research method applied in publications [III], [IV], [VI], and [V]. Archival data (as presented in Section 3.3.2) are analyzed with content analysis approach. Six OSS projects, the OHLOH repository and one ecosystem (as listed in Table 3.1) are selected as case study subjects. Selection of these cases are mostly guided by the following criteria: (a) the code base of these projects have undergone continuous development, improvement, and optimization for past ten to twenty years, (b) these projects have been developed and maintained by a large team of individuals, (c) the properties of a forked project hold for the three BSD projects, (d) these projects have extensively been used in earlier research on the evolution of OSS projects [49] [44], and (e) results reported in this study can be stressed to OSS projects having similar properties, e.g., forking, domain, community structure, and size.

The basic theme of these publications is to examine and evaluate the extent to which socio-technical dependency holds at different abstraction levels of OSS projects. For this, a detailed examination of the technical artifacts as well as communication history at organizational level is presented; for OSS projects, such data can only be collected from project archives, thus archival data and content analysis methods seem most suitable for these studies.

Constructive research approach has been applied in publications [II], [VII] and [VIII]. Thematically, these publications propose models, methods, and framework for sculpting, analysis and visualization of the socio-technical perspectives of OSS projects. Additionally, to demonstrate the applicability of the proposals corresponding tools (e.g., Binoculars and Pomaz) are built and tested with the OSS project data.

Table 3.2: Research classification

Publication	Research Design	Research Methods	Data Collection Method	Data Analysis Methods
[I]	Qualitative	Systematic Literature Review	Systematic Literature Search Strategy	Data Synthesis for SLR
[II]	Qualitative	Constructive Research Approach	Archival data	Deductive Content Analysis
[III]	Qualitative	Case Study (Positivist)	Archival data	Deductive Content Analysis
[IV]	Qualitative	Case Study (Positivist)	Archival data	Deductive Content Analysis
[V]	Qualitative	Case Study (Positivist)	Archival data	Deductive Content Analysis
[VI]	Qualitative	Case Study (Positivist)	Archival data	Deductive Content Analysis
[VII]	Qualitative	Constructive Research Approach	Archival data	Deductive Content Analysis
[VIII]	Qualitative	Constructive Research Approach	Archival data	Deductive Content Analysis

Chapter 4

Introduction to Included Publications

Next, the main results and contributions of this thesis are summarized per included publication. In the following sections, the main results causative to each research question are presented. The discussed results are based on the observations and conclusions drawn in the included papers. Table 4.1 presents the synopsis of the results in relation to the research targets and corresponding research questions. Pointer to the publications and to the sections of the introductory part of this thesis that offer elaborated discussion are listed in column 3 of the table.

Additionally, publications presented in this chapter are roughly ordered in accordance to the logical ordering of the research targets presented in Figure 1.2. To help readers navigate thorough the publications, and to draw link between reported results and the research targets, an explicit mapping between the two (as shown in Figure 1.2) is offered bellow,

Publication [I] reports the results obtained from the SLR that primarily targeted to address RQ.1 formulated under the *theoretical basis* target. This study synthesizes the study results under the three dimensions of OSS projects to set the basis of this study, and to forecast future direction of works under each of the dimensions. In Publication [VII] these results are extended further to formally define the dimensions.

Publications [II], [III], [IV], and [VII] address RQ.2 defined under the *Measurement* research target. Publication [II] defines the generic OSS data model that mimics social, and technical dimensions of a project. In addition, this publication presents the graph theoretic methods that establish inter-relationships between the two dimensions. Publications [III], and [IV] propose the mathematical model to measure socio-technical congruence. In doing so, this model utilizes the methods and the data model defined in Publication [II]. Finally, Publication [VII] proposes the framework to integrate the data model, methods and the mathematical model to achieve fully automated analysis and interactive exploration of socio-technical dependency.

Publications [III], [IV], [V], [VI], and [VII] address RQ.3 within the *Empirical Insight* research target by mobilizing the proposed models and methods. These publications as a whole accumulated the empirical results comprising distinct OSS project context (as defined in Figure 1.1) to offer proposals, trends and patterns for socio-technical dependency in such projects. Publication [III] measures socio-technical congruence for a singleton project (e.g., the FreeBSD project), whereas, publication [IV] performs the same for forked BSD projects (e.g., FreeBSD, NetBSD, and OpenBSD projects). Publication [V] determines congruence in the context of Ruby Gems Ecosystem. Publication [IV] and [VI] measure project resemblance for frocked project and for an arbitrary sample of OSS projects extracted from OHLOH repository, respectively. Finally, publication [VII] evaluates the framework against two OSS projects (e.g., FFMpeg and GStreamer).

Finally, publications [VII] and [VIII] answers the RQ. 4 under the *Proposal* research target. In publication [VII] an architectural desing and corresponding tool implementation is done based on the proposed framework, whereas publication [VIII] demonstrates the implementation of the data model and the methods proposed in publication [II]. In addition, publication [VIII] distills recommendations and guidelines based on the lessons learned during the process of design, implementation and evaluation of the tool.

4.1 Publication [I]: A SLR on the evolution of OSS projects.

Introduction:
This paper reports on a systematic literature survey aimed at the identification and structuring of research on the evolution of OSS projects.

Objective:
The main objective, and thus the contribution of this study is to produce a systematic reporting of what constitutes the key contributions, the main research gaps, and potential future directions within the domain of OSS project evolution. Special attention is given for clustering studies according to the social, technical and socio-technical dimensions of OSS projects (defined in Section 2.2).

Method:
An obvious choice of methodology for this study is the Systematic Literature Review (SLR). A generic methodology for conducting a SLR is presented in Chapter 3. However, a meticulous instantiation of this method can be found in the publication [I]. In brief, this study poses 11 research questions, and defines a study protocol following the guidelines presented in [52] [23]. This protocol is piloted before commencing the actual study [I]. Relevant articles are searched from seven digital libraries. Articles are collected for thirteen years of span starting from 2000 till 2013. At the end of the search process, a total of 101 articles are selected for final review. The review process is the strict instantiation of the review protocol in order to minimize the reviewer bias [52] [23].

Result(s):
With the increasing dominance of open source software, the interest of understanding the evolutionary patterns of such projects is getting more and more momentum for the last decade. Study focus of these studies can be classified

Table 4.1: An overview of the results

Research Target	**RQ**	**Addressed in**		**Contribution**
		In Publication	**In the introductory part**	
Theoretical Base	RQ.1	[I] [VII]	Section 2.2, 4.1, 5.1 (RQ.1)	- Offers a solid basis for undertaking the research reported in this thesis. - Defines the three dimensions (e.g., social, technical and socio-technical) in the context of OSS projects. - Offers a categorical listing of the gaps and opportunities for future research on the topic.
Measurement	RQ.2	[II] [III] [IV] [VII]	Section 2.2.6, 4.2, 4.3, 4.4, 4.7, 5.1 (RQ.2)	- OSS data model to mimic the social and technical dimensions and their inter-relationships. - A generic set of graph theocratic methods to explore social, technical and socio-technical perspectives. - A mathematical model to measure socio-technical congruence. - A framework to fully automate the data driven analysis of OSS project.
Empirical Insight	RQ.3	[III] [IV] [V] [VI] [VII]	Section 4.3, 4.4, 4.5, 4.7, 5.1 (RQ.3)	- Measure socio-technical congruence within the defined context of OSS projects. - Measure inter-project resemblance.
Proposal	RQ.4	[VII] [VIII]	Section 4.7, 4.8, 5.1 (RQ.4)	- Architectural design concepts based on the proposed data model and the framework. - Semi-automated and automated tools that support data driven analysis and interactive exploration of socio-technical dependencies. - Recommendations and guidelines for the design and implementation of analytical tools for OSS data analysis and visualization.

broadly into four dominant themes, namely technical (or software) evolution, social (or community) evolution, socio-technical (or co-evolution) evolution, and prediction. However, technical evolution is by far the most dominant research theme among the four.

Methodologically, these studies predominantly follow empirical research method in their investigation. Data sources for these studies are collected from the concerned project repositories. This includes source code management system, commit records, change logs, bug tracking systems, and mailing archives. Selected case study projects are delimited to flagship OSS projects that are large in size with a large user and developer community and belong to the popular application domains. Highest counts go to Linux, Eclipse, Apache, Ant, Mozilla, GNOME, KDE, and ArgoUML, that fall within the domain of Operating Systems (OS), Application Software, Integrated Development Environments (IDE), Application Servers, Libraries, Desktop Environments, and Frameworks.

Within the technical evolution, the studies mostly verify the fitness of Lehman's law of software evolution for OSS projects. Originally, Lehman proposed five laws of software evolution, though ongoing investigation has produced three more. Briefly, this law states the followings: continuing change, increasing complexity, self-regulation, conservation of organizational stability, conservation of familiarity, continuing growth, declining quality, and feedback system. An elaborated discussion on these laws can be found in [58]. Empirically validated metric suits are most commonly used to evaluate these laws. Metrics include for instance, source code metrics, complexity metrics, object oriented metrics, and project and product metrics. Reported results have both conformance (either complete or partial) and contradiction with the laws of software evolution. For instance, the growth rate of OSS varies between superlinear (i.e., greater than linear) and sub-linear (i.e., less than linear) [58]. This has both conformance and contradiction with the second and sixth law of evolution. Further details can be found in the publication [I]. Additionally, other studies examine the evolution of code complexity, code cloning, code quality and documentation.

On the social evolution, studies are concerned on determining the motivation of developers to participate in a project, structure of the community, changes of the community dynamics during project evolution, and sustainability of the community.

However, study on socio-technical dependency receives only limited attention as compared to their study in isolation. Within the limited focus, the following observations are reported. It is identified that the number of contributors grow with the growth in product size. A study analogous to this reported that the increase of new packages and reported bugs are highly coherent with the increase of contributors and active users, respectively. Also, the increase in documentation and modularization levels are obliged to the rising number of developers and their contribution to a project.

On the validity of the included studies, it is found that most of the studies suffer from the external validity threat. In other words, the reported results can not be generalized to the extended population of OSS projects. One reason for this is that these studies use flagship projects as case study, which comprises only fraction of the whole spectrum of the OSS projects.

Statistical details on the reported results can be found in publication [I].

Finally, the open research areas within the concerned domains are elaborated in Section 5.3.1.

4.2 Publication [II]: The OSS data model and the methods.

Introduction:
While it is a challenge to acquire tacit knowledge from developers and users due to their distributed nature, open source development and user communities often produce a rich software repository as a byproduct of other activities. These repositories hold data on both technical and social interaction across the entire spectrum of the project's evolution. However, organization and representation of such data with unified analysis methods are yet a challenge.

Objective:
This paper explores the opportunity to derive data model and methods to unify data representation and analysis specific to the study of socio-technical dependency.

Method:
Methodologically, this paper follows a constructive approach, which consists of (a) a proposal of a model to offer an unified representation of an OSS project's structure, (b) a set of methods based on graph theoretic concept to examine socio-technical dynamics of such projects, and finally, (c) a tool *Binoculars* implementation to demonstrate the applicability of the approach with data extracted from two OSS projects, e.g., FFMpeg and Eucalyptus.

Result(s): The data model, as presented in Figure 4.1, is divided into domain specific and domain independent models. In the former, the two dimensions of OSS projects are modeled through the abstraction of corresponding project artifacts, and in the later, everything is modeled as mathematical graphs.

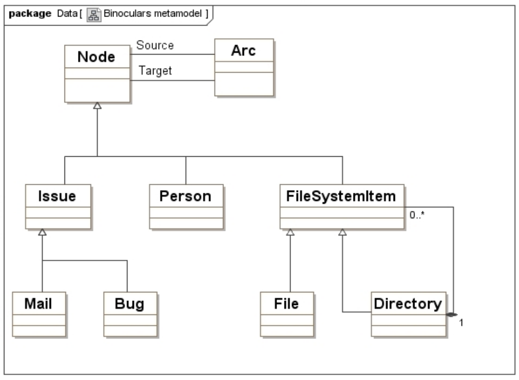


Figure 4.1: OSS project data model

Within the domain specific model, abstract class FileSystemItem and its two concrete subclasses, File and Directory, are used to mimic the technical domain (Section 2.2.2). The composition relation between Directory and FileSystemItem allows to model the real file system of the repository. To model the social dimension (Section 2.2.1), classes Person and Issue (with subclasses Mails and Bug) are used. While class Person represents each community member's portfolio, Issue class along with its subclasses tracks the communication and collaboration records.

The domain independent model consists of classes Node and Arc as depicted in the upper part of Figure 4.1. Due to the fact that the classes in domain specific model are the subclasses of abstract class Node, any pair of entity within the domain specific part can have relationship links (i.e., instance of Arc) based on a given criteria. For instance, a relationship graph among the developers (from Person class) can be drawn based on common email conversations (taken from Mails class) among them. Or, one can derive relationships among code files (from FileSystemItem) based on attribute referencing, and so on. Instantiation of the domain specific model can be done with the repository data discussed in Sections 2.2.1, 2.2.2 and 3.3.2. For instance, CVS/SVN checkout of the source code can be used to instantiate the FileSystemItem.

In order to derive relation among entities in domain specific model (as presented above), a set of six methods is defined. These methods are intended to explore the three dimensions of OSS projects. A complete listing of these methods along with implementation details can be found in the publication [II]. To offer a brief demonstration on the applicability of these methods, three of them are presented bellow with example,

The first method is called *interConnect* method. This method derives, for instance, the relation between the developers and the artifacts to which they have contributed. For example, Figure 4.2(a) shows the relation among the developers (D1, D2 and D3) and the code files (F1, F2, and F3) in which they have contributed to. Similarly, Figure 4.2(b) shows to which emails (M1, M2 and M3) the developers have posted or replied. In other words, interConnect operation generates relation graphs that show the trend and intensity of developers participation in the project.

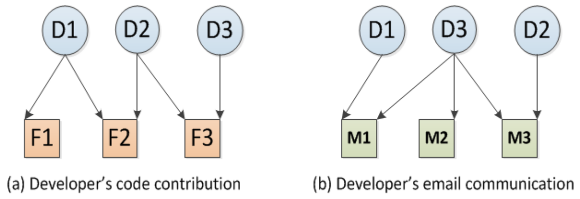

D1
D2
D3
F1
F2
F3
(a) Developer's code contribution
D1
D3
D2
M1
M2
M3
(b) Developer's email communication

Figure 4.2: Example of InterConnect method

The second method is called *intraConnect* method, which derives the explicit or the implicit network from a given graph generated using interConnect method. The two example networks shown in Figure 4.3(a) and (b) are derived from the graphs presented in Figure 4.2(a) and (b), respectively. The network in Figure 4.3(a) is an implicit coordination network which shows developers' relation based on common code contribution, whereases the second network

(Figure 4.3(b)) is an explicit coordination network showing developers relation based on common email conversations. In other words, the first graph identifies who should communicate with whom based on their task dependencies and contributions, and the second graph portrays the actual communication and interaction that took place among them.

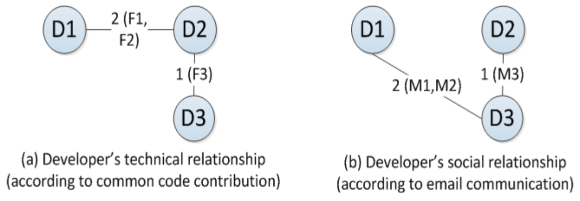

D1
2 (F1, F2)
D2
1 (F3)
D3
(a) Developer's technical relationship (according to common code contribution)
D1
D2
2 (M1,M2)
1 (M3)
D3
(b) Developer's social relationship (according to email communication)

Figure 4.3: Example of IntraConnect method

The third method is the *intersection* method. This method, as per the name goes, identifies the common relations between a given pair of graphs. For instance, Figure 4.4 shows the graph produced by taking intersection operation between the graphs shown in Figure 4.3. To put this method in practical perspective, the generated graph reveals the extent to which developers actual collaboration (Figure 4.3(b)) reflects the need for collaboration identified by the technical domain (Figure 4.3(a)).

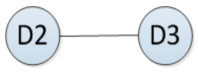


Figure 4.4: Example of intersection method

The implemented tool *Binoculars* and the case study transcript can be found in the publication [II].

4.3 Publication [III]: Socio-technical congruence in an OSS project.

Introduction:
The identification and management of work dependencies is a fundamental challenge in software development organizations, specially for OSS projects [16]. This paper builds on the idea of socio-technical congruence, as discussed in Section 2.2.5, to examine the relation between the structure of the technical domain to that of the coordination patterns of the developers, and vice-versa.

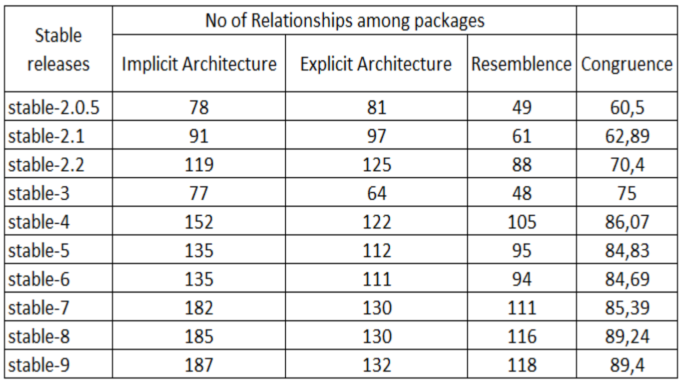
Objective:
The motive here is to propose a mathematical model to measure socio-technical congruence during the evolution of an OSS project. This method is inspired by the Conway's Law and the organizational theory method [17]. Furthermore, empirical evaluation of the method is carried out with repository data extracted from FreeBSD project.

Method:
This paper proposes and utilizes the mathematical model presented in Section 2.2.6 to measure socio-technical congruence. In brief, the paper derives the architectures and coordination networks (both implicit and explicit) based on the repository data extracted from FreeBSD project. Then, the congruence is measured using the model presented in equation 2.1.

The repository data comprises SVN checkouts of the source code for each stable release of the software, SVN commit history, and developers email communication. In FreeBSD project, several email archives are maintained to record commit history, bug reporting and developers' communication. These email archives contain data since 1994 and are updated every week. For this study, SVN checkout for 10 stable releases, and 30 email archive data is extracted. The email archives record the CVS/SVN commit history, bug reporting and developers communication (email and chat). A detail discussion on these data sources, data collection and processing is presented in the publication [III].

Result(s):

This study first verifies the extent to which the communication patterns of the developer community is congruent with the actual architectural dependencies. This congruence is measured between the explicit architecture and the implicit architecture for each stable release of the FreeBSD project (defined in Section 2.2.6). The reported congruence measure is shown in Figure 4.5. In this figure,



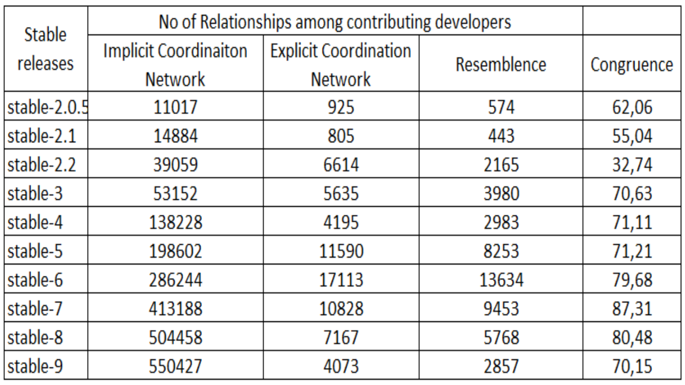
Stable releases	No of Relationships among packages			
	Implicit Architecture	Explicit Architecture	Resemblence	Congruence
stable-2.0.5	78	81	49	60,5
stable-2.1	91	97	61	62,89
stable-2.2	119	125	88	70,4
stable-3	77	64	48	75
stable-4	152	122	105	86,07
stable-5	135	112	95	84,83
stable-6	135	111	94	84,69
stable-7	182	130	111	85,39
stable-8	185	130	116	89,24
stable-9	187	132	118	89,4

Figure 4.5: Congruence between the developer's Communication Pattern and the Software Architecture in FreeBSD project

column 2 and 3 show the number of relations between packages identified by the respective architectures. The resemblance (in column 4) shows the number of common relations between the two, and the congruence (in column 5) reflects the socio-technical congruence measure between the two with respect to the explicit architecture (Section 2.2.6). According to this figure, the congruence measure varies between 75% to 82% starting from stable release 3 onward (first three releases are ignored as outliers [III]).

Based on the results it can be inferred that to a great extent the communication patterns of the contributing members within the community is due to the communication needs established by the concrete architecture.

Second, this study examines the extent to which the explicit architecture is congruent with the communication patterns of the developer community. To verify this, the congruence is measured between the implicit coordination network and the explicit coordination network for each stable release of the FreeBSD. Figure 4.6 reports the observed congruence level. According to this figure, the



Stable releases	No of Relationships among contributing developers			
	Implicit Coordinaiton Network	Explicit Coordination Network	Resemblence	Congruence
stable-2.0.5	11017	925	574	62,06
stable-2.1	14884	805	443	55,04
stable-2.2	39059	6614	2165	32,74
stable-3	53152	5635	3980	70,63
stable-4	138228	4195	2983	71,11
stable-5	198602	11590	8253	71,21
stable-6	286244	17113	13634	79,68
stable-7	413188	10828	9453	87,31
stable-8	504458	7167	5768	80,48
stable-9	550427	4073	2857	70,15

Figure 4.6: Congruence between the Software Architecture and the Developers Communication pattern in FreeBSD project

congruence values remain between 71% and 88% starting from stable release 3 (again ignoring the first three releases as outliers [III]). This result is closely tied to the results reported in Figure 4.5. This indicates that socio-technical congruence in reverse also holds for FreeBSD. In other words, the communication patterns of the contributing developers simulates the underlying architectural dependency of the software to a considerable extent.

Therefore, developers communication structure in FreeBSD project affects the architectural orientation of the produced software. However, utilizing the implicit architecture to recover the explicit one (a typical example of reverse engineering) remains an unsolved problem, and requires further investigation. In this study the implicit architecture is overestimating the explicit architecture for all the releases. For instance, in stable release 4 (see Figure 4.5), the implicit architecture identifies 153 relations among the 19 packages, whereas in reality there are only 49. This is due to the fact that all possible communication channels are used in this study to derive the explicit coordination network, which creates relations among developers that are not due to technical discussion. However, the selection is made based on the empirical results which suggested that developers rely heavily on informal ad hoc communications [43]. As a side effect, the implicit architecture derived from explicit coordination network holds extra links between packages. However, this side effect can be remedied through selective listing of communications of developers that limit within development discussion only. A further discussion on this is provided in Section 5.4.

4.4 Publication [IV]: Study of socio-technical congruence and project resemblance for forked OSS projects.

Introduction:
This paper empirically investigates the socio-technical dependencies for forked OSS projects within the scope of both intra-project and inter-project level (see Section 1.2).

Objective:
For intra-project level (i.e., within a project setup), socio-technical congruence is measured during the evolution of each forked project. The rationale here is to extend the observation reported in [III] to projects that exhibit both forking relationship and resembling project properties.

For inter-project study, resemblance in terms of software architecture and the community are examined for a given pair of forked projects. For forks, source code and hence also, the architecture are the same at initial stage. Thus it would be crucial to know whether any resemblance remains between them as the projects evolve.

Similarly, during fork, the parent community divides. This division of the communities is often followed by a rebuild and restructuring process. However, both the communities belong to the same origin, and individual developers have similar interest and technical expertise. Additionally, both the forked projects belong to the same application domain with similar technical framework. Thus it might be possible that the fragmented communities would maintain collaboration and contribute during the projects' evolution. Therefore it is interesting to find the extent to which communities of the forked projects collaborate.

Method:
This study follows a case study approach with data collected from three flagship OSS projects from BSD family, namely, FreeBSD, NetBSD, and OpenBSD.

To measure the socio-technical congruence, the approach presented in Section 2.2.6 is applied. This measurement is done per stable release per project.

However, to identify resembling architecture and the community, equation 2.1 is adopted. In the following a brief description of the approach is given,

Resemblance between the architectures for each pair of forked projects is studied at three abstraction levels. These are, package level, first directory level and n_{th} directory level (i.e., the last directory where the code files reside). This distinction is based on the hypothesis that forked projects might maintain homogeneous architectural design at a higher level of abstraction (e.g., package level), yet getting liberated at detailed architectural level (e.g., n_{th} directory level). The adoption of equation 2.1 is presented below with an illustrative example. The FreeBSD release 6 and NetBSD release 3.0 have 19 and 22 packages, respectively. Thus $|FreeBSD - release - 6|$ = 19 and $|NetBSD - release - 3.0|$ = 22. The intersection operation between these two architectures resulted in 16 packages that have the same names. Then the resemblance is calculated by taking each of these architectures as a reference architecture. For the above example, FreeBSD release 6 has 84.21% (16/19*100) and NetBSD release 3.0 has 72.72% (16/22*100) resemblance with each other.

A similar approach is followed to compare the communities. Before applying the equation, developer list per release is produced for each forked project. Then for a given pair of project releases, the union operation in the numerator identifies the number of developers whose names are lexically identical. Then resemblance is measured with respect to the real number of developers counted for the stable releases of the projects.

Further details on study design, data collection, processing and analysis are presented in the publication [IV].

Result(s):
The socio-technical congruence measure for each fork project offers conformance to the congruence measure observed for FreeBSD project in publication [III]. Figure 4.7 shows the congruence between the developer's communication patterns and the software architecture (i.e., between implicit architecture and explicit architecture) during the evolution of the three projects.

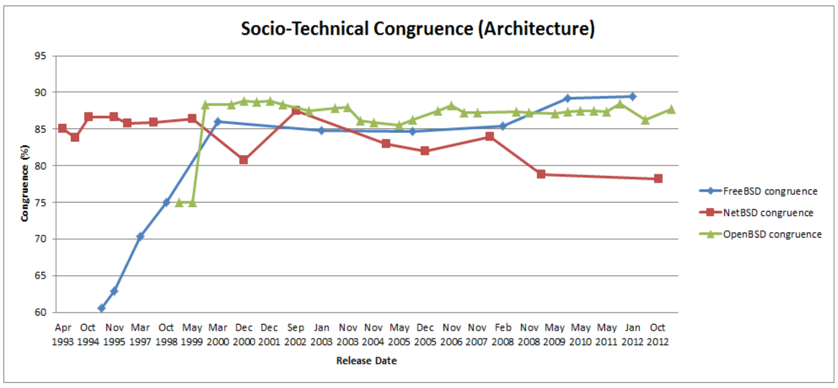


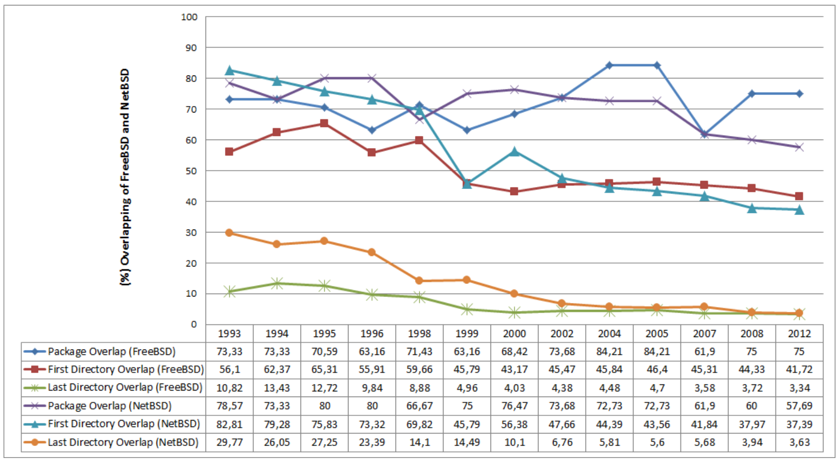
Figure 4.7: Congruence between the developer's Communication Pattern and the Software Architecture in the BSD projects

According to this figure, NetBSD (the maroon line) has a high and stable congruence from the very beginning of the project, which remains between 85% and 87,5% for the first twelve releases. However, for the recent releases, the project experienced a decrease in congruence which requires further investigation (a discussion on the topic is presented in Section 5.3.1).

FreeBSD and OpenBSD (the blue and the green line, respectively) exhibit a similar trend of congruence. For both these projects, there are drifts in congruence during the initial releases, which then took off and remained stable for subsequent releases. For instance, FreeBSD has a stable congruence level remained between 84,83% and 89,4% starting from stable release 5. For OpenBSD it is between 85,56% and 88,78% starting from third stable release. First few releases are considered outliers [IV].

Such high and stable congruence points to the fact that the implicit architectures derived from the communication patterns of the developer community effectively represents the explicit one. Therefore, it can be accredited that for the BSD forked projects, to a considerable extent the communication of the developer community may actually be due to the coordination needs as identified by the architectural dependencies. Congruence measures in reverse reveal a similar pattern.

Results on architectural resemblance reported that at higher abstraction level (e.g., package level) the architectures of the forked projects maintain high correspondence between them, which remains consistent as the projects evolve. However, at directory levels the design and implementation became more disjoint and independent. For instance, consider Figure 4.8, which shows architectural resemblance between FreeBSD and NetBSD. As can be seen from this figure, the package level congruence remain high throughout the project's evolution. For FreeBSD it remains between 61,9% and 84,21%, whereas for NetBSD it is between 57,69% and 80%. Contrary to this, directory level overlapping (d_1 and d_n) reveals a different trend. For example, at d_n level, the resemblance goes down to 3,63% and 3,34% from 29,77% and 10,82% with evolution of FreeBSD and NetBSD project, respectively. For other combination of forked projects almost similar trend is observed.



	1993	1994	1995	1996	1998	1999	2000	2002	2004	2005	2007	2008	2012
Package Overlap (FreeBSD)	73,33	73,33	70,59	63,16	71,43	63,16	68,42	73,68	84,21	84,21	61,9	75	75
First Directory Overlap (FreeBSD)	56,1	62,37	65,31	55,91	59,66	45,79	43,17	45,47	45,84	46,4	45,31	44,33	41,72
Last Directory Overlap (FreeBSD)	10,82	13,43	12,72	9,84	8,88	4,96	4,03	4,38	4,48	4,7	3,58	3,72	3,34
Package Overlap (NetBSD)	78,57	73,33	80	80	66,67	75	76,47	73,68	72,73	72,73	61,9	60	57,69
First Directory Overlap (NetBSD)	82,81	79,28	75,83	73,32	69,82	45,79	56,38	47,66	44,39	43,56	41,84	37,97	37,39
Last Directory Overlap (NetBSD)	29,77	26,05	27,25	23,39	14,1	14,49	10,1	6,76	5,81	5,6	5,68	3,94	3,63

Figure 4.8: Architectural resemblance between FreeBSD and netBSD

Investigation of the community resemblance reveals that participation of community members in both projects remains stable within a given range. To be specific, consider Figure 4.9, which shows community overlapping between NetBSD and OpenBSD projects. For NetBSD project it remains between 42,69% and 50,3%, whereas for OpenBSD it is between 34,42% and 38,31%. A similar trend is observed for other comparable projects. Details regarding the results are presented in the publication [IV].

Based on these observations, it can be affirmed that the traditional perception of forking in OSS projects, which is thought to have negative stimuli for sustainable evolution of the projects [77], can be effectively remedied though (a) maintaining a consistent and cohesive abstract architectural design to form a common ground of collaboration among the forked projects, (b) adopting a collaboration model in which members of a project could participate in other forks, and (c) maintaining a consistent and high socio-technical congruence within the project.

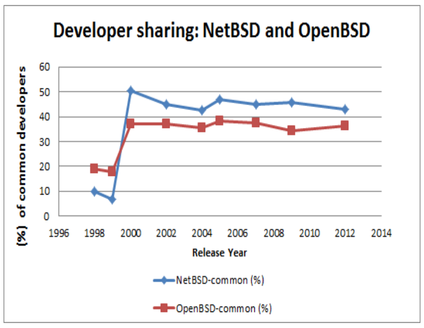


Figure 4.9: Community resemblance between NetBSD and OpenBSD projects

4.5 Publication [V]: Socio-technical congruence in the Ruby Ecosystem.

Introduction:
This paper offers an empirical analysis of the socio-technical congruence within the context of an OSS ecosystem.

Objective:
The objective is to measure the socio-technical congruence at the ecosystem level where the technical domain models the relations among the projects in the ecosystem. Therefore, this study measures the congruence between the developers coordination network to that of the interdependency among the ecosystem projects (explicit architecture in this case) to which they contributed. Overall, the following two investigations are piloted.

First, we verify whether there exist dependencies among ecosystem projects both from social and technical perspectives. The rationale here is that for a singleton project, the dependency between the social and technical dimensions are desirable. Because, a project often organizes itself around the product's architecture and task dependencies. However, dependencies among the projects in a ecosystem are not so explicit. This in turn could minimize inter-project collaboration. Therefore, this verification would justify the ground for this study by identifying the number of relations among the ecosystem projects, as well as among the developers participating in them.

Second, we measure the socio-technical congruence to exemplify the extent to which developer's interaction among the ecosystem projects approximate the projects' dependencies.

Method:
The method presented in Section 2.2.6 is applied to measure the socio-technical congruence. As a unit of study the Ruby Gems ecosystem is selected. A gem in this ecosystem is a software package that contains a Ruby application or library implemented using Ruby programming language [V]. Each gem in the Ruby ecosystem is equivalent to a project.

As the context of the study is an ecosystem instead of a singleton OSS project, the explicit and the implicit architecture for this study shows relations among the gems. For explicit architecture, a relationship between two gems represents development and runtime dependencies. Here, a development dependency defines a gem that is necessary at development time for further development, whereas a runtime dependency represents a gem that is necessary at runtime.

Data for this study is collected to derive the Ruby ecosystem architectures and the coordination networks that correspond to the one presented in Section 2.2.6. For the former, gems specification data is collected, that contain project metadata describing the gem name, dependencies to other gems, and URIs for the gem (usually a Github link). For the latter, Github is used. In GitHub, community collaboration is facilitated via among others issues and pull requests created by GitHub users. In this study, such issues and pull requests are extracted for each gem that is hosted in GitHub. This study collects data for 12,520 Ruby gems for each of which both the specification and GitHub issue and pull request records are found. Further details on the study method is presented in the publication [V].

Result(s):
The verification of having technical and social dependencies among the gems is done by examining the explicit architecture and the explicit coordination network, respectively. The architecture shows development or runtime dependencies among the gems, whereas the coordination network shows developers' communication among the gems, as seen by the GitHub issue tracking system.

The outcome reveals 141,029 relations among 12,520 gems for the explicit architecture, whereas the explicit coordination network shows 186,136 relations among 55,454 developers. Due to such staggering number of relationships both at social and technical domains, it is logical to measure socio-technical congruence.

However, reported congruence measure reveals a different picture. It is noted that socio-technical congruence is inversely proportional to the strength of coordination among the developers in the network. Here the strength of a relation measures the number of interactions taken place between a given pair of developers. Figure 4.10 shows the congruence level with respect to the developer coordination strength in the explicit coordination network. According to this figure, for edge weight ≥ 2, the congruence measure is 76.2%, which drops sharply with the increasing edge weight limit. For instance, congruence value drops to 46.3% for edge weight ≥ 8, which goes down as low as 36.1% for weight ≥ 19.

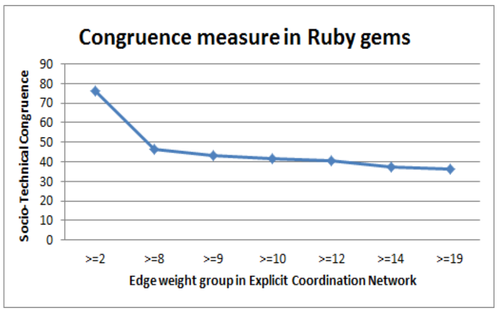


Figure 4.10: Socio-Technical Congruence in the Ruby ecosystem

As explained earlier, the edge weight in the explicit coordination network measures the strength of collaboration among the developers. Thus, communication and collaboration get strongly tied with the increased edge weight in this network. In this connection, the stated congruence measure shows that congruence decreases with the increased in developer collaboration.

A fine grained analysis to this matter reveals that the developers who have extensive collaboration (as seen in the Explicit Coordination Network) belong to the same gem or related gems that have dependency at implementation level. For instance, a network where edges that have weight ≥ 19 contains 362 developers. Around 79% of these developers (285 developers out of 362) work on the same gem jdbc-jtds. Therefore, it is resonable that these 285 developers should have extensive communication and collaboration. This observation is in line with the prior results reported in Section 4.3 and 4.4. This perception might partially explain the low congruence for the ecosystem.

Based on these results, the following observation can be devised: at ecosystem level, the collaboration patterns among the developers at inter-project level is not necessarily shaped by the communication needs indicated by dependencies among the gems. However, there is a strong indication that developer collaboration often exists within a project or projects that have implementation level dependencies.

4.6 Publication [VI]: Measuring projects' resemblance from developer contribution.

Introduction:
In OSS domain, developers often contribute to multiple projects. Such participation should be guided by some decisive properties of the projects. Hereby, the hypothesis is that these properties should resemble the projects that are contributed by the same group of developers. This paper investigates whether developers' participation in multiple OSS projects could identify project resembling properties.

Objective:
OSS projects are often classified based on their resembling properties. Popularly cited properties are the domain, language, project size, license, and project rating as discussed in Section 2.2.8. Due to open collaboration and participation model, OSS developers often participate in multiple projects. In this paper, the information of *which developers contribute to which OSS projects* are explored to identify the resembling projects. The rationale is that these resembling project properties should stimulate developer's participation, as developers should participate in projects which falls within their technical expertises (e.g., known languages, domain), and are affluent (e.g., large size and rating).

Method:
In order to address the issue, data is collected from OHLOH repository. Information of 530 top ranked OSS developers (according to their Kudo ranking [VI]) are collected. Additionally, meta-data of 4261 OSS projects that are contributed by these 530 developers are also collected. Extracted information includes user

name, ranking, total commits, and the list of contributed projects for each developer, and for each project, the domain (e.g., OS, server), project size (in LOC), language (e.g., C, C++), license (e.g., gpl, lgpl21), and project rating (calculated by OHLOH).

First an Implicit Network is created among the projects. In this network, two projects have a relation edge if both are contributed to by the same developer. An edge weight reflects the total number of developers who have contributed to both the projects. A partial snapshot of this network is shown in Figure 4.11, in which, for instance, projects Debian and x.Org have a relation edge with weight 17, denoting the 17 developers who have contributed to both projects.

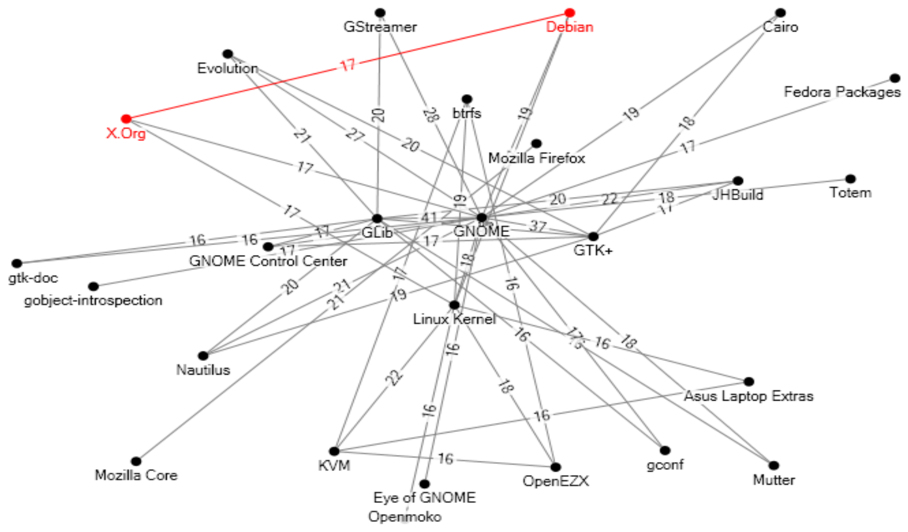


Figure 4.11: Partial snapshot of the Implicit Network

Then, for the projects present in each relation of the network, the resembling properties are identified. For instance, in the case of Debian and x.Org projects (in Figure 4.11) resemblance on the two properties are identified, including for instance, language (e.g., C), and domain (e.g., OS).

Result(s):
The first observation worth noticing here is that OSS developers often prefer to participate in projects that are affluent in terms of size, domain, and rating. For instance, high density of developers is noticed in the implicit network for projects that are very large (500KLOC) or large (50K-500KLOC) in size. Mutual developer participation count for a given pair of projects (that are very large or large) lies between 10 and 41. Additionally, this structure of sharing developers among multiple projects is analogous to the *small-world phenomenon* [87]. In a small-world structure several projects are connected with each other through one or more links, e.g., common developers. In this setup, with increasing number of common developers, the communities of related projects (as realized by the implicit network) become strongly interconnected.

The second observation is that the contributors often participate in projects that belong to the same domain, that utilizes same programming language(s) as development medium, and are of similar size. That is, projects that have high edge weight count (i.e., high developer participation) in the implicit network, are often exhibit resemblance in accordance to project domain, size and languages. For instance, around 98% of the projects that fall within the same domain or are the sub-projects of a larger one, attract large number of common developers. Similar observation holds for programming language and project size. For the former, high developer density was noticed for 87.03% resembling projects, whereas in case of the latter, very large and large size projects (64.13%) enjoy similar number of common contributors. Further analytical details are presented in the publication [VI].

Based on these observations following can be affirmed: if projects in an implicit network are connected with high edge weight count (e.g., ≥ 10) would most likely belongs to same domain, size and have similar technological platform. Likewise, having such resembling properties in common, projects should expect high degree of mutual developers participation.

4.7 Publication [VII]: A Framework for automated data analysis and visualization.

Introduction:
Comprehension of OSS projects is traditionally driven by the plethora of data produced and maintained by the projects. This data encapsulates the tacit knowledge on the evolution of the software, and provides the history of communication of the community. Acquisition and analysis of such data has been mostly manual or semiautomated and error-prone, mainly due to heterogeneous and unstructured data representation. This in turn increases the validity threat of the reported results and makes it incomparable across the studies. This paper proposes a framework to fully automate the analysis and visualization of OSS evolution data to tackle such challenges.

Objective:
The contribution of this paper is three-fold.

First, the formal definition of the three dimensions of OSS projects (social, technical and socio-technical) are given through the categorical analysis and discussion on the related research.

Second, a generic three-layer framework is proposed to fully automate the data driven analysis of OSS projects. This framework is inspired by the generations of the OSS quality models to achieve automation, as presented in Section 2.2.10. In addition, the framework should address the following challenges: data independence, re-usability, and inter-comparability of research results.

Third, a tool *Pomaz* is implemented that demonstrates how this approach produce generalized analytical results for social, technical, and socio-technical study of OSS projects.

Method:
In order to address the first objective, a literature review on the topic is carried

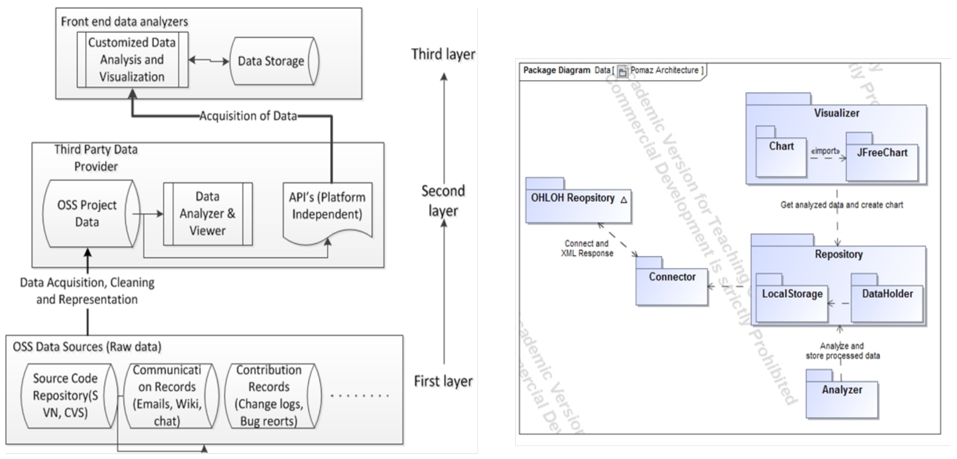
out. For this research approach presented in Section 3.2.1 is adopted in a limited capacity.

In proposing the framework, the highly generic and standard representation of OSS data provided by the third party data provider services are integrated with OSS project data sources and the data analysis logic.

Finally, a conceptual architecture encompassing the proposals of the framework is derived and implemented with a tool. The tool is further evaluated in answering research questions representative of each of the three dimensions of an OSS project.

Result(s):
Based on the review outcome, the three dimensions of OSS projects are defined, documentation of which is made in Section 2.2.

The proposed framework is presented in Figure 4.12(a). The first layer of this framework consists of the raw data sources offered by the hosting sites. For instance, most OSS projects provide public interfaces for each of the data sources to be accessed and downloaded. Nonetheless, other hosting facilities, such as Sourceforge and GitHub, provide similar facilities for the projects hosted by them.


Front end data analyzers
Customized Data Analysis and Visualization
Data Storage
Acquisition of Data
Third Party Data Provider
OSS Project Data
Data Analyzer & Viewer
API's (Platform Independent)
Data Acquisition, Cleaning and Representation
OSS Data Sources (Raw data)
Source Code Repository(SVN, CVS)
Communication Records (Emails, Wiki, chat)
Contribution Records (Change logs, Bug reorts)
Third layer
Second layer
First layer
Package Diagram Data [Pomaz Architecture]
Visualizer
Chart
«import»
JFreeChart
Get analyzed data and create chart
OHLOH Reopsitory
Connect and XML Response
Connector
Repository
LocalStorage
DataHolder
Analyze and store processed data
Analyzer
Academic Version for Teaching, Commercial Development is strictly Prohibited

(a) The Framework (b) The Architecture

Figure 4.12: The proposed framework and the architecture

The second layer constitutes the third party data providers. A representative example would be the OHLOH repository. For this layer, three essential components have been proposed. The first component is the data storage. The purpose of this storage is to perform data acquisition, essentially from all available sources provided by the first layer. Additionally, it will perform adequate data cleaning, and offer categorical representation of the data with unified and standard format. The second component is the data analyzer and viewer module would provide high level data analysis and visualization of selected projects. For instance, visualizing project activity for a selected period. The final component is a platform independent interface that comprises a set of API's to access the data. For instance, API's can be build to query the database, which will return the resultant data set in a platform independent format, such as XML, JSON. Realization of the services would offer the data in a unified and platform independent format.

The third layer in the framework is the front end data analyzers. This layer consists of a set of customized data analysis and visualization techniques and tools, and a local data storage unit. The techniques will be defined and implemented based on the need of different parties, e.g., research targeting to understand different perspectives of OSS projects. This layer makes use of the data provided by the second layer through the implementation of the standard APIs. This standardization of data removes the burden of semi-automated data gathering and preprocessing activities, and facilitates researchers and practitioners to concentrate more on the soundness of the underlying data analysis methodologies. Also, the data storage unit could cache the data collected through the API's as well as store the processed one for further use.

Within this framework, the second layer implements the data independence for the tools and methods built in third layer, whereas the data storage in the third layer ensures the re-usability of data.

The proposed architectural design for the framework is presented in Figure 4.12(b). The data provider in this reference architecture is the OHLOH repository, which corresponds to the second layer of the framework. The other modules correspond to the third layer of the framework. The Connector module is responsible for the low-level connection and data collection. This module also handles problems concerning missing data, false data or unavailability of the data provider repository. The Repository module implements the re-usability of the data. This two modules under the Repository module are used to store already collected and processed data in local disk, thus minimizing expensive server communication for repeated use of the same data. The analyzer accepts the user query, identifies and collects the required data from Repository, and carries out necessary computing. The resulting information is represented in a format in which the chart can show it, and it is also stored in LocalStorage. The Visualizer module handles all GUI operations (such as events) and user interactions in the application. The Chart module is responsible for these services which uses the JFreeChart[1] OSS library for visualization.

Finally, the tool ***Pomaz*** is implemented on top of the proposed architecture presented in Figure 4.12(b). Using the tool, it is possible to monitor the evolution of OSS projects from social, technical, and socio-technical perspectives, and set forecasts up regarding their future development. A demonstration of such analysis is presented in publication [VII] using the data extracted from FFmpeg and GStreamer projects. For this study, this tool is used to answer queries like, (a) How does the community affect the software?, (b) How does the community changes with the project evolution?, and (c) How does documentation follow the growth of software?.

[1]http://www.jfree.org/jfreechart/

4.8 Publication [VIII]: Recommendations to build graph based data analysis and visualization tool.

Introduction:
This paper offers a detailed discussion on the requirements to model and implement a *graph based* data analysis and visualization platform to address challenges, such as extensibility of the design and the tool, platform independent data-representation, and the inadequacy with graph visualization.

Objective:
Graph based data representation and visualization are most appropriate when there exist inherent relations among data elements. In such visualization, one can generate any number of links (i.e., edges) between two data points (i.e., nodes), and traverse easily a given path through the data. This visual experience can be further enhanced by using layout algorithms, navigation and interaction methods, and incremental exploration mechanisms.

However, such benefits can be easily undermined if the design and development does not address the inherent shortcomings of graph visualization methods and available technologies. This includes, (a) difficulties in visualizing and comprehending large graphs; (b) efficiency of a graph layout algorithm may be scale up-to several hundred nodes, but not beyond that; and (c) the time complexity for visualization, interaction and update of a graph is relatively high and increases with the increase in graph size.

This paper demonstrates an approach that effectively remedied the described deficiencies, and pretended recommendations and guidelines for future development.

Method:
Methodologically, this paper follows a constructive research approach in which an architectural design is first presented with a tool implementation, and then recommendations are advocated based on the experience gained during the process.

Result(s):
The design decisions in defining an architecture for a tool of this kind must provide a minimalistic kernel and well defined extension point(s) to built functionalities over the kernel. As OSS analysis tools of this kind operate on project data, a good practice would be to model a repository with generalized data representation for both the project and processed data. This repository with storing and retrieving functionality would form the system kernel and will provide interfaces to implement data analysis and visualization functionalities. Figure 4.13 shows the propsed architecture for Binoculars tool that accumulates these proposals. As shown in Figure 4.13, the module Repository forms the kernel for Binoculars. This repository holds the data extracted from OSS projects (Project Repository module) and processed data (Graph Repository module). XML is used to standardize the data representation for both the repositories.

Having this repository as the system kernel, it is now straightforward to build functionalities over it. As shown in the Figure 4.13, six modules are build on top of the kernel, each of which is intended to perform a specific task. For

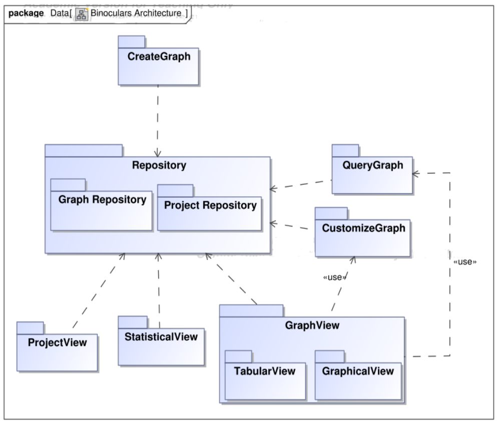
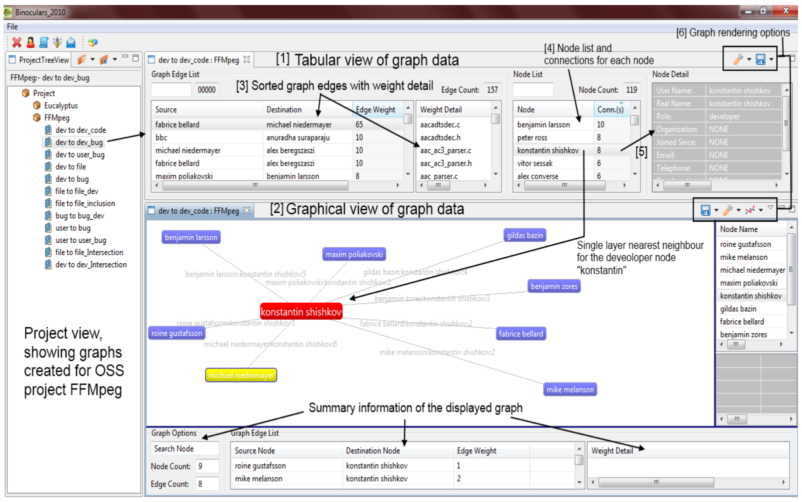


Figure 4.13: The architecture of the tool *Binoculars*

instance, CreateGraph module fetches project data from Project Repository, creates graphs based on user selected criteria, and stores them in Graph Repository. The GraphView module is a UI component responsible for drawing the graphs in the user interface and for handling user interactions. This architectural design along with XML data representation will archive the first two design challenges, i.e., the easy extensibility of the tool functionality and a platform independent data-representation.

The user interface of the tool Binoculars is presented in the Figure 4.14.


Binoculars_2010
File
ProjectTreeView
dev to dev_code : FFMpeg
[1] Tabular view of graph data
[2] Graphical view of graph data
[3] Sorted graph edges with weight detail
[4] Node list and connections for each node
[5]
[6] Graph rendering options
Single layer nearest neighbour for the deveoloper node "konstantin"
Project view, showing graphs created for OSS project FFMpeg
Summary information of the displayed graph
benjamin larsson
maxim poliakovski
gildas bazin
benjamin zores
konstantin shishkov
roine gustafsson
fabrice bellard
michael niedermayer
mike melanson
benjamin larsson:konstantin shishkov:5
maxim poliakovski:konstantin shishkov:2
gildas bazin:konstantin shishkov:4
benjamin zores:konstantin shishkov:3
roine gustafsson:konstantin shishkov:1
fabrice bellard:konstantin shishkov:2
michael niedermayer:konstantin shishkov:6
mike melanson:konstantin shishkov:2

FFMpeg:- dev to dev_bug

- Project
  - Eucalyptus
  - FFMpeg
    - dev to dev_code
    - dev to dev_bug
    - dev to user_bug
    - dev to file
    - dev to bug
    - file to file_dev
    - file to file_inclusion
    - bug to bug_dev
    - user to bug
    - user to user_bug
    - file to file_Intersection
    - dev to dev_Intersection

Graph Edge List 00000 Edge Count: 157

Source	Destination	Edge Weight
fabrice bellard	michael niedermayer	65
bbc	anuradha suraparaju	10
michael niedermayer	alex beregszaszi	10
fabrice bellard	alex beregszaszi	10
maxim poliakovski	benjamin larsson	8

Weight Detail
aacadtsdec.c
aacadtsdec.h
aac_ac3_parser.c
aac_ac3_parser.h
aac_parser.c

Node List Node Count: 119

Node	Conn.(s)
benjamin larsson	10
peter ross	8
konstantin shishkov	8
vitor sessak	6
alex converse	6

Node Detail

User Name:	konstantin shishkov
Real Name:	konstantin shishkov
Role:	developer
Organization:	NONE
Joined Since:	NONE
Email:	NONE
Telephone:	NONE

Node Name
roine gustafsson
mike melanson
michael niedermayer
maxim poliakovski
konstantin shishkov
gildas bazin
fabrice bellard
benjamin zores

Graph Options

Search Node
Node Count: 9
Edge Count: 8

Graph Edge List

Source Node	Destination Node	Edge Weight
roine gustafsson	konstantin shishkov	1
mike melanson	konstantin shishkov	2

Weight Detail

Figure 4.14: User Interface of the tool *Binoculars*

To cope with the third challenge, that is the performance inadequacy with graph visualization, this tool adopts the following approach, (a) provide appropriate level of data abstraction with incremental exploration which increase the efficiency of layout algorithms, and (b) use efficient algorithms. The exact instantiation of this approach is discussed below, with Figure 4.14 in mind.

A tabular view is much more efficient than graphs in displaying and rendering large data. Thus, Binoculars uses a tabular view for visualizing the entire graph information, that might consist (a) Graph with nodes and (weighted) edges; (b) Node list with degree count for each node; (c) description of each node; (d) Summary data on the graph; and (e) Options to render a graph. For instance, in Figure 4.14, items 3,4,5, and 6 show these implementations. Then, depending on the user query and customization options selected, an abstracted version of the graph (in tabular view) can be viewed in the graph view with associated details (for instance, item 2 in Figure 4.14). This design decision would substantially improve the user experience with the UI.

Chapter 5

Synthesis

This chapter derives the synthesis of the reported results by drawing them together in relation to the research questions and the included publications. This chapter is organized as follows. In Section 5.1, the synopsis of the reported results per research question is presented. The proposals on the socio-technical dependency for OSS projects is presented in Section 5.2. Section 5.3 discusses the overall implications of this research from both research and practitioners point of view. The main limitations to the validity of the results are discussed in Section 5.4. Finally, Section 5.5 draws the concluding remarks.

5.1 Research Questions Revisited

This section summaries the contributions of this thesis per research question.

RQ1 What conceptualization can be gained from the existing body of knowledge in relation to social, technical and socio-technical dimensions of the OSS projects?

The first motive and thus the contribution of this thesis is to build a synthesis of the existing research through an SLR. The outcome of this survey is twofold: first, it offers a solid basis for undertaking the research reported in this thesis, and second, it gives a categorical listing of the gaps and opportunities for future research on the topic.

In case of the earlier, it has been noticed that socio-technical aspect of OSS projects as a whole got minimal attention in research as compared to their study in isolation (discussed in Section 4.1). However, related studies on the topic have reported that efficacious socio-technical alignment is of great importance to software development project due to several implications. For instance, high degree of socio-technical dependency within a project leads to faster completion of modification requests with higher build success and product quality. In contrast, socio-technical gaps resulted in lower productivity with increase number of code changes and negative performance level within the organizations. Studying socio-technical dependency within the evolution of OSS projects would be a pragmatic approach in explaining several unsolved issues, such as OSS sustainability, quality, and productivity. Therefore, the study of socio-technical dependency within the context of OSS projects have been identified as a potential research area to be explored.

In case of the latter, a comprehensive set of future research based on the current progress on the topic has been documented. Reported research directions are broadly categorized on the three main dimensions of OSS projects, e.g., social, technical and socio-technical. Moreover, critical issues related to the research approach are presented. A complete reporting on the future research direction can be found in Section 5.3.1.

Nonetheless, perceived knowledge on the three dimensions have been utilized to unify their definition, and interpretation in relation to the scope of this thesis. These are documented in Section 2.2.

RQ2 How Socio-Technical dependency can be conceived in the OSS projects?

In comprehending socio-technical dependency, the first step would be to offer approach suitable to measure and explore the same within the context of OSS projects.

Accumulating the proposals of this thesis, the following approach can be offered to measure and comprehend socio-technical dependency from OSS project artefacts.

Step1 : Define an OSS data model to mimic the social and technical dimensions of OSS projects. This model should also support exploration of the interrelationships between the two dimensions.

Step2 : Define methods to dig deep into the three dimensions of the projects.

Step3 : Define a mathematical model to measure socio-technical dependency.

Step4 : Define a framework to achieve automated data driven analysis.

Step1: The OSS data model, as presented in Figure 4.1 and discussed in Section 4.2, effectively models the two dimensions of OSS projects by instantiating the essential repositories and communication channels. It also supports exploring inter-relations between the two dimensions in terms of relationship graphs.

Use of this model has several benefits as the model itself holds the basic building blocks of an OSS project and offers means to explore their relation. From architectural design perspective, this model could offer the kernel or the core of an analytical tool, and define extension points to build functionalities over it as per the requirements. This design principle is based on the Plug-in architectural design [47], which is often used by OSS projects to support rapid development in a distributed setup and to build a sustainable ecosystem [47]. In RQ4, this issue is discussed in detail.

This model could also offer the opportunity to achieve high content validity through a standard and platform independent data representation, and reusability. With content validity it would be possible to produce research results that are conformable, and comparable [85] to the research community.

Step2: A set of six methods are defined to explore interdependency and relation among the data sets presented in the model. These methods are based on the graph theoretic concepts, e.g., intra-connect, inter-connect, intersection, union, difference and exclusive-or. Each of these methods offer generic interface, which for instance take two data sets and derive a new relation for a given relation criteria. These methods can be used to analyze and produce data with newer insight for each of the three dimensions of the projects. In Section 4.2, three of these methods are presented with example cases. Other pragmatic scenarios in which these methods could be used includes, deriving the explicit and implicit architectures and coordination networks (see Section 2.2.6), deriving the contribution patterns and expertise of the community members, communication structure of the developer community, and cross-section of the community that have similar interest.

Step3: The mathematical model as explained in Section 2.2.6 measures the socio-technical dependency quantitatively. In doing so, this model accepts two architectures or coordination networks (both implicit and explicit), and outputs the congruence between the two as a percentile value with respect to the reference architecture or network. For deriving the architectures and the networks, inter-connect, intra-connect and intersection methods are used.

Step4: Finally, the proposed framework (see Section 4.7) can be used to achieve autonomy on data analysis. The target of this framework is to bring the concerned services under a common platform to offer automated and cohesive data analysis facilities. This framework integrates the OSS project data sources to that of the third party data provider services, which in turn can be hooked up with analysis methods and tools intended to serve specific requirements.

Now, a seamless integration of the proposed OSS data model and the methods to this framework can be done. This integration could offer a fully automated analysis service that address all the concerns presented in Section 1.2. Depending on the need, this integration can be done either at the second layer or at the third layer of the framework. An example integration is presented in Figure 5.1, in which the proposed model and methods are implemented in the third layer.

In this figure, the OSS data model would hold and present the data extracted from the third party service providers. Then the methods and the mathematical model could be implemented to operate on this data.

Additionally, each empirical study contributing to this thesis offers a detailed, coherent, and systematic methodological approach in conducting empirical and constructive investigation with OSS project data. This research approach accumulates the best practices and recommendations encountered during the SLR and during the process of this thesis. For each study, the details of the study protocol, study projects, data sources, data collection, refining and presentation procedure, data analysis and interpretation approach are documented along with the associated validity threats. In cases where applicable, research replication packages are also offered [IV]. The overall research methodology is presented in details in Chapter 3 with links to the included publications for case specific adoption.

RQ3 To what extent socio-technical dependency holds for the OSS projects?

The results obtained from empirical investigation of socio-technical dependency within the context of OSS projects are summarized below.

At intra-project level, socio-technical congruence is measured for the three

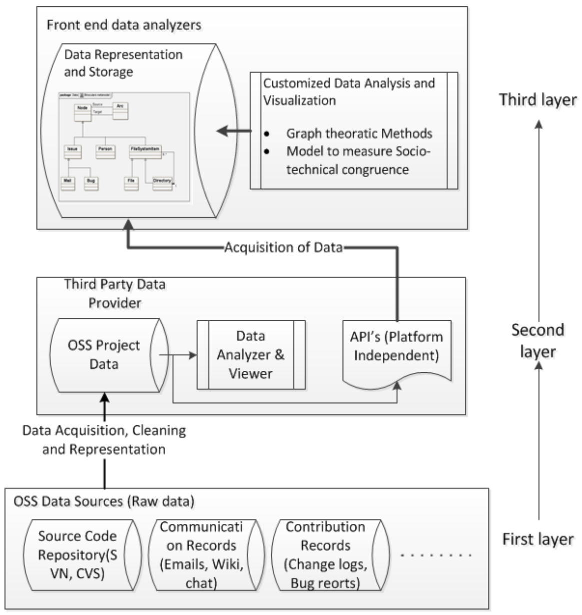

Front end data analyzers
Data Representation and Storage
Customized Data Analysis and Visualization
• Graph theoratic Methods
• Model to measure Socio-technical congruence
Third layer
Acquisition of Data
Third Party Data Provider
OSS Project Data
Data Analyzer & Viewer
API's (Platform Independent)
Second layer
Data Acquisition, Cleaning and Representation
OSS Data Sources (Raw data)
Source Code Repository(SVN, CVS)
Communication Records (Emails, Wiki, chat)
Contribution Records (Change logs, Bug reorts)
First layer

Figure 5.1: An integration of the model and the methods to the Framework

BSD forked projects. Reported results reveal significantly high (over 80% in almost all cases) and consistent leve of congruence during the entire period of the projects' evolution. This implies that to a considerable extent the communication of the contributing developers in the community may actually be due to the coordination needs as identified by the architectural dependencies, and vice-versa.

In an effort to extend this observation to inter-project level, socio-technical congruence was examined within the Ruby Gems OSS ecosystem, in which the technical domain constitutes the Ruby gems (i.e., projects) and their interrelation. However, results reveal relatively contrasting picture, with clues to reason such observation.

Overall the results show that in Ruby Gems ecosystem the congruence measure decreases with the increase in developers' participation in those projects. That is, projects that have large number of common contributors constitute rather small part of the ecosystem. However, a closer scrutiny to these group of projects reveals that these projects either belong to the same gem or are sub-projects to a larger gem or have development level dependency. Therefore, people participating in these projects might require consistent collaboration and coordination. On the contrary, for other projects in the ecosystem, the dependency exists only at runtime or at development time. Such dependencies do not necessarily impose any coordination needs among the projects, which might be the root cause of such low congruence measure.

Study on Inter-project resemblance due to common developer participation reveals that developers are keen to participate and contribute in projects that exhibit the following resembling properties: similar domain, similar technical platform (e.g., programming language), and similar project size. This investigation first starts with BSD projects which exhibit forked relation and resemble each other on the above three properties. Referring to discussion in Section 4.4, it can be affirmed that a significant portion of the community from each forked project contributes to the other. This participation count in general increases as the project matures, and remains stable within the given range.

Intuitively this observation might seem obvious, as forked projects resemble each other with respect to various important properties, and this should stimulate developer participation on those projects. Therefore, to absorb this observation as a pattern, further investigation is required. In order to do this, resembling projects are identified from an arbitrary sample of OSS projects that are contributed by a certain set of developers. The results reveal that for each resembling factor listed above, developer participation is significantly high among the resembling projects. For instance, around 98% of the projects that fall within the same domain or are the sub-projects of a larger one, attract large number of common developers (between 10 and 44). Similar observation holds for programming language and project size.

This observation for anonymous collection of projects combining with the results obtained for forked projects lead to affirm the following collaboration model, Resembling properties (e.g., domain, language, and size) of OSS projects often from a favorable ground for developers to participate in those projects. This observation holds in reverse as well. That is, two projects having a significant number of shared developer community most likely resemble on the given project properties.

RQ4 What architectural design and tool concept can be offered to conceptualize the socio-technical dependency in the OSS projects?

This thesis offers a set of architectural design and corresponding tool implementations that are based the proposed models, methods, and the framework. Table 5.1 lists the tools, their underlying design principle, and the methods they implement.

Table 5.1: Design principle and methods used for the tools

Tool Name	**Design Principle**	**Models and Methods**	**Autonomy**	**Requirement Achieved**
Binoculars	Plug-in Architectural design	- OSS data model. - Graph based methods. - Mathematical model for socio-technical congruence.	Semi-Automated	- Unified data representation. - Extensibility. - Reusability. - Automation.
Pomaz		- Framework. - Graph based methods.	Automated.	

As the architecture of both tools are based on the Plug-in design principle, a background note on the topic would be suitable for discussion. According to literature [47], a Plug-in architecture should comprise three functional building blocks, the core, the plug-in manager and the plug-ins. Figure 5.2 shows this conceptual model of a plug-in based system.

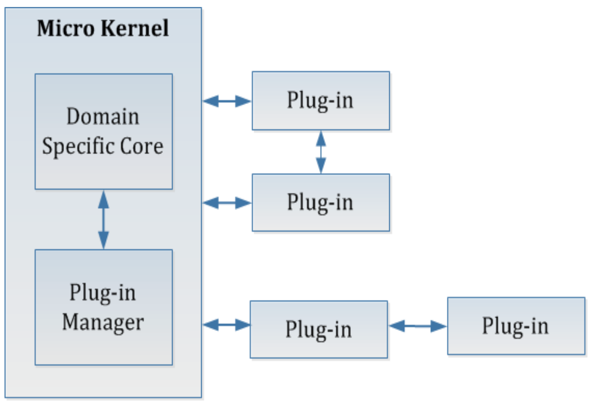


Figure 5.2: The conceptual model of a Plug-in architecture

In brief, the core along with the plug-in manager (often termed as micro-kernel) provides the fundamental services upon which real functionalities can be built. The services are subjective to the intended purpose of the system. However, generally they provide services like, handling the startup, registering and unregistering the plug-ins as required, data services, and most importantly, offer well defined extension points (or interface protocols) to deploy plug-ins. On the other hand, the plug-ins offer the real implementation of the functionalities and are hooked to the core by adhering to interface protocols.

Benefits of such design includes rapid and distributed development of the system functionalities and unit testing, decomposition of a complex system so that only required plug-ins are loaded at a given time, and building an ecosystem around the system.

To summarize, the Binoculars architecture adheres to this plug-in design principle. As discussed in Section 4.8, the OSS data model along with data access interfaces forms the micro-kernel of the system. Data within the model are represented with XML format. The data analysis and visualization techniques are then implemented as plug-ins to this kernel. Each of these plug-ins are plugged into the kernel through the implementation of the interfaces. Gained experience is formulated as guidelines to built analytical tools of this kind with graph based visualization approach, a discussion on which can be found in Section 4.8.

However, Binoculars is a semi-automated tool, in which manual checking of the extracted data is required to ensure its correctness. To tackle the challenge, the tool Pomaz is implemented. This tool offers a fully automated data analysis among others. As presented in Section 4.7, Pomaz architecture is built in accordance to the proposed framework. Within this architecture (Figure 4.12(b)), the OHLOH data repository is used as the data source. Thus, this repository constitutes the second layer of the framework that offer ready to use data in a platform independent format. The tool itself offers the implementation for the third layer. Its architecture conceptually resembles the Plug-in design principles (see Figure 4.12(b)). The Repository module constitutes the kernel whereas the analysis and visualization modules are implemented as plug-ins to operate on the repository data.

5.2 Proposals on the Socio-Technical Dependency in OSS Projects

Putting all the observations and discussions together, the following proposals can be offered in relation to the proposition (see Section 1.2) inferred in this thesis. The argument here is that these observations would set the stage in deriving formal theories on Socio-Technical dependency in OSS projects.

High degree of socio-technical congruence can be considered as the implicit underlying principle for building team collaboration and coordination within the developer community of long lived OSS projects. Even being highly distributed community of developers, and mostly using passive communication channels, OSS communities are tied together by maintaining task dependent communication. Such communication is often ad-hoc, adaptive and situated as it cope with rapid and continuous changes in the underlying software.

The inter-project collaboration model is significantly influenced by the resembling properties among the projects. Resembling properties (e.g., project domain, size, and programming language) often form a favorable ground, thus creating a stimuli for developers to participate in those projects. This observation holds in reverse as well, where two projects enjoying high density of common developer participation are most likely resemble on the given project properties.

5.3 Implications

This thesis offers a holistic view and understating on the contemporary phenomenon the *Socio-Technical Dependency* within the context of OSS projects. On one hand, this study proposes novel models, methods, framework, and tool support to assess socio-technical dependency, while on the other hand, empirical investigations are carried out through the mobilization of the proposals for a wide spectrum of OSS project setup.

This thesis offers the following connotations in relation to the reported results.

5.3.1 Implication for Research

One of the definitive implications of undertaking an SLR [I] is to envisage future research directions on the concerned topic. This is done through the aggregation and meta-analysis of the collected primary studies through the adoption of a review protocol. Thus, such research directions are contemporary and rational as they are elicited from, and backed by, the state-of-the-art research. The SLR carried out to address RQ.1 offers such opportunity to forecast research agendas along the concerned dimensions (e.g., technical, social and socio-technical). Recall that one of the motive of RQ.1 is to formulate future research agendas on the concerned domains along with plotting ground for this thesis. These agendas are summarized below.

OSS evolution research mostly focused on analyzing and understanding the evolution of the software [I]. Research under this dimension has produced good sample of analytical results which are available for further examination, assessment and comparison [83], accumulation of which suggest following potential research agendas.

On the Law's of OSS evolution: The most common study agenda is the fitness measure of Lehman's law for OSS evolution. However, a closer scrutiny to the reported results [I] reveal both facsimile and contradicting to this law. For instance, the growth rate of OSS systems vary between super-linear (i.e., greater than linear) and sub-linear (i.e., less than linear). This has both conformance and contradiction with the second and sixth law of evolution [58].

Comprehension of such results suggest that the laws and theory appear to be breaking down through nonconforming data and findings [I]. Thus, Lehman's laws of software evolution, which is primarily based on the study of the large close source systems, are not sufficient to justify or account for the evolutionary patterns and behavior of OSS. Nonetheless, these laws did not consider the community dimension of OSS projects, which is an essential force for the sustainable evolution of the OSS.

Thus, a promising research agenda would be to examine the underlying ontologies for software evolution [83] considering the OSS specific characteristics, and then re-assess the laws of software evolution to fit for OSS.

On the metric set for software evolution: Software evolution studies mostly utilize metrics that are empirically validated in prior studies [I]. These metrics are derived for closed source projects, and are primarily used to verify the Lehman's law of software evolution. Though these metrics provide valuable insight to OSS evolution, they do not consider OSS specific properties, such as development practices, organizational structure, product and project specific characteristics, and most importantly, the community dynamics. Thus an empirically validated set of metrics in favor of explicit representation of these aspects of the projects are required to complement the existing one.

Study on the community evolution identifies several key properties [I], which lay the foundation for further research in this direction, a glim of which is portrayed below.

On the community building: Studies reported that the majority of OSS projects failed to attract members to attain the critical mass. Only few flagship projects are able to attract developers. Factors influencing the motivation to join a community have been studied, and several phenomena are proposed. Yet, it is not identified what exclusive properties initiate the community building process at the nebula stage of the project. Thus following research tracks can be considered relevant:

- Why some projects are able to attract contributors during the nebula stage of the project, while most of them can not?

- What formation of the community refers to a balance one, and how the community structure changes towards a balance structure during its evolution?

- Can a visible pattern be identified within the domain of OSS projects for the above two cases?

On the migration of responsibility and sustainability: It has been reported that migration of developers from one release to the next is high and that the developers take more responsibility as they gain experience. Yet it is a common phenomenon in open source domain that developers freely join or leave the project. And when a developer leaves, his responsibilities must be handed over for sustainability of the project. The following topics should be relevant research problems in relation to this:

- How responsibility migrates among the developers? Does this migration follow preferential-attachment?, i.e., is the responsibility handed over to the developers who are in close connection to the outgoing developers.
- What impact such migration has on the project evolution?

In contrast to the technical and social evolution research, the socio-technical dimension is least entertained till the reporting of the review. This inadequacy in current research, yet profound impact of high socio-technical congruence in software projects lead this thesis to be conducted. However, other potential research directions includes for instance the following.

Sub-project evolution with their community: Large OSS projects often encompass many sub-projects. Often ecology of sub-communities formed around these sub-projects, and a centralized governance govern the communities [96]. Study on the formation and evolution of sub-projects and their communities have revealed many key characteristics [I]. Yet, analyzing the co-evolution of the two, and its impact on the overall project remains untouched. The following agendas would be worth to investigate.

- Is there a correlation between the evolution (growth, complexity, change) of sub-projects and associated sub-communities? Does the community change with the change in the sub-project?
- How does a community form around a newly added sub-project?
- What attributes of a sub-project attract new developers to join?
- What happens to the sub-community when a subproject is deleted or merged to other sub-project?
- What dependencies lead to inter-project communication?
- What kind and level of communication and collaboration take place between sub-communities?
- Is there a correlation between project and sub-project evolution?

Additionally, there are certain research issues popped out of the research results reported in this thesis, which are hereby summarized,

On the recovery of the architecture from developers coordination: High level of Socio-Technical congruence proclaims that implicit architecture of a software has resemblance to its explicit counterpart. In other words, it is possible to derive the explicit architecture to a certain extent if the project exhibits high socio-technical congruence. This verification is carried out in [III]. However, in this study, the implicit architecture overestimates the corresponding explicit one for each stable release [III], with more links between packages than in the explicit one. The reason for such overestimation could be that the implicit architecture is derived from all possible communication records among the developers. Such records consist of general conversations that are well outside the development discussion. For instance, *FreeBSD-chat* archive collects only the email threads related to the general discussion. Inclusion of such mailing threads cause additional relations among developers in explicit communication network which are not due to development discussion. This in turn creates additional links among packages in the corresponding implicit architecture. Therefore, a study on deriving the implicit architecture from explicit task and development related mailing threads and its resemblance to that of the explicit one needs to be carried out.

On the Models and methods: The proposed mathematical model and the graph theoretic methods are used to measure socio-technical congruence for a number of OSS project setup. This measurement and the associated empirical investigations lead to offer proposals on the socio-technical dependency. However, statistical evaluation is required to justify the impact of this measures and proposals to that of the OSS success factors, e.g., productivity, maintainability, quality, and sustainability.

On the Socio-Technical Congruence in an ecosystem: Study of socio-technical dependency at ecosystem level needs to be fine tuned to achieve deeper insight and reasoning. One way to achieve this is to examine socio-technical congruence at different abstraction levels of an ecosystem, for instance, first at intra-project level (i.e., within each project of the ecosystem), then for a cluster of projects that are tightly connected (e.g., sub-projects of a larger one), and then for the whole ecosystem. The measurement should be done at discrete time intervals to derive trends and patterns. Initiating such study would provide means for validating the observations reported in this thesis.

On the Framework: The proposed three layer framework [VII] is an initial step towards a full fledged solution for automated analysis of OSS projects. With the rise of data services such as OHLOH, future analysis tools could be developed as plug-ins loaded online in third party service containers. This is analogous to social network sites like Facebook, which acts as an open container for custom applications in addition to providing API for hosted data. Such an approach can be taken to extend the framework in the future.

Finally, a couple of issues related to the research approach can be improved to increase the acceptability of the results reported in this thesis.

External validity: This is a major threat for any empirical research, and specially with OSS projects. Due to sheer number of OSS projects, it is practically impossible to gain comprehensive external validity of the documented results. However, results could be verified within a focused population of projects having similar properties. Even though this thesis covers results from 6 flagship OSS projects, one ecosystem and a public data repository, yet results can not be generalized for any class of projects. Thus, achieving external validity for the proposed models, methods, and empirical results could be a possible extension.

Framework for data collection and representation: As discussed throughout this thesis, OSS projects often produce large volume of data representing their development and evolution history. Research to date explores the repositories that maintain these data, a list of which is provided in [I]. However, data collection and representation in these repositories vary significantly from project to project. Furthermore, data from the same source may have different formatting (e.g., emails are often free of format even in listing the senders credentials). Due to these facts, it is a challenging task to collect relevant data following a standard format from OSS repositories. In this context, researchers often employ their own means to collect and represent data for research. This reduces the compatibility and comparability of the reported results even if they use same data sources. Taking these issues in consideration, a framework for uniform data collection and representation can be developed to make the results cohesive and comparable to each other.

In this thesis, the well accepted approach for data collection, refining, and analysis are adopted. Additionally, each step of this process is documented and uniformly used in each included paper. Replication package is offered to better support reproducibility whenever possible [IV].

5.3.2 Implication for Practice

Apart from having implications on further research on the track, reported study has significance from practitioners perspective. This includes the followings,

The SLR offered as part of answering the RQ.1 could be a hands-on guide for practitioners. It categorically presents empirically validated and accepted metrics, methods, datasets, and tools used for evaluating socio-technical aspects of an OSS project. This could be a reference point for the professionals who would like to built analytical tools and frameworks through the utilization of the best practiced approaches.

From the design perspective, the proposed data model and the corresponding architectural design presented as part of answering RQ.2 and RQ.4, respectively, advocate and apply the Plug-in architectural design principle. This essentially provides evidence that such modeling of OSS projects is a pragmatic approach for rapid building of functionalities as plug-ins. This should also support building an ecosystem of a distributed development community.

On the other hand, the framework presented in relation to RQ.2 and RQ.4, fosters the design considerations for achieving higher autonomy on data driven analysis of OSS projects. Additionally, adoption of this proposal would support the following enhancements: data independence, re-usability of the data, increase of the validity measure and comparability of the research results. Here, data independence ensures the fact that methods derived for OSS data analysis should be independent of OSS data, and can be applied across OSS projects. Re-usability should enforce logging of data to local repositories to minimize costly request over Internet for subsequent access. Additionally, a standard representation and access mechanism of data across OSS projects would produce cohesive research results, with greater validity, confidence, and comparability.

Additionally, analytical tools built on top of the proposed methods and models (offered as part of RQ.4) would provide a quick index to organizations to assess how well organizational activity is actually aligned with the current and planned sub-division of inter dependent tasks in the project. This periodical assessment would help improve software development processes and practices.

Apart from these, the outcomes of the empirical investigation carried out to answer RQ.3 have pragmatic implications as well, a description of which is presented below.

As discussed in Section 5.3.1, one of the implicit implications of having high degree of socio-technical congruence is that the implicit architecture could complement the traditional reverse engineering process in recovering the architecture of legacy systems. That is, given a comprehensive historical record of task dependent community communication and developers contribution to the code-base, the recovery of the system architecture can be achieved to a certain extent.

The OSS projects studied in this thesis are ideally implementing the observed model with high socio-technical congruence during their entire evolution history. Therefore, it could be argued that the alignment between the social and technical dimensions plays a pivotal role in forming cohesive and organized community driven projects, which eventually leads to their successful evolution with high quality.

Also, the collaboration model of developers who participate in multiple projects is analogous to the *richer gets rich* and the *small world* phenomenon. Such collaboration observed for OHLOH projects as well as for forked projects. For both cases, a high density of developer participation is observed for projects that exhibit resemblance in programming languages, domain, and project size. This collaboration might have impact on project success, as the productivity of the contributors is boosted by providing them a dense communication channel to acquire more quantity and variety of information and knowledge resources [94].

Finally, for forked projects, this collaboration model could be one of the properties to overrule the traditional perception of forking which is thought to have negative stimuli for sustainable project evolution as presented in Section 4.4.

5.4 Limitations and Threats to Validity

Research work in this thesis has been conducted with well established and practiced software engineering research methods and strategies, a detailed reporting of which is made in Chapter 3. Even though the validity of the research outcome can be questioned for several other factors that influence the research. In literature, the validity of a research designates the trustworthiness of the results. That is, to what extent the reported results are original and not biased by the researchers' subjective point of view [79].

Current literature presents different ways to classify aspects of the validity and threats of a scientific research. For this thesis, three perspectives of the validity and threats are considered [100] [79]. Table 5.2 summarizes these validity threats and state them in relation to the concerned publications and the mitigation measure carried out.

Table 5.2: An overview on the validity threats

Validity Threat	**Description**	**Concerned Publication**	**Mitigated**
External validity	How results can be generalized	[II][III][IV][V][VI][VII][VIII]	Partial
Internal validity	Confounding factors that can influence the findings	[I]	Mitigated
		[II][III][IV][V][VI][VII][VIII]	Partial
Construct validity	Relationship between theory and observation	[I]	Mitigated
		[II][III][IV][V][VI][VII][VIII]	Partial

In what follows, a brief description of each of the validity perspectives, and their mitigation process in relation to individual publication included in this thesis.

External validity

This perspective of validity identifies the extent to which (a) the reported results can be generalized, and (b) are of interest to the relevant people outside the investigated case [100] [79]. In assessing external validity, researchers often make effort to extend the findings to other relevant cases. However, there is no population from which a statistically representative sample has been drawn [79]. Yet the motive is to define a theory, that enables analytical generalization to the population having common characteristics and relevance.

Internal validity

This aspect of validity concerns how one or more factors affect the investigated factor [79]. If the research misses one such factor that might have impact on the investigated one and thus influence the overall observation, then there is a threat to internal validity.

Construct validity

Construct validity reflects to what extent the actual operational measures represents the intended / planned one for investigating the research questions [79]. Any deviation between the two raises a threat to the construct validity.

Most validity threats to an SLR (publication [I] in Table 5.2) is from internal and construct validity. Because carrying out a survey is mostly a manual task, it is a subject to human interpretation that might be biased [23]. To minimize this, guidelines suggested by [79] in conducting SLR is adopted. In particular, all steps in the review were documented and reviewed in advance, including the selection criteria and attribute definitions.

Case study research (publications [II][III][IV][V] in Table 5.2) and constructive research (publications [VI][VII][VIII] in Table 5.2) done in this thesis are subject to all the three validity threats discussed above.

For these studies, a total of six OSS projects, Ruby gems ecosystem and the OHLOH repository were used. These projects are large in size having over 20 years of development and evolution history, and the community size (both developer and user community) exceeds well over the critical mass [I]. These projects mostly belong to the domain of operating systems, multimedia and cloud computing. Additionally, OSS evolution studies often used these projects as case studies. Thus, it might be possible to stress the results reported in this thesis to the population having similar properties, e.g., domain, project size, technical platform, and evolution history. Yet, claiming complete external validity on the basis of this reasoning is out the scope of this thesis.

Furthermore, internal and construct validity for these studies are mostly due to (a) lack of statistical analysis validation for the proposed models and methods, (b) limitations in accuracy for programs developed for data collection and analysis, and (c) missing historical data, that limits the study to make use only of available data. However, missing data is observed only for NetBSD and OpenBSD projects, for which a total of one month email conversation record can not be retrieved due to missing links to the corresponding repository.

5.5 Conclusions

Fred Brooks, in his classic book *The Mythical Man Month* [10] provides an analogy on *Why did the (mythical) Tower of Babel Fail?*, even though the people had a clear mission, manpower, materials (raw), time, and required technology. The project failed because of communication, and its consequences on the organization [10]. In software systems, schedule disasters, functional misfits and system bugs arise from a lack of communication and coordination among group of people involved in the development process [10]. Coordination in software development must be ad-hoc, flexible and situated in order to (a) map the coordination needs devised by the underlying technical dependencies of the software, and (b) to effectively encounter uncertainties probed by rapid and continuous changes in the underlying software.

Successful OSS projects, being the utmost specimen of unconventional software development process, posed the query, do such projects exhibits high quality of coordination that could explain their sustainable evolution? and if so, how that could be conceived? This thesis is a sincere and humble effort to offer a deeper understanding on the topic through the lens of socio-technical dependency.

In doing so, this thesis first examines the state-of-the-art literature concerning the social, technical and socio-technical dimensions of OSS projects, and derive the need and scope for this work.

Second, this thesis proposes models, and methods to measure socio-technical dependency quantitatively from OSS project artifacts. This includes proposal of (a) a OSS data model, (b) a set of six graph theocratic methods, and (c) a mathematical model of socio-technical congruence. The data model mimics the social, and technical dimensions by representing the data taken from OSS project artifacts, and the methods reveals their interdependency in terms of architectures and coordination networks. The mathematical model then measures the socio-technical dependency quantitatively by unitizing the architectures and the networks.

Third, the thesis carries out empirical investigations for distinct OSS project context to build proposals, trends and patterns of socio-technical dependency within such projects. This is done by applying the proposed models and methods.

Finally, this thesis presenters a set of proposals to automate the analysis and visualization of the socio-technical dependency in OSS projects. This includes, proposal for (a) a three layer framework, (b) plug-in based architectural designs for building tools on top of the framework and the OSS data model, and (c) developing corresponding tool supports (e.g., Binoculars and Pomaz), and thereby distill lessons learned during the tool-building experience as a practitioners guide.

Future work along this track should offer statistical verification of the proposed methods and models in relation to the success factors of OSS projects, e.g., sustainability, maintainability, evolution, management. Additionally, extending the results presented in this thesis in deriving guidelines and models for building sustainable open source ecosystems would be a promising research direction.

Bibliography

[1] T. Aaltonen and J. Jokinen. Influence in the linux kernel community. *International Conference of Open Source Systems*, pages 203–208, 2007.

[2] A. Al-Ajlan. The evolution of open source software using eclipse metrics. In *International Conference on New Trends in Information and Service Science*, pages 211–218, 2009.

[3] D. Atkins, T. Ball, T. Graves, and A. Mockus. Using version control data to evaluate the impact of software tools. *Proceedings 21st International Conference on Software Engineering*, 24:324–333, 1999.

[4] C. Baldwin and K. Clark. Design rules: The power of modularity. *MIT Press*, 2000.

[5] S. G. R. G. Biehl J.T., Czerwinski M. Fastdash: A visual dashboard for fostering awareness in software teams. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1313–1322, 2007.

[6] C. Bird. Sociotechnical coordination and collaboration in open source software. In *International Conference on Software Maintenance*, pages 568–573, 2011.

[7] F. Bolici, J. Howison, and K. Crowston. Coordination without discussion? socio-technical congruence and stigmergy in free and open source software projects. *2nd STC, International Conference on Softwafre Engineering*, 2009.

[8] A. Bonaccorsi and C. Rossi. Altruistic individuals, selfish firms? the structure of motivation in open source software. *First Monday (1-5)*, pages 203–208, 2004.

[9] F. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[10] F. P. Brooks. The mythical man-month. *Anniversary Edition: Addison-Wesley Publishing Company*, 1995.

[11] T. Browning. Applying the design structure matrix to system decomposition and integration problems: a review and new directions. *IEEE Transactions on Engineering Management*, 48(3):292–306, 2011.

[12] R. Burton and B. Obel. Strategic organizational diagnosis and design. *Kluwer Academic Publishers*, 1998.

[13] A. Capiluppi, J. González-Barahona, I. Herraiz, and G. Robles. Adapting the "staged model for software evolution" to free/libre/open source software. In *International workshop on Principles of software evolution*, pages 79–82, 2007.

[14] A. Capiluppi, P. Lago, and M. Morisio. Evidences in the evolution of os projects through changelog analysis. In *Proceedings of the 3rd Workshop on Open Source Software Engineering (International Conference on Softwafre Engineering'03)*, pages 19–24, 2003.

[15] K. Carley and Y. Ren. Tradeoffs between performance and adaptability for c^{3}i architectures. *International Command and Control Research and Technology Symposium*, 2001.

[16] M. Cataldo, J. Herbsleb, and K. Carley. Socio-technical congruence: A framework for assessing the impact of technical and work dependencies on software development productivity. *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 2–11, 2008.

[17] M. Cataldo, P. Wagstrom, J. Herbsleb, and K. Carley. Identification of coordination requirements: implications for the design of collaboration and awareness tools. In *Computer supported cooperative work*, pages 353–362, 2006.

[18] M. Cataldo, P. Wagstrom, J. Herbsleb, and K. Carley. Identification of coordination requirements: Implications for the design of collaboration and awareness tools. *Conference on Computer supported cooperative work, Banff, Canada*, 2006.

[19] D. Challet and Y. L. Du. Microscopic model of software bug dynamics: Closed source versus open source. *International Journal of Reliability, Quality and Safety Engineering*, 12(6), 2005.

[20] R. Chang, S. Yang, J. Moon, W. Oh, and A. Pinsonneault. A social capital perspective of participant contribution in open source communities: The case of linux. In *Hawaii International Conference on System Sciences*, pages 1–10, 2011.

[21] M. Conway. How do committees invent? *Datamation*, 14(4):28–31, 1968.

[22] J. Cook, L. Votta, and A. Wolf. Cost-effective analysis of in-place software processes. *Transactions on Software Engineering*, 24:650–663, 1998.

[23] B. Cornelissen, A. Zaidman, A. Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *Transactions on Software Engineering*, 35(5):684–702, 2009.

[24] K. Crowston and J. Howison. The social structure of free and open source software development. *First Monday*, 10(2), 2005.

[25] R. Daft and K. Weick. Towards a model of organizations as interpretation systems. In *Academy of Management Review*, volume 9, pages 284–295, 1984.

[26] P. Dewan and R. Hegde. Semi-synchronous conflict detection and resolution in asynchronous software development. *European Conference on Computer Supported Cooperative Work*, pages 158–179, 2007.

[27] N. Ducheneaut. Socialization in an open source software community: A socio-technical analysis. *Computer Supported Cooperative Work*, 14(4):323–368, 2005.

[28] K. Ehrlich, M. Helander, G. Valetto, S. Davies, and C. Williams. An analysis of congruence gaps and their effect on distributed software development. *Proc. Socio-Technical Congruence Workshop at International Conference on Softwafre Engineering Conf.*, 2008.

[29] A. Fink. The survey handbook. In *Sage Publications*, 2003.

[30] J. Froehlich and P. Dourish. Unifying artifacts and activities in a visual tool for distributed software development teams. *International Conference on Softwafre Engineering*, pages 387–396, 2004.

[31] A. Garcia, P. Greenwood, G. Heineman, R. Walker, Y. Cai, H. Y. Yang, E. Baniassad, C. Lopes, C. Schwanninger, and J. Zhao. Assessment of contemporary modularization techniques. *ACM SIGSOFT Software Engineering Notes*, 35(5):31–37, 2007.

[32] D. M. German. Using software trails to reconstruct the evolution of software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16:367–384, 2004.

[33] E. Gilbert and K. Karahalios. Codesaw: A social visualization of distributed software development. *Human-Computer Interaction - INTERACT*, pages 303–316, 2007.

[34] P. Gloor. *Swarm Creativity*. Oxford University Press, 2006.

[35] R. Glott, A. Groven, K. Haaland, and A. Tannenberg. Quality models for free/libre open source software-towards the silver bullet? *EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 439–446, 2010.

[36] M. Godfrey and Q. Tu. Evolution in open source software: A case study. In *International Conference on Software Maintenance*, pages 131–142, 2000.

[37] M. Goeminne and T. Mens. A framework for analyzing and visualizing open source software ecosystems. In *International workshop on Principles of software evolution*, pages 42–47, 2010.

[38] J. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. Amor, and D. German. Macro-level software evolution: A case study of a large software compilation. *Journal Empirical Software Engineering*, 14(3):262–285, 2009.

[39] R. Grinter, J. Herbsleb, and D. Peffy. The geography of coordination: Dealing with distance in r&d work. *Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pages 306–315, 1999.

[40] J. Gutsche. The evolution of open source communities. *Topics in Economic Analysis and Policy*, 52(1):1000–1014, 2005.

[41] R. Henderson and K. Clark. Architectural innovation: the reconfiguration of existing product technologies and the failure of established firms. In *Administrative Science Quarterly*, volume 35, pages 9–30, 1990.

[42] J. Herbsleb. Global software engineering: The future of socio-technical coordination. *Future of Software Engineering*, pages 188–198, 2007.

[43] J. Herbsleb and R. Grinter. Architectures, coordination, and distance: Conway's law and beyond. *IEEE Software*, 16(5):63–70, 1999.

[44] I. Herraiz. A statistical examination of the evolution and properties of libre software. *International Conference on Software Maintenance*, pages 439–442, 2009.

[45] E. V. Hippel. Task partitioning: an innovation process variable. *Research Policy*, 19(5):407–418, 1990.

[46] Q. Hong, S. Kim, S. Cheung, and C.Bird. Understanding a developer social network and its evolution. In *27th International Conference on Software Maintenance*, pages 323–332, 2011.

[47] S. Jansen and G. van Capelleveen. Quality review and approval methods for extensions in software ecosystems. *Software Ecosystem*, pages 187–217, 2013.

[48] A. Jedlitschka and D. Pfahl. Reporting guidelines for controlled experiments in software engineering. In *Proceedings of ACM/IEEE International Symposium on Empirical Software Engineering*, pages 95–104, 2005.

[49] W. Jingwei, R. Holt, and A. Hassan. Empirical evidence for soc dynamics in software evolution. In *IEEE International Conference on Software Maintenance*, pages 244–254, 2007.

[50] E. Kasanen, K. Lukka, and A. Siitonen. The constructive approach in management accounting research. *Journal of Management Accounting Research*, 1(5):243–263, 1993.

[51] C. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *Transactions on Software Engineering*, 25(4):493–509, 1999.

[52] B. A. Kitchenham. Procedures for performing systematic reviews. In *Technical Report TR/SE-0401, Keele University, and Technical Report 0400011T.1, National ICT Australia*, 2004.

[53] B. A. Kitchenham, R. Pretorius, D. Budgen, O. P. Brereton, M. Turner, M. Niazi, and S. Linkman. Systematic literature reviews in software engineering- a tertiary study. *Information and Software Technology*, 52(8):792–805, 2010.

[54] H. Klein and M. Myers. A set of principles for conducting and evaluating interpretative field studies in information systems. *MIS Quarterly - Special issue on intensive research in information systems*, 23(1):67–88, 1999.

[55] S. Koch. Software evolution in open source projects - a large-scale investigation. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(6):361–382, 2007.

[56] I. Kwan, M. Cataldo, and D. Damian. Conway's law revisited: The evidence for a task-based perspective. *IEEE Software*, 29(1):90–93, 2012.

[57] I. Kwan, A. Schröter, and D. Damian. Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. *Transactions on Software Engineering*, 37(3):307–324, 2011.

[58] M. Lehman. Software evolution. *Encyclopedia of Software Engineering, 2nd Edition, John Wiley and Sons Inc.*, pages 1507–1513, 2002.

[59] W. Li and R. Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, 80(7):1120–1128, 2007.

[60] M. Longu, M. Lanza, T. Gibra, and R. Robbes. The small project observatory: visualizing software ecosystems. *Science of Computer Programming*, 75(4):264–275, 2010.

[61] K. Lukka. Konstruktiivinen tutkimusote: luonne, prosessi ja arviointi. *Rolin, K., Kakkuri-Knuutila, M.-L., Henttonen, E. (eds.) Soveltava Yhteiskuntatiede ja Filosofia, Gaudeamus, Helsinki*, pages 111–133, 2006.

[62] K. Manikas and K. M. Hansen. Software ecosystems: A systematic literature review. *Journal of Systems and Software*, 86(5):1294–1306, 2013.

[63] D. Messerschmitt and C. Szyperski. Software ecosystem: Understanding an indispensable technology and industry. *MIT press book 1, London, England.*, 2005.

[64] R. Milev, S. Muegge, and M. Weiss. Design evolution of an open source project using an improved modularity metric. In *International Conference of Open Source Systems'09*, pages 20–33, 2009.

[65] A. Mockus and J. Herbsleb. Expertise browser: A quantitative approach to identifying expertise. *International Conference on Softwafre Engineering*, pages 503–512, 2002.

[66] K. Nakakoji, K. Yamada, and E. Giaccardi. Understanding the nature of collaboration in open-source software development. In *Asia Pacific Software Engineering Conference*, pages 827–834, 2005.

[67] E. Nasseri and S. Counsell. System evolution at the attribute level: An empirical study of three java international conference of open source systems and their refactoring. In *International conference on Information Technology Interfaces*, pages 653–658, 2009.

[68] K. Ngamkajornwiwat, D. Zhang, A. Koru, L. Zhou, and R. Nolker. An exploratory study on the evolution of international conference of open source systems developer communities. In *Hawaii International Conference on System Sciences*, page 305, 2008.

[69] G. Olson and J. Olson. Distance matters. *Human Computer Interaction*, 15(2 and 3):139–178, 2007.

[70] H. Orsila, J. Geldenhuys, A. Ruokonen, and I. Hammouda. Update propagation practices in highly reusable open source components. In *International Conference of Open Source Systems*, pages 159–170, 2008.

[71] K. Piirainen and R. Gonzalez. Seeking constructive synergy: Design science and the constructive research approach. *Design Science at the Intersection of Physical and Virtual Design*, pages 59–72, 2013.

[72] G. Porruvecchio, S. Uras, and R. Quaresima. Social network analysis of communication in open source projects. In *Proceedings of 9th International Conference on Agile Processes in Software Engineering and Extreme Programming*, pages 220–221, jun 2008.

[73] R. G. R, G. Lilien, and G. Mallapragada. Location, location, location: How network embeddedness affects project success in open source systems. *Management Science*, 52(7):1043–1056, 2006.

[74] E. Raymond. The cathedral and the bazaar. *O'Reilly & Associates, Cambridge MA*, 1999.

[75] J. Roberts, I.-H. Hann, and S. Slaughter. Understanding the motivations, participation, and performance of open source software developers: A longitudinal study of the apache projects. *Management Science*, 52:984–999, 2006.

[76] G. Robles, J. Amor, J. Gonzalez-Barahona, and I. Herraiz. Evolution and growth in large libre software projects. In *International workshop on Principles of software evolution*, pages 165–174, 2005.

[77] G. Robles and J. Gonzalez-Barahona. A comprehensive study of software forks: Dates, reasons and outcomes. In *International Conference of Open Source Systems*, pages 1–14, 2012.

[78] C. Robson. Real world research. In *John Wiley & Sons, 3rd Edition*, 2011.

[79] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[80] R. Sangwan, P.Vercellone-Smith, and C. Neill. Use of a multidimensional approach to study the evolution of software complexity. *Journal of Innovations in Systems and Software Engineering*, 6(4):299–310, 2010.

[81] A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb. Tesseract: Interactive visual exploration of socio-technical relationships in software development. *International Conference on Softwafre Engineering, IEEE*, pages 23–33, 2009.

[82] A. Sarma, Z. Noroozi, and A. V. der Hoek. Palantr: Raising awareness among configuration management workspaces. *International Conference on Softwafre Engineering*, pages 444–454, 2003.

[83] W. Scacchi. Understanding open source software evolution: Applying, breaking, and rethinking the laws of software evolution. *Applying, Breaking, and Rethinking the laws of software evolution, John Wiley and Sons Inc.*, pages 1043–1056, 2003.

[84] S. Shah. Motivation, governance, and the viability of hybrid forms in open source software development. *Journal of Management Science*, 52(7):1000–1014, July 2006.

[85] A. Shenton. Strategies for ensuring trustworthiness in qualitative research projects. *Education for Information*, 22(2):63–75, 2004.

[86] M. Simmons, P. Vercellone-Smith, and P. Laplante. Understanding open source software through software archeology: The case of nethack. In *IEEE/NASA Software Engineering Workshop*, pages 47–58, 2006.

[87] P. V. singh. The small-world effect: The influence of macro-level properties of developer collaboration networks on open-source project success. *ACM Transactions on Software Engineering and Methodology*, 20(2), 2010.

[88] M. Sosa, S. Eppinger, and C. Rowles. The misalignment of product architecture and organizational structure in complex product development. *Management Science*, 12(50):1674–1689, 2004.

[89] C. Souza, S. Quirk, E. Trainer, and D. Redmiles. Supporting collaborative software development through the visualization of socio-technical dependencies. *ACM SIGGROUP Conference on Supporting Group Work*, pages 147–156, 2007.

[90] C. D. Souza, D. Redmiles, L.-T. Cheng, D. Millen, and J. Patterson. How a good software practice thwarts collaboration – the multiple roles of apis in software development. *12 th Conference on Foundations of Software Engineering*, pages 221–230, 2004.

[91] K. Stewart, D. Darcy, and S. Daniel. Observations on patterns of development in open source software projects. In *Proceedings of the fifth workshop on Open source software engineering*, pages 1–5, 2005.

[92] G. Valetto, M. Helander, K. Ehrlich, S. Chulani, M. Wegman, and C. Williams. Using software repositories to investigate socio-technical congruence in development projects. *International Workshop on Mining Software Repositories*, page 25, 2007.

[93] Y. Wang, D. Guo, and H. Shi. Measuring the evolution of open source software systems with their communities. *ACM SIGSOFT Software Engineering Notes*, 32(6), 2007.

[94] D. Watta. Networks, dynamics, and the small world phenomenon. *Amer. J. Sociology 105*, pages 493–527, 1999.

[95] M. Wattenberg, S. Rohall, D. Gruen, and B. Kerr. E-mail research: Targeting the enterprise. *Human-Computer Interaction*, 20:139–162, 2005.

[96] M. Weiss, G. Moroiu, and P. Zhao. Evolution of open source communities. *International Conference of Open Source Systems*, 203:21–32, 2006.

[97] H. Wen, R. D'Souza, Z. Saul, and V. Filkov. Evolution of apache open source software. *Modeling and Simulation in Science, Engineering and Technology, Springer*, pages 199–215, 2009.

[98] R. Wendel, J. Bruijn, and M. Eeten. Protecting the virtual commons, information technology & law series. In *T.M.C. Asser Press*, pages 44–50, 2003.

[99] R. Wieringa and H. Heerkens. Designing requirements engineering research. In *In Proceedings of the Fifth International Workshop on Comparative Evaluation in Requirements Engineering*, pages 36–48, 2007.

[100] C. Wohlin, P. Runeson, M. Höst, M. Ohlson, B. Regnell, and A. Wesslén. Experimentation in software engineering: An introduction. In *Kluwer Academic*, 2000.

[101] Y. Yamauchi, M. Yokozawa, T. Shinohara, and T. Ishida. Collaboration with lean media: How open-source software succeeds. In *Conference on Computer supported cooperative work*, pages 329–338, 2000.

[102] Y. Ye, K. Nakakoji, Y. Yamamoto, and K. Kishida. The co-evolution of systems and communities in free and open source software development. *Free/Open Source Software Development. IGI Global*, pages 59–83, 2005.

[103] R. Yin. Case study research: design and methods. *Sage Publications*, 2013.

[I] M.M. Syeed, I. Hammouda, and T. Systä. Evolution of Open Source Software Projects: A Systematic Literature Review. *Journal of Software, vol. 8, no. 11, 2013*, pages 2815–2829. November, 2013.

Evolution of Open Source Software Projects: A Systematic Literature Review

M.M. Mahbubul Syeed, Imed Hammouda, Tarja Systä
Department of Pervasive Computing, Tampere University of Technology, Tampere, Finland
Email: {mm.syeed, imed.hammouda, tarja.systa}@tut.fi

Abstract— **Open Source Software (OSS) is continuously gaining acceptance in commercial organizations. It is in this regard that those organizations strive for a better understanding of evolutionary aspects of OSS projects. The study of evolutionary patterns of OSS projects and communities has received substantial attention from the research community over the last decade. These efforts have resulted in an ample set of research results for which there is a need for up-to-date comprehensive overviews and literature surveys.**

This paper reports on a systematic literature survey aimed at the identification and structuring of research on evolution of OSS projects. In this review we systematically selected and reviewed 101 articles published in relevant venues. The study outcome provides insight in what constitutes the main contributions of the field, identifies gaps and opportunities, and distills several important future research directions.

Index Terms— **Open Source; Evolution; Systematic Literature Review.**

I. Introduction

Research on Open Source Software (OSS) has gained momentum over the last decade as commercial use of OSS components continues to expand [1]. Much of the research has focused on evolutionary aspects of open source development in answer to long-term sustainability and viability concerns of community-based software projects [2].

Examples of such research include collecting experiences and building theories of OSS adoption in terms of planning, process improvement, community involvement and software maintenance [3] [4]. Often well-established theories of software evolution, such as Lehman's law [5], are studied in the context of OSS to assess evolutionary and quality characteristics such as survivability, growth potential, maintainability, and ease of adoption.

To keep track of the latest research findings in the area of OSS evolution, there is a need for comprehensive literature studies that summarize and structure the existing body of knowledge. In this article, we present a study for systematic selection, characterization and structuring literature that concerns evolution of open source projects. Our objective, and thus contribution is to produce a systematic reporting of what constitutes the key contributions, the main research gaps, and potential future directions in the field.

In order to perform the study, we have adopted a systematic literature review (SLR) approach [6] for the systematic selection and characterization of existing literature. SLR is a recommended methodology for aggregating knowledge about a specific software engineering topic or research question [7] [8], through the systematic analysis of relevant empirical studies [6]. For example, SLRs were popularly utilized to acquire, conceptualize and structure knowledge in various fields of software engineering including, dynamic analysis [9], fault prediction [10], global software engineering [11], and business process adoption [12].

To carry out this review we adopted a review protocol following the guidelines suggested in [13] and the survey process used in [9]. Keeping the research motivation in mind, we posted 11 research questions in four categories, e.g., target, approach, target group and outcome. Target refers to the different facets and dimensions of OSS projects explored; approach refers to the method, metrics, and tools used for the study; target group refers to the domain of OSS projects studied with selection motives, and finally the outcome refers to the findings and validation of the results reported in the articles. We also discuss the implications of the findings and provide recommendations for future research. The data extracted from the articles are documented under the attribute set developed for answering the research questions. This data is provided in the review website [14] and can be used by the research community to get a holistic view on OSS evolution studies.

The paper is organized as follows: In Section II we discuss the review protocol and the research questions. Answers to the research questions, and a discussion on open areas in the field of OSS and evolution are presented in Section III and IV respectively. Section V discusses validity issues related to the review process. Finally, concluding remarks are presented in Section VI.

II. Review Methodology

Evidence-based Software Engineering (EBSE) relies on aggregating the best available evidence to address engineering questions posed by researchers. A recommended methodology for such studies is Systematic Literature Review (SLR) [6]. Performing an SLR involves several discrete tasks, which are defined and described by Kitchenham in [13]. As a starting point, SLR recommends to pre-define a review protocol to reduce the possibility of researcher bias [13]. Along those guidelines and following

This work was supported in part by the Nokia Foundation Grant and TiSE graduate school funding, Finland.

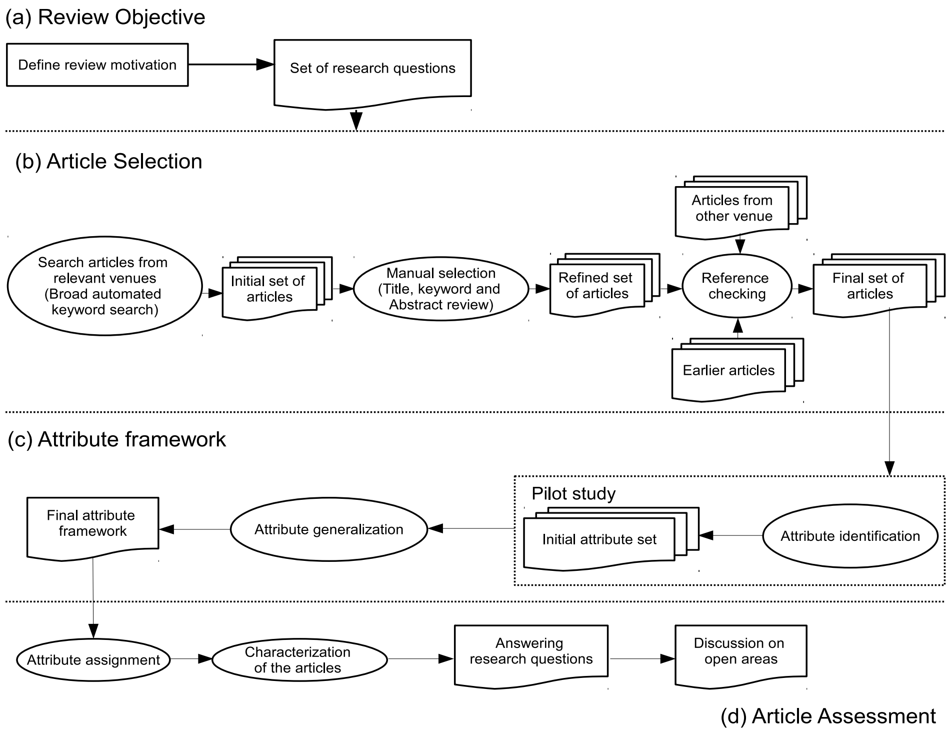


Figure 1. Overview of systematic literature review

the review process described in [9], Figure 1 shows the tasks involved in the review protocol of this study. The tasks are discussed in the subsequent subsections.

A. Research Questions

The research questions we have defined fall within the context of OSS projects and their evolution strategies. In total we have formulated 11 questions, as presented in Table I. These questions are proposed to portray the holistic view of OSS evolution studies. This covers aspects like the focus of the study, methodological detail, case study projects, data sources, and validation mechanisms.

B. Article Selection

This section describes the article selection process (phase (b) in Figure 1) that includes defining the inclusion criteria for article selection, an automated keyword search process to search digital libraries, a manual selection from the initial set of articles, and the reference checking of the listed articles.

Inclusion criteria. Along the research questions shown in Table I, we have defined the following selection criteria in advance that should be satisfied by the reviewed articles:

- Subject area of the articles must unveil strong focus on evolution of OSS projects. Authors must explicitly state the target of the study (e.g., software evolution, community evolution, co-evolution, prediction) and provide detail evidence of research methodology, data sets, and statistical detail of case study projects.
- Articles must exhibit a profound relation to OSS projects and take into consideration those aspects that are particularly attributed to the OSS community and projects. Articles using OSS as a case study are taken into account only if they satisfy the above criterion.
- Articles published in referred journals and conferences are included for the review. Similar to most SLRs, books are not considered for the review.

The suitability of the articles was determined against the above mentioned selection criteria through a manual analysis (discussed later in this section) of title, keywords, abstract. In case of doubt conclusions are checked [15].

Automated keyword search. Automatic keyword search is a widely used strategy in literature surveys [16] [17]. Thus we performed a broad automated keyword search to get the initial set of articles. First author of this article was responsible for the search process. Seven digital libraries were searched: IEEE Computer Society Digital Library; ACM; ScienceDirect; SpringerLink; Google Scholar; FLOSShub and Mendeley. These libraries are the popular sources for open source related research articles. All searches were based on the title, keywords and abstract. The time period for this search was from January, 2000 to January, 2013.

Knowing the fact that construction of search strings varies among libraries, we first defined search terms according to our inclusion criteria. Then to form the search strings, we combined these search terms following the guidelines of the digital library searched. The list of search terms that were used is as follows.

TABLE I.
RESEARCH QUESTIONS

Category	Research Questions	Main Motivation
Target	Which facets of OSS projects were explored and what statistical distribution the articles have in those facets?	To decompose the articles according to their study focus and intensity of studies in each focus area.
	What are the dimensions of OSS projects explored under each study facet?	To determine the specific aspect(s) of OSS projects explored in evolution studies within each facet.
Approach	What are the research approaches followed in the studies?	To identify the general research approach followed in evolution studies (e.g., empirical studies with quantitative or qualitative data analysis).
	What are the datasets or data sources of OSS projects mostly exploited in evolution studies?	To identify the data sources of an OSS project that are used for the evolution studies.
	What metric suits are evaluated and what tools are used for metric data collection?	To explore the metric suits used for evolution study and the popularly used tools for data extraction.
Target group	What is the portfolio of projects analyzed for evolution studies and what are their domains?	To determine the mode of evolution studies (e.g., horizontal or vertical) by statistically measuring the studied OSS projects and their domains.
Outcome	Does the concern on "OSS evolution study" follow an increasing trend?	To identify the beginning and growth of research interest in the field OSS project evolution.
	What contributions are made in literature to analyze the evolution of software?	To explore what results are presented to enhance the understanding of OSS projects evolution. (e.g., do evolution of OSS projects conforms to the theory of software evolution?)
	What contributions are made in literature to analyze the evolution of organization or community?	
	What contributions are made in literature to analyze the interdependency in the evolution of the software and organization?	
	How are the research approaches and results of the articles typically validated?	To identify the approaches employed to evaluate the research approaches and study results (e.g., internal validity, external validity, construct validity).

Terms representing OSS: "Open source" or OSS or "Open Source Software" or "Open Source Software projects" or FLOSS or "Libre Software" or "F/OSS".

Terms representing evolution study: "evolution" or "structural evolution" or "evolution of software" or "project evolution" or "project history" or "software evolution" or "community evolution" or "co-evolution".

Automated keyword search ended up with 181 articles consisting of 46 journal articles and 135 conference articles.

Manual selection. Recent studies [15] [9] pointed out that (a) current digital libraries on software engineering do not provide good support for automated keyword search due to lack of consistent set of keywords, and (b) the abstracts of software engineering articles are relatively poor in comparison to other disciplines. Thus it is possible that the 181 articles identified through automated search process might contain irrelevant ones and some relevant might be missing. Due to this fact the first author performed a manual selection on these articles by reviewing the title, keywords and abstract (and in case of doubt, checking the conclusion [15]). To reduce the researcher bias in this selection process, the domain experts (second and third author) examined the selected articles against the selection criterion. Any disagreement was resolved through discussion. This process ended up with 97 articles consisting of 21 journal articles and 76 conference articles.

Reference checking. To ensure the inclusion of other relevant but missing articles (as mentioned above), the first author performed a non-recursive search through the references of the 97 selected articles. This process identified 4 additional conference articles.

Final set of articles. The article selection process finally ended up with 101 articles (21 journal and 80 conference articles). A complete list of these articles along with year and venue wise distribution can be found in our review website [14].

C. Attribute Framework

The next step in the review protocol was the construction of an attribute framework (phase (c) in Figure 1). This framework was used to characterize the selected articles and to answer the research questions. Following is a brief description of this process.

Attribute identification. The attribute set was derived based on two criteria: (a) The domain of the review (i.e., evolution of OSS projects) and (b) the research questions. A pilot study was run for this step, as shown in phase (c) of Figure 1. This phase consists of a number of activities.

First, we performed an exploratory study on the structure of 10 randomly selected articles (from the pool of 101 articles). This study led to a set of eight general attributes that can be used to describe the articles and to answer the research questions. This attribute list is shown in the *Attribute* column of Table II.

Second, this list of attributes was refined further into a number of specific sub-attributes to get precise

TABLE II.
ATTRIBUTE FRAMEWORK

Attribute	Sub Attribute	Brief Description
General		Publication Type, Year of Publication
Study Type		Empirical, comparative, case study, tool implementation.
Study Target	Software evolution	Code, architecture, bug/feature
	Community evolution	Developer and user community
	Co-evolution	Combined evolution of software and community
	Prediction	Studies on predicting evolution of OSS projects
Case Study	OSS projects studied	List of OSS projects studied
	Programming language	Target programming languages of OSS projects
	Project size	Size measure of OSS projects (in KLOC for latest release)
	Project domain	Application domain of the OSS projects covered
Data Source	Source code	Code base, CVS/SVN
	Contributions	Change log, bug tracking systems
	Communication	Mailing list archive, chat history
	External sources	Sourceforge, github, ohloh.
Methodology	Methods	Concrete methods applied
	Metrics	Type of metrics used
	Tool implementation	Tools implemented for the study
	Tools used	Existing tools, algorithms used for study
Results	Growth rate	Defines the growth rate of an OSS project during its evolution.
	Measure of evolution	Qualitative, Quantitative
	Prediction classification	
	Summary	Other findings
Evaluation / Validation		Validation process for a study

description of each of the general attributes and fine tune the findings on the research questions. To do this, we made a thorough study of the same set of articles and wrote down words of interest that could be relevant for a particular attribute (e.g., "software evolution", or "community evolution" or "co-evolution" for *Study target* attribute). The result after reading all articles was a (large) set of initial sub attributes. This data extraction task was performed by the first author of this survey.

Attribute generalization and final attribute framework. We further generalized the attributes and sub-attributes to increase their reusability [9]. For example, sub-attributes "mailing list archive" or "chat history" are intuitively generalized to *Communication*. This final attribute list was then examined and validated by the domain experts (second and third authors). This reduces the change of researcher bias, as neither of the domain experts had any connection with this process. The final attribute framework is shown in Table II.

D. Article Assessment

The article assessment step consists of four distinct activities as shown in phase (d) of Figure 1. In this section we focus on the first two steps.

Attribute Assignment. Using the attribute framework from the previous section, we processed all articles and assigned the appropriate attribute sets to each of the articles. These attributes effectively capture the essence of the articles in terms of the research questions and allow for a clear distinction between (and comparison of) the articles under study.

The assignment process was performed by the first author of this survey. During this process, authors' claim of contribution is assessed against the results presented in the articles. For example, to validate the claim on the target of the study (e.g., software or community evolution), we assessed what relevant data sources are explored, what metrics and methods are used, and the duration and process of data collection. Also, we did not draw any conclusions from what was presented in an article if it was not explicitly mentioned. For example, we left the attribute field *study type* empty if it was not mentioned in the article.

Characterization of the reviewed articles. Since the attribute assignment process is subject to different interpretations, different reviewers may predict different attribute subsets for the same article [9]. As the attribute assignment process is carried out by the first author of this paper, the quality of the assignment needed to be verified to avoid reviewer bias [9]. This verification task was carried out by the domain experts who assessed the data collection table against the reviewed articles. Any disagreements were resolved through discussion. This characterization of articles is presented in our review website [14].

Next we discuss the results of this review by answering the research questions and discussing open areas in this field.

III. Review results

Given the article selection and attribute assignment (as presented in review website [14]), the next step is to present and interpret the study findings. We start with discussing answers to the research questions based on the study outcome. List of OSS projects that are studied in the review articles are provided in the website [14].

RQ1. Which facets of OSS projects are explored *and what statistical distribution the articles have in those facets?*

An in-depth study on the selected articles led us to decompose the OSS evolution articles into four facets:

- Software evolution: articles under this facet explore evolutionary behavior of OSS systems and derive patterns of evolution to evaluate them against the laws of software evolution. Such studies also measure the issues that concern the commercial world. This includes for instance, study the evolutionary patterns of code complexity, maintainability, sustainability, and quality in an OSS project.
- Community evolution: articles under this facet studies how the social networks of developers and users evolve over time while building the product.
- Co-evolution: articles under this facet examine the evolution of OSS systems with the associated communities, and explore relationship between the two through different collaboration models.
- Prediction: articles under this facet deal with defining and examining prediction models to simulate the evolution of OSS projects. For instance, developing methods to support error prediction for the purpose of preventive maintenance and building quality software.

Figure 2 shows the distribution of articles (published in both journal and conferences) under each facet.

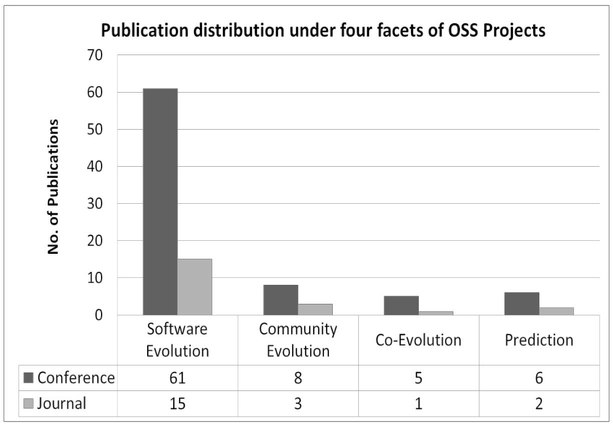


Figure 2. Article distribution under each facet of evolution study

From this figure it is evident that the facet *software evolution* got maximum attention over others. The reason of such bias distribution of articles can be defended by the fact that the development history of OSS projects is relatively new compared to its proprietary counterpart [18].

Software evolution is the most studied facet.

RQ2. Does the interest on"OSS evolution study" follow an increasing trend?

OSS development has appeared and diffused throughout the world of software technology, mostly in the last ten to thirteen years. During this period, a growth in interest for better understand the patterns of OSS evolution has been noticed. The increasing trend in number of publications between the year 2000 and 2012 (as shown in Figure 3) assist this claim.

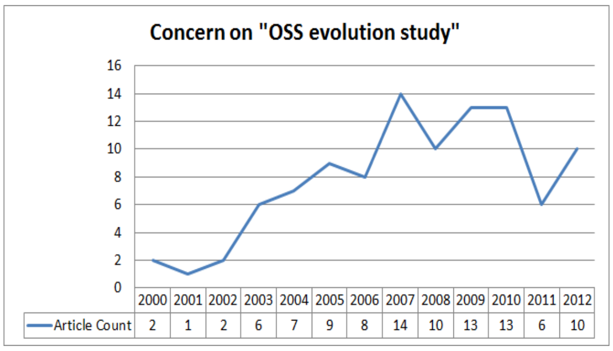
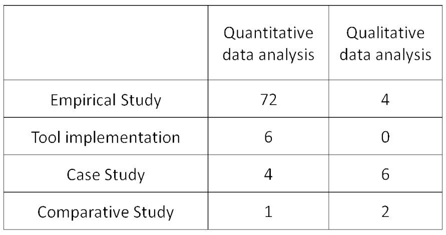


Figure 3. Concern on "OSS evolution study" over the decade

Research on OSS evolution follows an increasing trend.

RQ3. What research approaches are followed in the studies?

Research methodologies followed in the reviewed articles can be categorized into four distinct approaches: empirical study, case study, comparative study and tool implementation. Each of these studies use OSS project data for either quantitative analysis or qualitative analysis. Figure 4 shows the count of published articles according to this classification. As can be seen from the Figure, 75% of the studies (76 articles out of 101) followed empirical approach with either quantitative or qualitative data analysis.



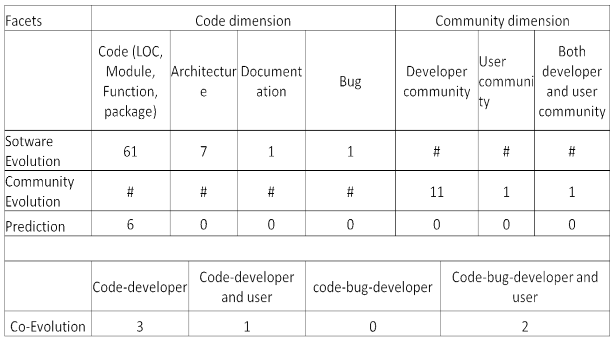
	Quantitative data analysis	Qualitative data analysis
Empirical Study	72	4
Tool implementation	6	0
Case Study	4	6
Comparative Study	1	2

Figure 4. Distribution of articles under the classification of research approaches followed in the studies

Empirical research is the most frequent research methodology used to study OSS evolution.

RQ4. What are the dimensions of OSS projects explored under each study facet?

This research question gives a fine grained view on the dimensions of OSS projects explored by the evolution articles. Figure 5 provides a two dimensional view of OSS projects, e.g., code and community dimensions with their constituent parts. As can be seen from this figure, software evolution and prediction facets mostly utilize the code dimension. Whereas the community evolution facet puts more emphasis on developer community than user community or their combination. The study on co-evolution of the code and community mostly explores the code base, bug reports, developer and user community.



Facets	Code dimension				Community dimension		
	Code (LOC, Module, Function, package)	Architectur e	Document ation	Bug	Developer community	User communi ty	Both developer and user community
Sotware Evolution	61	7	1	1	#	#	#
Community Evolution	#	#	#	#	11	1	1
Prediction	6	0	0	0	0	0	0

	Code-developer	Code-developer and user	code-bug-developer	Code-bug-developer and user
Co-Evolution	3	1	0	2

Figure 5. Dimensions of OSS projects that are explored to study a facet, # means not applicable

Code dimension (e.g., source code) is mostly studied in the articles as compared to community dimension or their combination.

RQ5. What is the portfolio of projects analyzed for evolution studies and what are their domains?

In general, the study of evolutionary behavior and patterns of OSS projects requires access to historical data representing their development, growth and success story. These studies thus delimited to flagship OSS projects that are large in size with a large user and developer community and belong to popular application domains. In this regard, our findings reported that most of the OSS projects studied are from the domain of Operating Systems (OS), Application Software, Integrated Development Environments (IDE), Application Servers, Libraries, Desktop Environments and Frameworks. Example projects under these domains include Linux, Eclipse, Apache, Ant, Mozilla, GNOME, KDE, and ArgoUML. These projects have more than 5 years of development and evolution history. Figure 9 in the appendix presents the domain wise classification of the studied OSS projects with the count representing their frequency of use in the evolution studies. This finding gives support to the fact that OSS evolution studies are mostly vertical and thus unable to put light on the whole population of OSS projects, as vast majority of projects are failures [19]. Only a few articles, according to our study, report horizontal studies with a large and random sample of OSS projects (ranging between 200 to 4000 OSS projects).

Large and successful OSS projects are often selected as case study projects.

RQ6. What are the datasets or data sources of OSS projects mostly exploited in evolution studies?

To analyze the evolutionary behavior of OSS projects, information contained in project data sources need to be explored. These data sources are termed as repositories which contain a plethora of information on the underlying software and its development processes [20] [18]. Studies based on such data sources offer several benefits: this approach is cost effective, requires no additional instrumentation, and does not depend on or influence the software process under consideration [20]. Evolution studies on the OSS projects effectively explored these repositories produced by the projects as well as the external sources. Figure 6 presents the OSS repositories in both categories and the count of articles that utilizes those repositories. According to this figure, repositories maintaining the code base (e.g., CVS/SVN, change log) are the most explored sources. This is obvious because most of the articles (as discussed in RQ1) studied either the evolutionary patterns or the prediction models for the evolution of the system. Among the external sources, SourceForge.net is the most popular repository hosting thousands of OSS projects and having the maximum number of downloads.

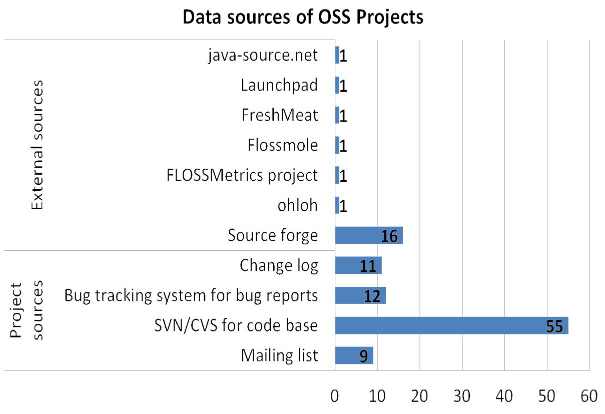


Figure 6. Data sources of OSS Projects

Repository maintaining the source code of the projects are mostly explored in the studies.

RQ7. What metric suits are evaluated, and what tools are used for data collection and analysis?

The reviewed articles used metrics mostly to measure the evolutionary patterns and antecedents of certain aspects of the studied projects, such as, code complexity, structural complexity, architectural patterns, predicting error proneness and maintainability, and collaboration patterns within the community. Mostly empirically validated metrics are selected for these studies. Widely used metric suites are listed in Table III.

For metric data collection, synthesis and interpretation, a number of existing tools are used. Figure 8 in the appendix provides a list of such tools used along with

TABLE III.
METRICS

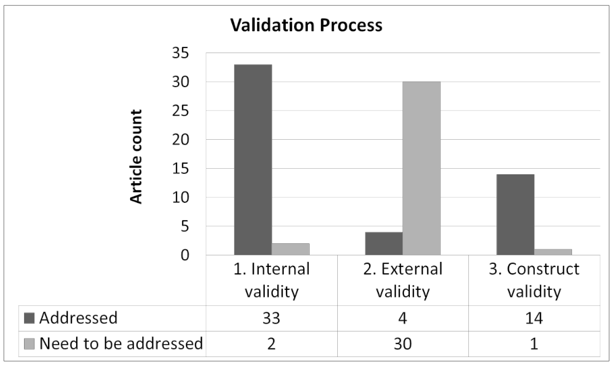
Metric Category	Example Metrics
Source code metrics	source line of code, line of code, number of functions
Code complexity metrics	interface complexity, Halstead suite of complexity metric, cyclometic complexity, structural complexity
Object oriented metrics	Chidamber and Kemerer, L&K (Lorenz and Kidd's eleven metrics, Li's metric suite for OO programming, modularity metrics
Product level metrics	product size, releases, application domain, version frequency
Project metrics	metrics related to the OSS community structure and communication, application domain, number of developers, users, project popularity, success, application domain, no of commits, no of messages sent

their usage area and popularity count. As most of these tools are third party applications, the accuracy of the data collection and analysis is constrained by the performance of these tools. This also puts impact on the validity of the results.

Empirically validated metric suites to evaluate the source code are mostly used in the articles.

RQ8. How are the research approaches and results of the articles typically validated?

Threats to validity in experimental studies can be categorized into three types: external, internal and construct validities [21]. External validity means to what extent the results can be generalized to the whole population (or outside the study settings). Internal validity means that changes in the dependent variables can be safely attributed to changes in the independent variables. Construct validity means that the independent and dependent variables accurately model the abstract hypotheses.



	1. Internal validity	2. External validity	3. Construct validity
■ Addressed	33	4	14
▨ Need to be addressed	2	30	1

Figure 7. Validation Process of the research approaches

Our survey results concerning these validity issues reveal that 41% (42 out of 101) of the articles measured and reported the validity threats of the underlying research methodology and the research result. Figure 7 plots the number of articles that perfoms validity measure under each category. In this figure, the articles which took quantifiable measure to minimize a validity threat are counted under *addressed* field and the articles which admited the threats as a delinquent to the study are counted under *Need to be addressed* field.

Almost all the studies (30 out of 34) suffers from external validity threats and thus suffers from generalizability of the results to the population of OSS projects.

RQ9. What contributions are made in literature to analyze the evolution of software?

Analysis of the selected articles identifies a good sample of empirical studies which were conducted to verify the fitness of the Lehman's law of software evolution in the domain of OSS projects. These results have both conformance (either complete or partial) and contradiction with the laws of software evolution. In Table IV we provide a comprehensive summary of these studies. We believe this would provide a holistic view on the suitability of these laws in OSS domain, and will create the future pathway in deriving evolutionary patterns and laws for OSS evolution.

Other empirical analysis on OSS projects reveal several stimulating properties/characteristics of the system evolution. In Table V we summarizes these findings according to their primary focus of study.

RQ10. What contributions are made in literature to analyze the evolution of organization or community?

OSS projects typically come with a highly distributed community of developers and users. Members of such community share a common interest in the project. They often interact with one another to share knowledge, and collaborate in contributing and developing the project [50]. Communities are the core of OSS projects, and for the successful evolution of a project, it should have a large number of developers (authors and contributors) [51]. Other studies to date, report distinct properties of OSS communities. Such as, necessity of a critical mass of the core developer team [52], motivation of joining a project [50], change of roles of the community members through contributions [53], social dynamics and pattern of the community [54] [55]. These results are summarized according to the study focus in Table VI.

RQ11. What contributions are made in literature to analyze the interdependency of the software and organization evolution?

The successful evolution of OSS projects depends on the co-evolution of the community (both developer and user) and the software [5]. Related research in this track identifies that the number of contributors grows with the growth in product size [30]. This observation is replicated in [65], with evidence that the increase of introduced packages and reported bugs are highly coherent with the increase of contributors and active users, respectively. Also, the increase in documentation and modularization

TABLE IV.
FITNESS OF LEHMAN'S LAW IN OSS PROJECTS

Ref	Growth Rate	Brief Description
[3]	Super-linear growth.	Super-linear growth pattern in system level is due to accumulated linear growth of the large set of driver subsystems. Contradict with Lehman's 4th law.
[22]	Super-linear growth.	The project has doubled in size in terms of SLOC and number of packages in every 2 years.
[23]	Super-linear growth.	Contradicts Lehman's 1st and 3rd law.
[24]	Linear or Super-linear growth.	Majority of the large projects grow linearly with few having a super linear growth. Both contradict with Lehman's law.
[25]	Average growth in size is 17%, while decreasing in structural complexity on average 13%.	Contradicts Lehman's second law.
[26]	Super-linear growth.	The growth is due to increase in time interval between subsequent releases in recent past.
[27]	Super-linear growth (for large project). Linear or Sub-linear growth (for small project).	Large projects are large in size (in LOC), more active in number of revisions, and have more programmers than those are not.
[28]	Linear and Super-linear growth.	Linear pattern was noted for NOF, NOC, LOC and size measures. Super-linear growth for plug-ins and downloadable source code.
[29] [30] [2] [31]	Linear growth.	Conformance to all six laws of Lehman considering the growth in size, coupling and complexity.
[4]	Linear growth.	• Conformance to Lehman's 6th and 2nd law (growth pattern measured in source code level, file level, module level and complexity). • Conformance to Lehman's 1st and 5th law holds (according to changing rate and growing rate of handled uncommented line of code). • Contradicts with Lehman's 6th law (for file level growth).
[3]	Linear or Sub-linear growth.	16 out of 18 studied systems follow a growth pattern linear or close to linear.
[32] [33]	Not mentioned.	Conformance to Lehman's 1st, 2nd, 3rd and 6th law.
[32] [33]	Not mentioned.	Contradicts with 4th,5th,7th and 8th law.
[34]	Linear growth.	A declining at linear rate was noticed at module level.
[35]	Linear growth.	Follows a pattern of stagnated growth with decrease in size at some points.
[36]	Linear growth.	• The evolution pattern do not consistently conform to Lehmans laws. • Decrease in structural complexity (e.g., McCabe cyclomatic complexity). • Increase in calculation logic (e.g., Halstead complexity increased). • Increase in modularity.

levels are obliged to rising number of developers and their contribution in a project [59]. In this regard, more presence of users in the community drives more changes in the code base.

IV. Avenue to Future Research

The final step of the survey (see Figure 1) consists of formalizing the tacit knowledge acquired through the study of the review articles in order to distill further research directions. To conduct this step, we analyzed the results reported in section III to identify gaps, commonalities and contradictions, and look for most and least frequently used research approach in each facet.

ON SOFTWARE EVOLUTION

Understanding the most common study facets (as displayed in Figure 2) gives the impression that *open source software evolution* is the most widely investigated field. Research under this facet has produced good sample of analytical results which are available for further examination, assessment and comparison [5]. Our review on this direction suggests following research directions,

Law's of evolution for Open Source Software. The most common study topic under this facet is to evaluate the fitness of Lehman's law of evolution to open source software. These results are summarized in Table IV. A closer scrutinize to the table data reveals that the results have both facsimile and contradicting to the Lehman's law. For instance, the growth rate of OSS, presented in column 2 of Table IV varies between super-linear (i.e., greater than linear) and sub-linear (i.e., less than linear). This has both conformance and contradiction with the second and sixth law of evolution.

Comprehension of these results suggest that the laws and theory appear to be breaking down through non-conforming data and findings (Table IV). Thus Lehman's laws of software evolution which is primarily based on the

TABLE V.
EVOLUTION OF THE SOFTWARE

Focus of Study	Results	Reference
Code Complexity Evolution	Project size increases with an improvement on some quality measures, e.g. decrease in complexity of the system.	[37]
	The procedure and file level complexity remains unchanged.	[38]
	• Decline trend in structure complexity (e.g., McCabe cyclomatic complexity). • Increase in calculation logic (e.g., Halstead complexity). • Increasing trend in system's modularity.	[39]
	Increase in coupling, interface complexity and cyclomatic complexity.	[2]
	System level decay in terms of the underlying structure due to the addition of new folders and functionalities.	[40]
	Increase in complexity for large files as they undergo frequent changes.	[41]
	Decrease in modularity due to major architectural and implementation changes.	[42]
	There exists positive correlation between error probability across classes and the code bad smells.	[43]
Code Quality Improvement	Modularity of the software system increase due to periodic refactoring and cleanup activities after major changes.	[42]
	Periodical refactoring is needed to prevent system decay and stagnation, and to improve architectural quality.	[41] [43]
	The development strategies and practices of OSS projects support to maintain the reliability and quality of the software.	[44]
	OSS is less entropic than proprietary applications having low unit maintenance costs.	[45]
Code cloning	Code cloning (or code duplication) does not have a large impact on post-release defects (quality).	[46]
	Presence of duplicate code in the software does not make it more difficult to maintain.	[47]
Documentation	The documentation process often start with an initial upfront maintenance effort (to create the initial documentation or writing a book), which is then updated according to the changes in the project.	[45]
Sub-project Evolution (projects under a project)	• Existing sub-projects might be merged or removed. • New sub-projects might be introduced. • Sub-projects might follow different trend models in the growth, complexity and changes. E.g., Eclipse.	[28]
OSS Dynamics	• SOC (Self Organized Criticality) occurs during the evolution of OSS. • SOC can be used as a conceptual framework for understanding OSS evolution dynamics.	[48]
	• The evolution dynamics of OSS may not follow SOC. • The past of an OSS project does not determine its future except for relatively short periods of time.	[49]

study of the large close source systems, is not sufficient to justify or account for the evolutionary pattern and behavior of the open source software. As none-the-less these laws did not consider the community dimension of the OSS projects which is an integral part of sustainable evolution of the open source software.

To deal with this problem, a viable route would be to examine the underlying ontologies for software evolution [5] considering the OSS specific characteristics, and then re-assess the laws of software evolution to fit in OSS domain.

Metric set for software evolution. Software evolution studies mostly utilize metrics that are empirically validated in prior studies (as presented in Table III). These metrics are derived for closed source projects, and are primarily used to verify the Lehman's law of software evolution. Though these metrics provide valuable insight to OSS evolution, they do not consider the community dynamics. Thus an empirically validated set of metrics in favor of explicit representation of the community is required to complement the existing metric set.

Predicting the future. Prediction of OSS projects is one area that is least popular among the study facets (Figure 2). Yet future research should focus on developing reliable prediction models and methods supporting error prediction, measuring maintenance effort and cost of OSS projects. Because, the commercial organizations, for instance, requires such prediction models to assess an open source component for adoption [66].

Study the existence of SOC. Another direction of research would be to study the notion of SOC (Self Organized Criticality) in OSS projects. SOC dynamics articulate that the current state of a project is determined (or at least, heavily influenced) by events that took place long time ago. Existential exploration of SOC in the domain of OSS projects reveals contradictory results (Table V). Thus future research can take further step in validating the existence of SOC and its implication on the evolution of open source software.

TABLE VI.
ORGANIZATIONAL (OR COMMUNITY) EVOLUTION

Focus of Study	Contribution	Reference
Community Formation	A group of core developers quickly emerges at the center of the community at the early stage of the project and becomes the key contributors.	[54]
	The OSS community evolution follows small world property with a strong community structure and modularity. Initially it has fairly dynamic nature which gradually settles down into fixed groups.	[56]
	During the evolution, the communication between core and peripheral developer's decreases, and sub-community of developers forms.	[57]
	The growth of OSS community follows "rich gets richer" phenomenon. This means a healthy sized community often attracts new developers.	[58]
Community Structure and Activity	• 57% of the studied projects has only one or two developers. • Only 15% has more than 10 developers. This category constitutes only flagship projects.	[51]
	• About 83% projects have only one or two stable developers. • 16% of the projects attains the size of the core team.	[59]
	Developers play different role ranging from core developers to passive users. These roles are implicit, and are mainly defined and determined according to the level of contributions made to the project.	[53]
	Developer's role changes through accumulated contributions over a period of time. Changes in developer roles and community structure are significantly associated with the quality of contributions, but not with contribution quantity.	[53] [57]
	A Pareto distribution on the size of the developer community was identified. That is a vast majority of the OSS projects fail to take off and soon become abandoned. Probable reasons for such failure include projects inability to attract developers to attain a critical mass of developers, and insufficient communication and collaboration.	[60] [54]
Community Migration	• The migration of old developers from previous release to a new one is very high. • Developer's code maintenance activity increases with increase in experience. • The community has a natural 'regeneration' process for its voluntary contributors. This process increases the probability of code adoption that are left by the outgoing developers.	[32]
	Core developers should promote community regeneration process by creating an organizational ecosystem. The ecosystem will provide the openness of the system, process and communication which will attract others to join.	[61]
Sustainability	• OSS development is prominently a community-based model. • The community must be a sustainable community for the long term survivability of the project.	[62] [50]
	• A stable and healthy core team in the community is essential for the sustainability of OSS projects. • Core developers introduce less structural complexity and remove existing structural complexity from the code.	[2]
	Sustainability can be credited to the following motivating factors: (a) people are benefited from participating in a thriving OSS community, (b) improve technical competence as a developer, and (c) projects have well-defined modular design and cheap means of communication.	[63]
	It is important to maintain a balance composition of all the different roles in a community for sustainability. For example, to an extreme, if most of the community members are passive users then the system will not evolve.	[64]
Evolution of Sub-communities (Communities of sub-projects)	• Ecology of the sub-communities are formed around the sub-projects. • Sub-projects are often governed by a common governance, e.g. Apache. • Members in sub-communities often collaborate due to mutual task dependencies. • Sub-communities also compete for the project resources.	[50]

ON COMMUNITY EVOLUTION

Study on the community evolution identifies several key properties (reported in Table VI), which lay the foundation for further research in this direction. We propose the followings to be investigated.

Community building. Studies reported that the majority of OSS projects failed to attract members to attain the critical mass. Only few flagship projects are able to attract developers. Factors influencing the motivation to join a community has been studied (e.g., [62] [50]), and several phenomena are proposed. For instance, rich gets richer phenomenon. Yet it is not identified what exclusive properties initiate the community building process at the nebula stage of the project. Following research questions can be considered relevant,

- Why some projects are able to attract contributors during the nebula stage of the project, while most of them can not?
- What formation of the community refers to a balance one, and how the community structure changes towards a balance structure during its evolution?

- Can a visible pattern be identified within the domain of OSS projects for the above two cases?

Migration of responsibility and sustainability. It has been reported that migration of developers from one release to the next is high and that the developers take more responsibility as they gain experience. Yet it is a common phenomenon in open source domain that developers freely join or leave the project. And when a developer leaves, his responsibilities must be assigned to someone else. For instance, the codebase maintained by a outgoing developer should be taken care of by others. Else it will be abandoned and discarded from subsequent releases. Thus it will be beneficial to explore the followings,

- How responsibility migrates among the developers? Does this migration follow preferential-attachment?, i.e., is the responsibility handed over to the developers who are in close connection to the outgoing developer.
- What impact such migration has on the project evolution?

ON CO-EVOLUTION

It is turned out from our review that the understanding of co-evolution of the code and the community in OSS projects has received little attention in literature (Figure 2). As a consequence, the community dimension and corresponding communication channels (e.g., mailing archives, bug tracking systems) are explored seldom, as can be seen from Figure 5 and Figure 6 respectively. Study on co-evolution in OSS projects, however, is becoming increasingly popular. Because, in such projects the code evolution is dependent on the contribution of community members, and that a successful evolution of the code is required for the survival of the community. The following research directions can be considered relevant.

Exploring socio-technical congruence. In the OSS projects contributions made by the community members not only drive the system evolution but also redefine the role of these contributing members and thus change the social dynamics of the OSS community [53]. In this connection, it will be very interesting to investigate the phenomenon *socio-technical congruence* in OSS projects. Socio-technical congruence which is a conceptualization of Conway's law [67] states that there should exists a match between the coordination needs established by the technical domain (i.e., the architectural dependency in the software) and the actual coordination activities carried out by project members (i.e., within the members of the development team) [67]. This concept was already explored in closed source projects, and reported a high correlation with software build success, quality, and faster rate of modification [68]. Thus socio-technical congruence plays a pivotal role in conceptualizing the co-evolution in a project. Surprisingly, this notion as a research area has not been given much attention among open source researchers. Although it is identified and reported as a desired property for collaborative development activities like OSS projects [69]. Considering the lack of focus in this direction, we propose the following to investigate.

- Does the essence of socio-technical congruence as a conceptualization of Conway's law holds for OSS project? Can it be stated as an implicit characteristics or property of successful OSS project?
- What quantitative approach/method can be utilized to verify the existence of socio-technical congruence in OSS projects? What repositories can be used for this purpose?
- What correlation can be derived between socio-technical congruence and the quality/sustainability of OSS projects?

Sub-project evolution with their community. Large open source projects often encompass many sub-projects. Such as, sub-projects in Eclipse, GNU, Linux, and Apache. Often ecology of sub-communities formed around these sub-projects, which are governed by a common governance [50]. Study on the formation and evolution of sub-projects and their communities have revealed many key characteristics, which are listed in Table V and Table VI, respectively. Yet the interdependency in evolution between the two and their impact on the overall project evolution remain untouched. The following would be worth to investigate.

- Does there exist a correlation between the evolution (growth, complexity, change) of the sub-projects and their associated sub-communities? Does the community change with the change in the sub-project?
- How does a community form around a newly added sub-project?
- What attributes of a sub-project attract new developers to join?
- What happens to the sub-community when a sub-project is deleted or merged to other sub-project?
- What dependencies lead to inter project communication?
- What kind and level of communication and collaboration takes place between sub-communities?
- Does there exist a correlation between the project evolution and the sub-project evolution?

ON RESEARCH METHOD

A number of issues related to the research approach can be improved to increase the acceptability of the reported results. We pointed out the followings,

External validity of the results. Empirical study is the most popular research approach employed in evolution studies (Figure 4). These studies, however, are horizontal in nature (as reported in RQ5) considering only flagship OSS projects. Due to this approach of studying OSS projects, the reported results suffer from generalizability threat, as reported in Figure 7. Yet to make these finding applicable and hold for the extended region of OSS projects, explicit measure should be taken. An interesting route to deal with this is to categorize the findings (current or future) according to the project domain, or similar organizational structure and practices, or similar product size and complexity. This will reveal the broader picture which can then be compared and possibly merged for proposing a more general evolutionary pattern and behavior for OSS.

Framework for the data collection and representation. As discussed in RQ6, OSS projects often produce large volume of data representing their development and evolution history. Research to date, explores the repositories that maintain these data, a list of which is provided in Figure 6. However, data collection and representation in these repositories vary significantly from project to project. Furthermore, data from the same source may have different formatting (e.g., emails are often free of format even in listing the senders credentials). Due to these facts, it is a challenging task to collect relevant data following a standard format from OSS repositories. In this context, researchers often employ their own means to collect and represent data for research. This reduces the compatibility and comparability of the reported results even if they use same data sources. Taking these issues in consideration, a framework for uniform data collection and representation can be developed to make the results cohesive and comparable to each other.

V. Threats to Validity

Carrying out a survey is mostly a manual task. Thus most threats to validity relate to the possibility of researcher bias [9]. To minimize this, we adopted guidelines on conducting SLR suggested by Kitchenham [13]. In particular, we documented and reviewed all steps we made in advance, including selection criteria and attribute definitions.

In what follows, the description related to validity threats pertaining to the article selection, the attribute framework, and the article characterization is discussed.

A. Article Selection

Following the advice of Kitchenham [13], the inclusion criteria is set at the time of defining the review protocol, and the criteria are based on the research questions. This reduces the likelihood of bias. Articles satisfying this selection criterion are considered. For collecting relevant articles we first performed automated keyword search and then performed manual selection. The first step condenses the selection bias whereas the latter ensures the relevance of the selected articles. Finally, a non recursive search through the references of the selected articles is performed. This increases the representativeness and completeness of our selection. To further minimize the selection bias and reviewer bias, domain experts (second and third author) verified the relevance of the selected articles against the selection criteria.

B. Attribute Framework

The construction of the attribute framework may be the most subjective step [9]. Thus we take the following steps to acknowledge this fact: the attribute set is derived based on the research questions and domain of study. Then a pilot study is carried out to further refine the attribute framework. Furthermore, the representativeness of the framework is examined by domain experts (second and third author).

C. Article Assessment

Similar to the construction of the attribute framework, the process of assigning the attributes to the research articles is subjective and may be difficult to reproduce [9]. We address this validation threat through an evaluation process where domain experts assess the collected data against reviewed articles.

VI. Discussion

In this paper we have reported a systematic literature review (SLR) on the evolution studies of Open Source Software projects. To carry out this study we adopted a review protocol following the guidelines presented in [13] and [9]. A set of 101 articles (21 journal and 80 conference articles) were selected for the review. Through a detailed reading of a subset of the selected articles, we derived an attribute framework that was consequently used to characterize the articles in a structured fashion.

We also posed a set of research questions in advance that are investigated and answered throughout the study. The attribute framework was sufficiently specific to characterize the articles in answering the research questions. The set of articles and collected data under this attribute framework is presented in our review website [14]. None-the-less, an elaborated discussion on the validity of the review process is also presented.

The characterization of the reviewed articles will help researchers to investigate previous studies from the perspective of metrics, methods, datasets, tool sets, and performance evaluation and validation techniques in an effective and efficient manner. We also put an elaborated discussion on the most significant research results. In summary, this article provides a single point reference on the state-of-the-art of OSS evolution studies which could benefit the research community to establish future research in the field.

Related works in this track carried out a literature review on open source software evolution [70]. This study explores the software evolution, mostly emphasizing on research methods, metrics, and data analysis. Contrast to this, our review provides a holistic view of the evolution of OSS projects, concerning all the facets studied to date.

Yet our reported result pertaining to the conflicting reporting of Lehman's law of software evolution confirmed the findings in [70]. This suggests that future work in the area of OSS evolution should explore more to unify the findings through comprehensive study on the the open areas discussed in this paper.

REFERENCES

[1] R. Grewal, G. Lilien, and G. Mallapragada, "Location, location, location: How network embeddedness affects project success in open source systems," in *Management Science*, vol. 52, no. 7, 2006, pp. 1043–1056.

[2] S. Suh and I. Neamtiu, "Studying software evolution for taming software complexity," in *Australian Software Engineering Conference*, 2010, pp. 3–12.

[3] G. Robles, J. Amor, J. Gonzalez-Barahona, and I. Herraiz, "Evolution and growth in large libre software projects," in *IWPSE'05*, 2005, pp. 165–174.

[4] C. Roy and J. Cordy, "Evaluating the evolution of small scale open source software systems," in *CIC 2006*, 2006, pp. 123–136.

[5] W. Scacchi, "Understanding open source software evolution: Applying, breaking, and rethinking the laws of software evolution," in *Applying, Breaking, and Rethinking the Laws of Software Evolution*. John Wiley and Sons Inc, 2003.

[6] B. A. Kitchenham, R. Pretorius, D. Budgen, O. P. Brereton, M. Turner, M. Niazi, and S. Linkman, "Systematic literature reviews in software engineering- a tertiary study," *IST*, vol. 52, no. 8, pp. 792–805, 2010.

[7] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software," in *Engineering Technical Report EBSE-2007-01*, 2007.

[8] M. Petticrew and H. Roberts, "Systematic reviews in the social sciences: A practical guide," in *Blackwell Publishing*, 2005.

[9] B. Cornelissen, A. Zaidman, A. Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *TSE*, vol. 35, no. 5, pp. 684–702, 2009.

[10] C. Catal and B. Diri, "A systematic review of software fault prediction studies," *Expert Systems with Applications*, vol. 36, no. 4, pp. 7346–7354, 2009.

[11] D. Łmite, C. Wohlin, T. Gorschek, and R. Feldt, "Empirical evidence in global software engineering: a systematic review," *ESE*, vol. 15, no. 1, pp. 91–118, 2010.

[12] A. Pourshahid, D. Amyot, A. Shamsaei, G. Mussbacher, and M. Weiss, "A systematic review and assessment of aspect-oriented methods applied to business process adaptation," *JSW*, vol. 7, no. 8, pp. 1816–1826, 2012.

[13] B. A. Kitchenham, "Procedures for performing systematic reviews," in *Technical Report TR/SE-0401, Keele University, and Technical Report 0400011T.1, National ICT Australia*, 2004.

[14] M. M. Syeed, "http://reviewossevolution.weebly.com/," 2013.

[15] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "Lessons from applying the systematic literature review process within the software engineering domain," *JSS*, vol. 80, no. 4, pp. 571–583, 2007.

[16] S. Beecham, N. Baddoo, T. Hall, H. Robinson, and H. Sharp, "Motivation in software engineering: A systematic literature review," *IST*, vol. 50, no. 9-10, pp. 860–878, 2008.

[17] T. Dyba and T. Dingsyr, "Empirical studies of agile software development: A systematic review," *IST*, vol. 50, no. 9-10, pp. 833–859, 2008.

[18] D. Atkins, T. Ball, T. Graves, and A. Mockus, "Using version control data to evaluate the impact of software tools," in *ICSE*, 1999, p. 324333.

[19] C. B. K. Beecher, A. Capiluppi, "Identifying exogenous drivers and evolutionary stages in floss projects," *The Journal of Systems and Software*, vol. 82, no. 5, pp. 739–750, 2009.

[20] J. Cook, L. Votta, and A. Wolf, "Cost-effective analysis of in-place software processes," *TSE*, vol. 24, no. 8, p. 650663, 1998.

[21] D. Perry, A. Porter, and L. Votta, "Empirical studies of software engineering: A roadmap," in *The Future of Software Engineering, Finkelstein A (ed.). ACM Press: New York NY*, 2000.

[22] J. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. Amor, and D. German, "Macro-level software evolution: a case study of a large software compilation," *Journal Empirical Software Engineering*, vol. 14, no. 3, pp. 262–285, 2009.

[23] M. Godfrey and Q. Tu, "Evolution in open source software: A case study," in *ICSM*, 2000, pp. 131–142.

[24] A. Capiluppi, J. Gonzlez-Barahona, I. Herraiz, and G. Robles, "Adapting the staged model for software evolution to free/libre/open source software," in *IWPSE '07*, 2007, pp. 79–82.

[25] D. Darcy, S. Daniel, and K. Stewart, "Exploring complexity in open source software: Evolutionary patterns, antecedents, and outcomes," in *HICSS '10*, 2010, pp. 1–11.

[26] G. Robles, J. M. Gonzalez-Barahona, M. Michlmayr, and J. Amor, "Mining large software compilations over time: Another perspective of software evolution," in *MSR '06*, 2006, pp. 3–9.

[27] K. Stefan, "Software evolution in open source projectsa large-scale investigation," *Journal of Software Maintainance and Evolution: Research and Practice*, vol. 19, pp. 361–382, 2007.

[28] T. Mens, J. Fernndez-Ramil, and S. Degrandsart, "The evolution of eclipse," in *International Conference on Software Maintenance (ICSM)*, 2008, pp. 386–395.

[29] A. Bauer and M. Pizka, "The contribution of free software to software evolution," in *Sixth International Workshop on Principles of Software Evolution*, 2003, pp. 170–179.

[30] A. Capiluppi, "Models for the evolution of os projects," in *ICSM '03*, 2003, pp. 65–74.

[31] S. McIntosh, B. Adams, and A. Hassan, "The evolution of ant build systems," in *MSR'10*, 2010, pp. 42–51.

[32] Y. Lee, J. Yang, and K. Chang, "Metrics and evolution in open source software," in *QSIC'07*, 2007, pp. 191–197.

[33] G. Xie, J. Chen, and I. Neamtiu, "Towards a better understanding of software evolution: An empirical study on open source software," in *ICSM'09*, 2009, pp. 51–60.

[34] S. Ali and O. Maqbool, "Monitoring software evolution using multiple types of changes," in *ICET'09*, 2009, pp. 410–415.

[35] A. Capiluppi and J. Ramil, "Studying the evolution of open source systems at different levels of granularity: Two case studies," in *IWPSE*, 2004, pp. 113–118.

[36] M. M. Simmons, P. Vercellone-Smith, and P. Laplante, "Understanding open source software through software archeology: The case of nethack," in *30th SEW*, 2006, pp. 47–58.

[37] K. Stewart, D. Darcy, and S. Daniel, "Observations on patterns of development in open source software projects," in *5th WOSSE*, 2005, pp. 1–5.

[38] A. Capiluppi and J. Ramil, "Studying the evolution of open source systems at different levels of granularity: Two case studies," in *IWPSE'04*, 2004, pp. 113–118.

[39] M. Simmons, P. Vercellone-Smith, and P. Laplante, "Understanding open source software through software archeology: The case of nethack," in *SEW '06*, 2006, pp. 47–58.

[40] A. Capiluppi and T. Knowles, “Software engineering in practice: Design and architectures of floss systems,” in *Open Source Ecosystems: Diverse Communities Interacting, IFIP Advances in Information and Communication Technology*, vol. 299/2009, 2009, pp. 34–46.

[41] A. Capiluppi and J. Ramil, “Change rate and complexity in software evolutions,” in *WESS’04*, 2004.

[42] R. Milev, S. Muegge, and M. Weiss, “Design evolution of an open source project using an improved modularity metric,” in *OSS’09*, 2009, pp. 20–33.

[43] W. Li and R. Shatnawi, “An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution,” *Journal of Systems and Software*, vol. 80, no. 7, pp. 1120–1128, 2007.

[44] K. Hayashi, K. Kishida, K. Nakakoji, Y. Nishinaka, B. Reeves, A. Takasbima, and Y. Yamamoto, “A case study of the evolution of jun: an object-oriented open-source 3d multimedia library,” in *ICSE’01*, 2001, pp. 524–533.

[45] B. Dagenais and M. Robillard, “Creating and evolving developer documentation: understanding the decisions of open source contributors,” in *FSE’10*, 2010, pp. 127–136.

[46] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. Hassan, “An empirical study on inconsistent changes to code clones at the release level,” in *WCRE ’09*, 2009, pp. 85–94.

[47] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto, “Is duplicate code more frequently modified than non-duplicate code in software evolution?: An empirical study on open source software,” in *IWPSE-EVOL ’10*, 2010, pp. 73–82.

[48] W. Jingwei, R. Holt, and A. Hassan, “Empirical evidence for soc dynamics in software evolution,” in *IEEE International Conference on Software Maintenance (ICSM 2007)*, 2007, pp. 244–254.

[49] I. Herraiz, J. Barahona, and G. Robles, “Determinism and evolution,” in *Proceedings of the 2008 international working conference on Mining software repositories (MSR’08)*, 2008, pp. 1–10.

[50] M. Weiss, G. Moroiu, and P. Zhao, “Evolution of open source communities,” *OSS*, vol. 203, pp. 21–32, 2006.

[51] A. Capiluppi, P.Lago, and M. Morisio, “Evidences in the evolution of os projects through changelog analysis,” in *Proceedings of the 3rd Workshop on Open Source Software Engineering (ICSE03)*, 2003, pp. 19–24.

[52] A. mockus, R. Fielding, and J. herbsleb, “Two case studies of open source software development: Apache and mozilla,” *ACM Trans. Software Engineering and Methodology*, vol. 11, no. 3, pp. 309–346, 2002.

[53] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye, “Evolution patterns of open-source software systems and communities,” in *IWPSE*, 2002, pp. 76–85.

[54] K. Ngamkajornwiwat, D. Zhang, A. Koru, L. Zhou, and R. Nolker, “An exploratory study on the evolution of oss developer communities,” in *HICSS*, 2008, p. 305.

[55] Q. Hong, S. Kim, S. Cheung, and C. Bird, “Understanding a developer social network and its evolution,” in *27th ICSM*, 2011, pp. 323–332.

[56] Q. Hong, S. Kim, S. Cheung, and C.Bird, “Understanding a developer social network and its evolution,” in *27th ICSM*, 2011, pp. 323–332.

[57] R. Chang, S. Yang, J. Moon, W. Oh, and A. Pinsonneault, “A social capital perspective of participant contribution in open source communities: The case of linux,” in *HICSS*, 2011, pp. 1–10.

[58] M. Weiss, G. Moroiu, and P. Zhao, “Evolution of open source communities,” *OSS*, vol. 203, pp. 21–32, 2006.

[59] A. Capiluppi, P. Lago, and M. Morisio, “Characteristics of open source projects,” in *CSMR ’03*, 2003, p. 317.

[60] F. Hunt and P. Johnson, “On the pareto distribution of sourceforge projects,” in *Proceedings of Open Source Software Development workshop*, 2002, pp. 122–129.

[61] Y. Yunwen and K. Kishida, “Toward an understanding of the motivation open source software developers,” in *ICSE ’03*, 2003, pp. 419–429.

[62] S. Shah, “Motivation, governance, and the viability of hybrid forms in open source software development,” in *Management Science*, vol. 52, pp. 1000–1014.

[63] J. Gutsche, “The evolution of open source communities,” *Topics in Economic Analysis and Policy*, vol. 5, no. 1, 2005.

[64] G. Robles, J. M. Gonzalez-Barahona, and M. Michlmayr, “Evolution of volunteer participation in libre software projects: Evidence from debian,” in *1st OSS*, 2005, pp. 100–107.

[65] Y. Wang, D. Guo, and H. Shi, “Measuring the evolution of open source software with their communities,” *ACM SIGSOFT Software Engineering Notes*, vol. 32, no. 6, pp. 1–7, 2007.

[66] M. Syeed, T. Kilamo, I. Hammouda, and T. Systa, “Open source prediction methods: a systematic literature review,” in *Proceedings of 8th. OSS, Springer*, 2012.

[67] S. Bendifallah and W. Scacchi, “Work structures and shifts: An empirical analysis of software specification teamwork,” in *11th ICSE*, 1989, p. 260270.

[68] N. Nagappan, B. Murphy, and V. Basili, “The influence of organizational structure on software quality: an empirical case study,” in *ICSE*, 2008, p. 521530.

[69] T. Browning, “Applying the design structure matrix to system decomposition and integration problems: a review and new directions,” in *IEEE Transactions on Engineering Management*, vol. 43, no. 3, 2001, p. 292306.

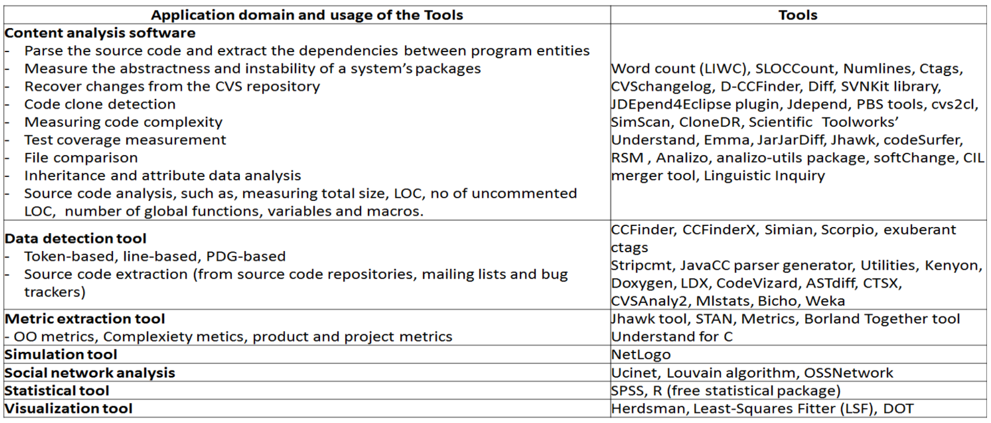
[70] H. Breivold, M. Chauhan, and M. Babar, “A systematic review of studies of open source software evolution,” in *APSEC*, 2010, pp. 356–365.

M.M. Mahbubul Syeed received his B.Sc degree in Computer Science and Information Technology from Islamic University of Technology, Bangladesh in September, 2002 and his M.Sc degree in Information Technology from Tampere University of Technology, Finland in April, 2010. He is currently working towards his Ph.D. degree and working as a researcher in the same university. His current research interest includes study of Open Source Software ecosystem.

Imed Hammouda is currently an associate professor at Tampere University of Technology (TUT) where he is heading the international masters programme at the Department of Pervasive Computing. He got his Ph.D. in software engineering from TUT in 2005. Dr. Hammouda’s research interests include open source software, software architecture, software development methods and tools, and variability management. He is leading TUTOpen - TUT research group on open source software. He has been the principal investigator of several research projects on various open initiatives. Dr. Hammouda’s publication record includes over fifty journal and conference papers.

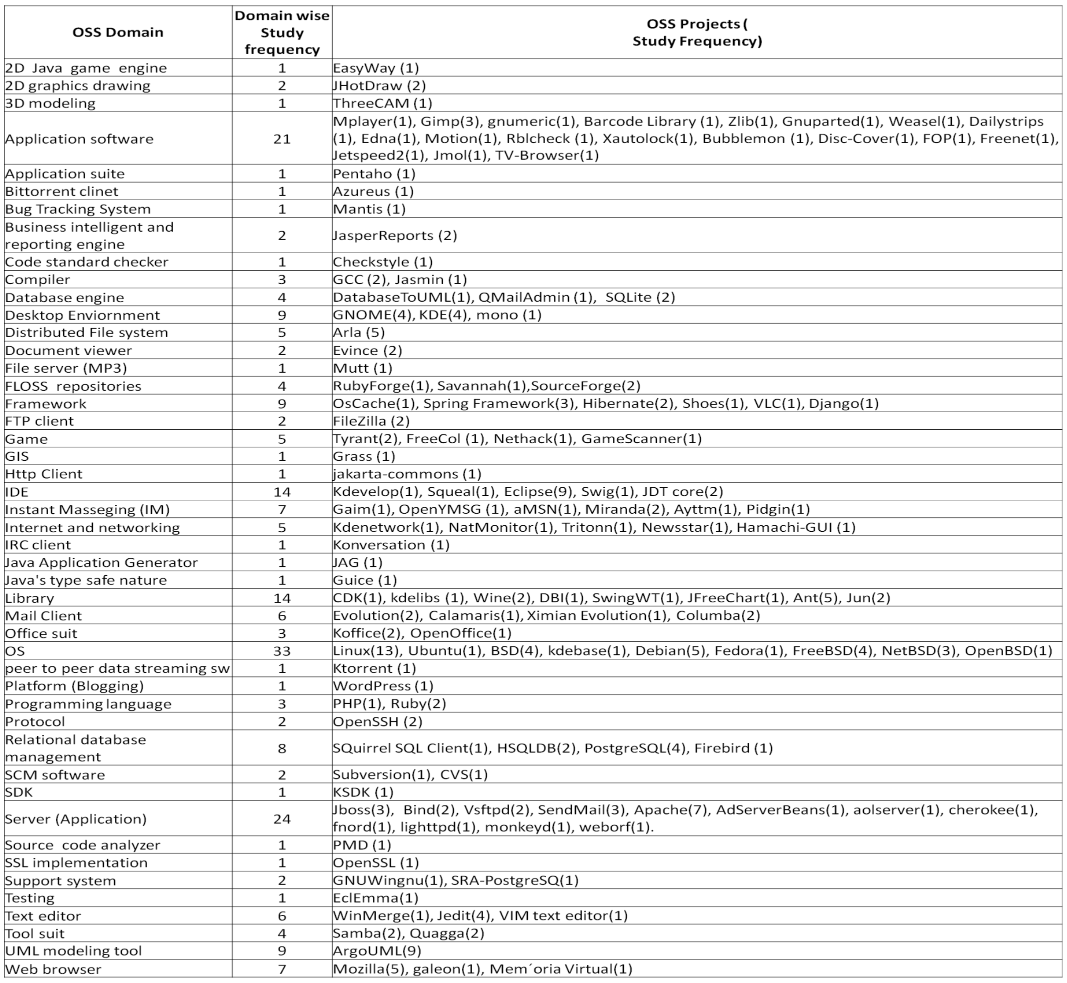
Tarja Systä is a professor at Tampere University of Technology, of Pervasive Computing Department. Her current research interests include software maintenance and analysis, software architectures, model-driven software development, and development and management of service-oriented systems.

APPENDIX



Application domain and usage of the Tools	Tools
Content analysis software - Parse the source code and extract the dependencies between program entities - Measure the abstractness and instability of a system's packages - Recover changes from the CVS repository - Code clone detection - Measuring code complexity - Test coverage measurement - File comparison - Inheritance and attribute data analysis - Source code analysis, such as, measuring total size, LOC, no of uncommented LOC, number of global functions, variables and macros.	Word count (LIWC), SLOCCount, Numlines, Ctags, CVSchangelog, D-CCFinder, Diff, SVNKit library, JDEpend4Eclipse plugin, Jdepend, PBS tools, cvs2cl, SimScan, CloneDR, Scientific Toolworks' Understand, Emma, JarJarDiff, Jhawk, codeSurfer, RSM , Analizo, analizo-utils package, softChange, CIL merger tool, Linguistic Inquiry
Data detection tool - Token-based, line-based, PDG-based - Source code extraction (from source code repositories, mailing lists and bug trackers)	CCFinder, CCFinderX, Simian, Scorpio, exuberant ctags Stripcmt, JavaCC parser generator, Utilities, Kenyon, Doxygen, LDX, CodeVizard, ASTdiff, CTSX, CVSAnaly2, Mlstats, Bicho, Weka
Metric extraction tool - OO metrics, Complexiety metics, product and project metrics	Jhawk tool, STAN, Metrics, Borland Together tool Understand for C
Simulation tool	NetLogo
Social network analysis	Ucinet, Louvain algorithm, OSSNetwork
Statistical tool	SPSS, R (free statistical package)
Visualization tool	Herdsman, Least-Squares Fitter (LSF), DOT

Figure 8. Tools (OSS and Proprietary) used for evolution studies



OSS Domain	Domain wise Study frequency	OSS Projects (Study Frequency)
2D Java game engine	1	EasyWay (1)
2D graphics drawing	2	JHotDraw (2)
3D modeling	1	ThreeCAM (1)
Application software	21	Mplayer(1), Gimp(3), gnumeric(1), Barcode Library (1), Zlib(1), Gnuparted(1), Weasel(1), Dailystrips (1), Edna(1), Motion(1), Rblcheck (1), Xautolock(1), Bubblemon (1), Disc-Cover(1), FOP(1), Freenet(1), Jetspeed2(1), Jmol(1), TV-Browser(1)
Application suite	1	Pentaho (1)
Bittorrent clinet	1	Azureus (1)
Bug Tracking System	1	Mantis (1)
Business intelligent and reporting engine	2	JasperReports (2)
Code standard checker	1	Checkstyle (1)
Compiler	3	GCC (2), Jasmin (1)
Database engine	4	DatabaseToUML(1), QMailAdmin (1), SQLite (2)
Desktop Enviornment	9	GNOME(4), KDE(4), mono (1)
Distributed File system	5	Arla (5)
Document viewer	2	Evince (2)
File server (MP3)	1	Mutt (1)
FLOSS repositories	4	RubyForge(1), Savannah(1),SourceForge(2)
Framework	9	OsCache(1), Spring Framework(3), Hibernate(2), Shoes(1), VLC(1), Django(1)
FTP client	2	FileZilla (2)
Game	5	Tyrant(2), FreeCol (1), Nethack(1), GameScanner(1)
GIS	1	Grass (1)
Http Client	1	jakarta-commons (1)
IDE	14	Kdevelop(1), Squeal(1), Eclipse(9), Swig(1), JDT core(2)
Instant Masseging (IM)	7	Gaim(1), OpenYMSG (1), aMSN(1), Miranda(2), Ayttm(1), Pidgin(1)
Internet and networking	5	Kdenetwork(1), NatMonitor(1), Tritonn(1), Newsstar(1), Hamachi-GUI (1)
IRC client	1	Konversation (1)
Java Application Generator	1	JAG (1)
Java's type safe nature	1	Guice (1)
Library	14	CDK(1), kdelibs (1), Wine(2), DBI(1), SwingWT(1), JFreeChart(1), Ant(5), Jun(2)
Mail Client	6	Evolution(2), Calamaris(1), Ximian Evolution(1), Columba(2)
Office suit	3	Koffice(2), OpenOffice(1)
OS	33	Linux(13), Ubuntu(1), BSD(4), kdebase(1), Debian(5), Fedora(1), FreeBSD(4), NetBSD(3), OpenBSD(1)
peer to peer data streaming sw	1	Ktorrent (1)
Platform (Blogging)	1	WordPress (1)
Programming language	3	PHP(1), Ruby(2)
Protocol	2	OpenSSH (2)
Relational database management	8	SQuirrel SQL Client(1), HSQLDB(2), PostgreSQL(4), Firebird (1)
SCM software	2	Subversion(1), CVS(1)
SDK	1	KSDK (1)
Server (Application)	24	Jboss(3), Bind(2), Vsftpd(2), SendMail(3), Apache(7), AdServerBeans(1), aolserver(1), cherokee(1), fnord(1), lighttpd(1), monkeyd(1), weborf(1).
Source code analyzer	1	PMD (1)
SSL implementation	1	OpenSSL (1)
Support system	2	GNUWingnu(1), SRA-PostgreSQ(1)
Testing	1	EclEmma(1)
Text editor	6	WinMerge(1), Jedit(4), VIM text editor(1)
Tool suit	4	Samba(2), Quagga(2)
UML modeling tool	9	ArgoUML(9)
Web browser	7	Mozilla(5), galeon(1), Mem´oria Virtual(1)

Figure 9. OSS Projects analyzed for evolution studies

[II] M.M. Syeed, T. Aaltonen, I. Hammouda, and T. Systä. Tool Assisted Analysis of Open Source Projects: A Multi-faceted Challenge. *International Journal of Open Source Software and Processes, vol. 3, no. 2, 2011*, pages 43–78, April-June, 2011.

Tool Assisted Analysis of Open Source Projects: A Multi-Faceted Challenge

M.M. Mahbubul Syeed, Tampere University of Technology, Finland

Timo Aaltonen, Nokia Research Center, Finland

Imed Hammouda, Tampere University of Technology, Finland

Tarja Systä, Tampere University of Technology, Finland

ABSTRACT

Open Source Software (OSS) is currently a widely adopted approach to developing and distributing software. OSS code adoption requires an understanding of the structure of the code base. For a deeper understanding of the maintenance, bug fixing and development activities, the structure of the developer community also needs to be understood, especially the relations between the code and community structures. This, in turn, is essential for the development and maintenance of software containing OSS code. This paper proposes a method and support tool for exploring the relations of the code base and community structures of OSS projects. The method and proposed tool, Binoculars, rely on generic and reusable query operations, formal definitions of which are given in the paper. The authors demonstrate the applicability of Binoculars with two examples. The authors analyze a well-known and active open source project, FFMpeg, and the open source version of the IaaS cloud computing project Eucalyptus.

Keywords: Computer Science, Open Source Software (OSS), Reverse Engineering, Social Network Analysis, Software Repositories, Tool Support

INTRODUCTION

Open Source Software (OSS) is currently a widely adopted approach to developing and distributing software. Successful open source projects are typically complex, both from the point of view of the code base and with respect to the developer and user community. Such a project may consist of a wide range of components, coming with a large number of versions reflecting their development and evolution history. While it is a challenge to acquire knowledge from developers and users due to their distributed nature, open source development and user communities often produce a rich software repository as a byproduct. In addition to source code and other software artifacts, there are repositories containing other sources of information such as bug reports, mailing lists, and revision history logs.

DOI: 10.4018/jossp.2011040103

A variety of tools and techniques have been proposed to study open source projects. However, most of these techniques suffer from fragmentation and lack of synergies. For instance, many tools apply reverse engineering techniques to study the software side of OSS (Knab, Pinzger, & Bernstein, 2006; Zhou & Davis, 2005). Other works have considered social network analysis techniques to study the social model of OSS (Martinez-Romo, Robles, Ortuo-Perez, & Gonzalez-Barahona, 2008; Kamei, Matsumoto, Maeshima, Onishi, Ohira, & Matsumoto, 2008).

We argue that the separation between the two is artificial. The dimensions are complementary, not discrete categories. They can be used together to provide a lot of useful information. To make a decision about using OSS as a part of a software product to be developed, it is essential to be able to understand the role of the developer community in the development and maintenance of different parts of the OSS code base. This would also help in estimating and planning the future development and maintenance activities of the software product. For instance, understanding the relations of the code and developer community structures helps in understanding where the expertise lay within the developer community, which in turn helps in deciding who to contact concerning issues related to a specific part of the OSS code base.

In this paper we address such challenges as follows: first, a compact and extensible metamodel is proposed which captures both the code base and the community dimension of OSS projects; second, a set of reusable formal descriptions of operations are given, which allows the relationships between the code structure and the community structure to be queried and third, a tool, Binoculars, corresponding to both the metamodel and the defined operations is implemented to study the feasibility of our approach. The tool, Binoculars is able to (a) merge a community view with source code views; (b) provide different perspectives to view the data presented in the form of a graph. This feature can be used, for example, to identify and study the groups of developers working with the same (or related) code fragments, or to study whether communication structures of the developing community conform to the architecture of the software, or trace out the relationship between the developer and the user community in the context of the codebase; (c) render the graph information at different levels of abstraction; (d) provide query support for the graphs.

We also demonstrate the applicability of our approach and the tool with two examples. First, we analyze a well known and active open source project FFMpeg (FFmpeg, 2010) and show how to make queries essential from the point of view of development and maintenance of software relying on FFMpeg code. Then we analyze the open source version of the Eucalyptus project (Eucalyptus, 2011) and its community, due to its emerging impact in the field of cloud computing.

The paper is structured as follows. In Section TOWARDS A GENERIC OSS ANALYSIS TOOL we propose our approach. The tool, Binoculars, is described in Section TOOL SUPPORT. The applicability of Binoculars is discussed in Section CASE STUDY. We review known approaches and techniques for analyzing OSS projects in Section RELATED WORK and distinguish our work from that of existing ones. Finally, some concluding remarks and future development of the work are presented in Section DISCUSSION.

TOWARDS A GENERIC OSS ANALYSIS TOOL

Comprehension of a software project can be addressed at different abstraction levels (Rosso, 2006). At the lowest level, the file system (code base) can be analyzed and refactored by employing reverse engineering techniques to extract code dependency and the architecture of the system. At the next highest level of abstraction, the community structure and people's (developers and users) activities, communication, and social interaction can be analyzed. It should be noticed that dependency exists at each abstraction level as well as between the levels. For example, code level dependency arises due to interface dependency, inclusion, code dependency and so on. Also at the community level, interpersonal communication leads to social dependencies among people in the organization. Interlevel dependency arises due to the fact that developers work on the project artifacts, fixing reported bugs, incorporating new features whereas users often posts bug reports, asking for new features and so on.

Social network analysis (SNA) (Rosso, 2009) in this regard can be used to capture and effectively model these dependencies through graphs, where entities (e.g., developers, users, code files, bugs) are presented as vertices and the dependencies among the entities are modeled as edges (or arcs). For example, social networks can be dug from the model based on the persons working on the same files (or issues) or we can study who are working with the same bug reports. Different measures and metrics in SNA can be applied on these relationship graphs to get an insight of the project. For instance, metrics like density, centrality, closeness, betweenness (Carrington, Scott, & Wasserman, 2005) can be applied on the resultant graphs.

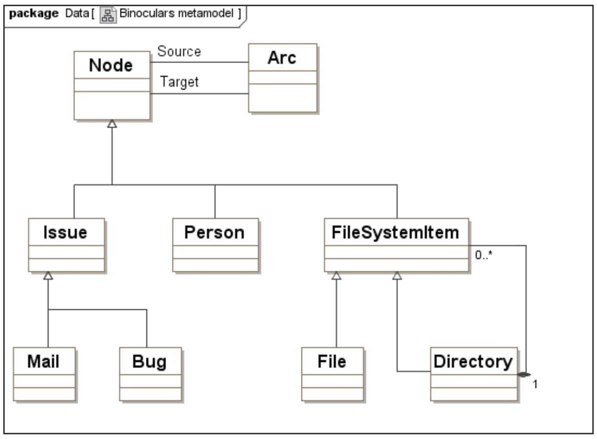
We approach the goal of building a generic and customizable OSS analysis tool by starting from a relatively simple metamodel that can later be extended by the concepts needed to support the different data sources and purposes (see e.g., Table 2 in Section RELATED WORK). This is essential, since OSS projects like any software projects, vary significantly. We believe that a small metamodel that is widely applicable through extensions is more useful than a comprehensive and detailed one, the use of which can be either limited or unnecessarily cumbersome.

The Binoculars metamodel is divided into domain independent and domain specific models. In the former, everything is modeled as mathematical graphs, whereas the latter introduces the open source related aspects. The domain independent model consists of classes Node and Arc, which are depicted in the upper part of Figure 1.

The domain specific model introduces concepts for modeling an open source project. The abstract class FileSystemItem (on the right hand side), whose concrete subclasses are File and Directory, is used for modeling the actual implementation. Introducing the composition relationship between Directory and FileSystemItem allows us to model the real file system of the implementation. Objects of the class Person (in the middle) form the open source community.

Finally, Issues (Mails and Bug reports) model the communication that takes place during development.

Figure 1. The Binoculars metamodel



Due to the fact that these are all subclasses of the abstract class Node, they can have links (instance of Arc) with any other types of Nodes. This allows us to mine, combine and in particular refine the information available for the subject project. The domain specific model could be more detailed and it could include more aspects. For example, in many reverse engineering activities the actual implementation is modeled to the level of (member) functions, or even programming-language-level statements. Moreover, the class Issue has several obvious derivations, like chat messages. Open source development is often based on patches delivered by the developers. The model could be augmented with the class Patch. Then the class File would have a composition relation with the class Patch. For the purposes of this paper we have selected a high level of abstraction for our demonstration purposes. Finally, the other aspect of OSS projects, i.e., licensing, may introduce new node types to be added to the metamodel.

In order to instantiate the model we utilize all available sources of information, like the code base, mailing lists, bug tracking tools and meta information from the project's www pages. The sources are the ones that fix the domain specific model. In Binoculars each of these sources are represented as repositories, namely person, file and issue, corresponding to the three subclasses of Node and their substructures, depicted in Figure 1. The Binoculars Metamodel. Extracted information is stored in these repositories and are interpreted and presented as graphs with nodes and edges. An edge may have a weight, showing the strength of a relationship. Also, an edge may be directed depending on the relationship. In such cases we denote edges as arcs.

We define a set of operations to operate on repositories for generating relationship graphs. The operations can also be performed on the generated graphs to derive other relationship graphs. To present these operations, we first define the repository structure, and then discuss the syntax of the operations which is followed by their description with illustrative examples. The formal definition of these operations with detailed discussion is provided in the appendix.

Repository Structure

Let, $\rho = \{R^1, R^2, \cdots, R^n\}$ be the set of repositories. Where each repository, $R^i = \{r_1^i, r_2^i, \cdots, r_m^i\}$ consists of repository elements r_j^i. Each repository element $r_j^i = \{c_{j1}^i, c_{j2}^i, \cdots, c_{jx}^i\}$ consists of attributes c_{jk}^i describing it. Each attribute element $c_{jk}^i = \{c_{jkp}^i\}$ consists of a number of attribute values c_{jkp}^i as shown in Figure 2.

For example, consider a set of repositories, $\rho = \{File^F, Person^P, Issue^I\}$ where the $File^F$ repository contains a description of the code files, the $Person^P$ repository contains detail of each person (either user or developer) involved in the project and $Issue^I$ repository contains information about the issues (e.g., reported bugs, feature requests) raised by the project personnel. For this illustration consider only the $File^F$ repository. The repository $File^F = \{file_1^F, file_2^F, \cdots, file_m^F\}$ consists of code files as repository elements. Each code file,

$$File_j^F = \{f_{name\,j1}^F,\ path_{j2}^F,\ extension_{j3}^F,\ developer_{j4}^F,\ includes_{j5}^F,\ copyright_{j6}^F\}$$

consists of six attributes referring to the name of the file, its path, extension, a list of the names of developers, included files and copyright information. This repository structure is shown in Figure 3.

Graph Operations

In this section the rationale and an abstract level discussion of the graph operations are presented. The syntax of these operations is presented in Figure 4. The syntax follows *Larch-like* notations (Guttag & Horning, 1993). As shown in Figure 4, we have four sorts, namely *repository*, *attributeName*, *graph* and *vertexSet* based on which graph operations are defined. Sort *repository* contains repository elements of a certain type, sort *attributeName* is the name of an attribute in a repository, sort *graph* represents a relationship graph, and finally sort *vertexSet* is a set of vertices in a given graph.

Operations are categorized in *repository operations* and *graph operations* (Figure 4). Repository operations are used to explore and combine information originating from different sources. Since we aim at an approach and a tool that can support studying both code and community dimensions of OSS projects, we need a way to show relationships between elements in two distinct repositories. So, we use the concept of affiliation networks (Rosso, 2009) for showing relationships between two sets of components (or vertices). In an affiliation network one set is called "actors" and the other is called "events". Thus an affiliation network is called a "2-mode network" and represented as a bipartite graph. In the context of Binoculars, actors denote attribute values, and events represent the repository elements. The other types of operations, graph operations, are used to support more detailed analysis of the information included in the graphs.

Repository operations are applied directly to repositories. We have one such operation, called the *interConnect* operation, which accepts a repository and an attribute name as argument. For each repository element the attribute contains one or more attribute values. The *interConnect* operation creates relations between repository elements and each of the attribute values, thus generating an affiliation network. For example, consider the *File* repository presented in Figure 3. Each code file in this repository has an attribute called developer that consists of a list of developer names who contributed to that file. Thus, it is quite natural to have a relationship between that code file and it's developers. Applying *interConnect* operation in this case would generate a bipartite graph (Figure 5), which is a 2-mode affiliation network showing the relationship between two distinct sets of entities, as code files and their developers. Thus, one can easily study the relationship between two components (actors and events) in a software project through such graphs. A formal definition of the *interConnect* operation is given in the appendix, section The interConnect operation.

The second category of operation is the graph operation. There are five operations belonging to this category, namely *intraConnect* operation and four set operations (Figure 4). The *intraConnect* operation is used to transform the 2-mode networks (generated by the *interConnect* operation) into two 1-mode networks. To do so, this operation accepts a graph generated by the *interConnect* operation and one of the vertex sets (either actors or events) in the input graph. Based on the vertex set, this operation generates relationships among the vertices in the other vertex set. Thus it is possible to investigate relationship from the perspective of the actors or the events. Let us consider the example graph above (Figure 5), which shows relationships between code files (events) and developers (actors). Application of the *intraConnect* operation on this graph could generate either a graph showing the relationship between developers (actors) based on common code file sharing (Figure 6) or a graph showing the relationship between code files (events) based on common developer involvement (Figure 7). Edge weights in these graphs show the relationship strength between pairs of nodes. For example, the edge weights in the graph shown in Figure 6 reveal how many code files are shared among the developers.

Here, the actors graph depicts collaboration among the members (developers) within the community based on common interest and responsibilities (e.g., code file sharing). On the other hand, the events graph can be used to understand the degree of *interlocks* (Rosso, 2009) between events as created by actors. For example, a developer working on two code files simultaneously creates an interlock between them. An interlock between the files thus represents the working area of the person involved as well as revealing dependency among the code files evolved due to common developer interest. If multiple developers are involved, then the interlock gets even stronger. This information would be useful for a new developer working on some part of the project. For

Figure 2. Graphical representation of the repository model

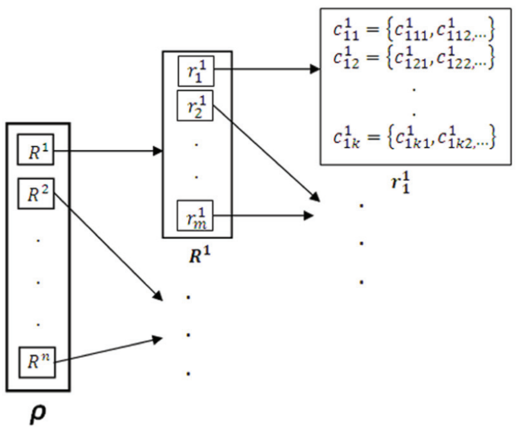
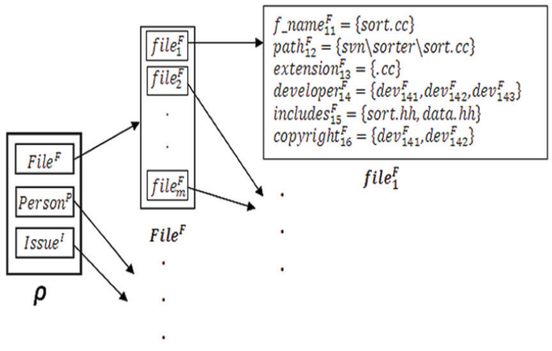

$c_{11}^{1} = \{c_{111}^{1}, c_{112}^{1}, \ldots\}$
$c_{12}^{1} = \{c_{121}^{1}, c_{122}^{1}, \ldots\}$
$c_{1k}^{1} = \{c_{1k1}^{1}, c_{1k2}^{1}, \ldots\}$
r_1^1
r_2^1
r_m^1
R^1
R^2
R^n
ρ

Figure 3. An example of the file repository

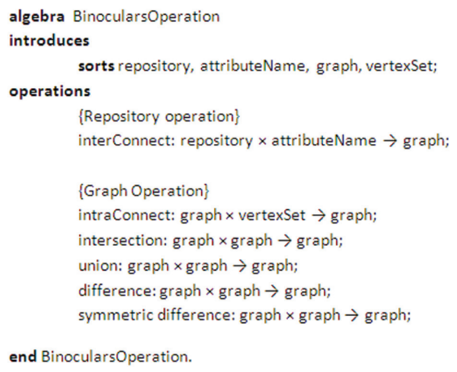

$f\_name_{11}^{F} = \{sort.cc\}$
$path_{12}^{F} = \{svn\backslash sorter\backslash sort.cc\}$
$extension_{13}^{F} = \{.cc\}$
$developer_{14}^{F} = \{dev_{141}^{F}, dev_{142}^{F}, dev_{143}^{F}\}$
$includes_{15}^{F} = \{sort.hh, data.hh\}$
$copyright_{16}^{F} = \{dev_{141}^{F}, dev_{142}^{F}\}$
$file_1^F$
$file_2^F$
$file_m^F$
$File^F$
$Person^P$
$Issue^I$
ρ

example, he can find out which developers previously worked on a certain part of the code base or what other files are significantly manipulated along with the file in question. The formal definition and weight calculation of the *intraConnect* operation are discussed in the appendix, section The intraConnect operation.

Set operations are commonly applied to graphs. Figure 4 lists four such set operations, namely intersection, union, difference and symmetric difference. These set operations have several implications for analyzing actors or events graphs, which include analyzing a community from multidimensional perspectives, identifying groups and their interrelationship based on a certain criterion. For example, one might wish to identify the group of people who work as developers as well as bug fixers in the community, or what are the groups in the developer community according to their responsibilities and how these groups are related to each other, or does the actual code file dependency match the dependency evolved due to common developer interest (an application of Conways law (Conway, 1968)). Next, these set operations are presented with an example from the perspective of the developer community.

Consider the two actor graphs shown in Figure 8. The graph in Figure 8(a) shows the relationship among the developers due to common code base sharing and the graph in Figure 8(b) shows the developer relationship in the com-

Figure 4. List of operations implemented in Binoculars

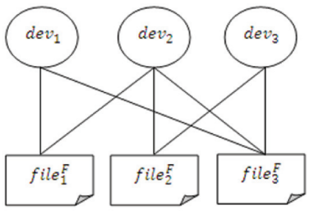


```
algebra BinocularsOperation
introduces
        sorts repository, attributeName, graph, vertexSet;
operations
        {Repository operation}
        interConnect: repository × attributeName → graph;

        {Graph Operation}
        intraConnect: graph × vertexSet → graph;
        intersection: graph × graph → graph;
        union: graph × graph → graph;
        difference: graph × graph → graph;
        symmetric difference: graph × graph → graph;

end BinocularsOperation.
```

Figure 5. 2-mode network showing relationship between developers and the code files



munity due to common bug solving. The edge weight in both the graphs shows how many code files or bugs are shared among the developers.

The application of the intersection operation keeps the common section of the input graphs, showing the region having multidimensional exposure. For example, one might be interested in identifying the community of developers who work both as a developer as well as an issue or bug solver in an OSS project. An edge weight in the resultant graph is calculated by adding the edge weights between the corresponding actor nodes in the input graphs. The graph shown in Figure 9 is such a graph generated by applying the intersection operation on the graphs in Figure 8. This graph shows the developer community who contribute to the code base as well as in bug solving. Formal definition and weight calculation of the intersection operation is also given in the appendix, section Set operations.

Union operation retains both the input graphs, thus showing at most three regions, one that is common to both the input graphs (if any) and the two that are present in either of the input graphs. Hence this operation can be used to exhibit the entire community with their focus domains of activity in the project. For example, the graph shown in Figure 10 is generated by applying union operation on the graphs in Figure 8. In this graph, the relation shown with a thick solid line is involved in both code base developments as well as in bug solving, but the relations shown with the thin solid line and thin dashed lines are the relations involved in code development and bug solving, respectively. A formal definition of union operation with the weight calculation procedure is presented in the appendix, section Set operations.

Difference and Symmetric difference operations are only used for distinct regions in the input graphs, excluding the common region. Thus, these operations are most suitable for

Figure 6. 1-mode network showing relationship among developers based on common code file sharing

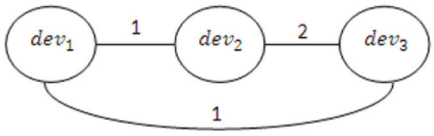
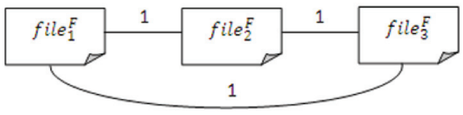


Figure 7. 1-mode network showing relationship among code files based on common developer involvement



finding out about specialized regions in a community. For example, one might be interested in finding out the group of people in the developer community working only as bug fixers or as developers. Figure 11 shows the graph generated by applying difference operation on the graphs shown in Figure 8. This graph shows the part of the developer community that only takes part in bug fixing. The formal definition and weight calculation procedure for difference and symmetric difference operations are given in the appendix, section Set operations.

TOOL SUPPORT

Architecture

Binoculars supports visualization of information through graphical representation. Emphasis was given to extracting relevant information, deriving relationships information which is represented and rendered through graphs. Figure 12 shows the architecture of Binoculars.

Binoculars architecture is based on the Eclipse RCP (Rich Client Platform) (Eclipse, 2010) and MVC (Model-View-Controller) architecture. RCP provides a flexible means to implement desktop applications either as plugins or as standalone applications. One of the main characteristics of MVC architecture is its ability to separate the business logic and application data from its presentation. In this architecture, the model stores data and during the course of editing, undo, and redo, the model is the only thing that endures. Thus all operations are applied to the model data for manipulation. Whereas Views are the way to present the model data in different ways. Whenever the underlying model changes, it would be updated and reflected in the corresponding views. And the interaction and synchronization between the model and its view is maintained by the Controller. Thus a controller is responsible for intercepting the requests from view and passing it to the model for appropriate action, the result of which is then passed back to the view for necessary updates.

In Binoculars, models are implemented in the ContentProviders module, views are implemented and maintained in the ViewGenerator module, and the controller is implemented in the Commands module as shown in Figure 12.

The ContentProviders models the graph information that is generated and supplied by the CreateEdge module. The CreateEdge module digs up the repository information for generating graph data. The underlying graph generating methods (i.e., operations that are defined in section Graph Operations and in the appendix) are implemented in this module. Graph informa-

Figure 8. (a) Relationship among developers based on common code file sharing (b) Relationship among developers due to common bug solving

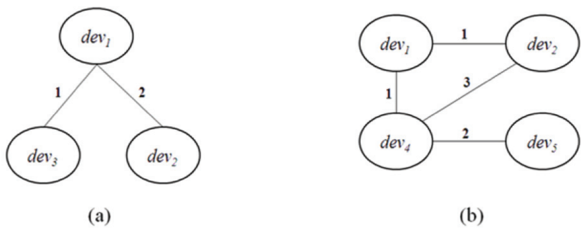
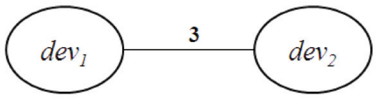


Figure 9. Intersection operation applied on graphs shown in Figure 8



tion is generated as XML data which contains the edge list, edge weight, details of the edge weight and associated node information. To help support mining the repositories and to generate XML data, the xmlParser module is used. This module uses the DOM XML parser (W3 Schools, 2010) to extract as well as to create new XML data for the graphs. The Repository module is the one which implements the metamodel presented in section TOWARDS A GENERIC OSS ANALYSIS TOOL and parses required information from different data sources.

The ViewGenerator module has two component modules, Views and ViewSorter. The Views module represents the data provided by the ContentProviders. Views can present model data in different ways, such as tabular, chart, tree and graphs. The ViewSorter module is utilized by the Views to organize its content when displaying. It is possible to open multiple instances of a view with customized data obtained from the same ContentProvider. This kind of customized visualization is handled by the Utility module.

There are many ways to customize a graph and to generate a new one based on user settings. For example, a graph can be customized by (a) selecting specific nodes or edges of the graph, (b) setting a weight range for the edges, (c) selecting attribute values for nodes, and (d) n-level nearest neighbor of a selected node. All these customizations of a graph are carried out by the commands implemented in the Commands module (in the middle of Figure 12). Required interfaces and a validation mechanism for the user inputs are implemented in the Dialogs module. Both Commands and ViewGenerator modules use these interfaces for user interaction. Appropriate command accepts the verified user data and passes it to the ContentProvider. The ContentProvider module updates the model and returns it to the corresponding views. The views are then updated accordingly.

The module Binoculars in Figure 12 contains the basic components that are required to run an RCP application. The most important components are the RCP main application class which implements the interface IApplication, one Perspective which holds menus and three place holders called folders, and a Workbench Advisor which controls the appearance of the application (menus, toolbars, perspectives, etc).

Figure 10. Union operation applied on graphs shown in Figure 8

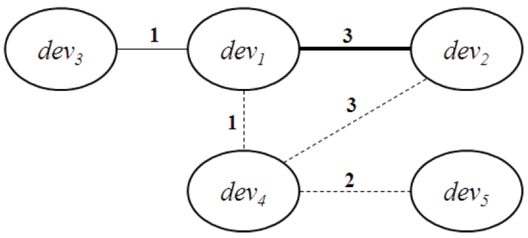


Figure 11. Difference operation applied on graphs shown in Figure 8

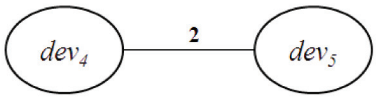
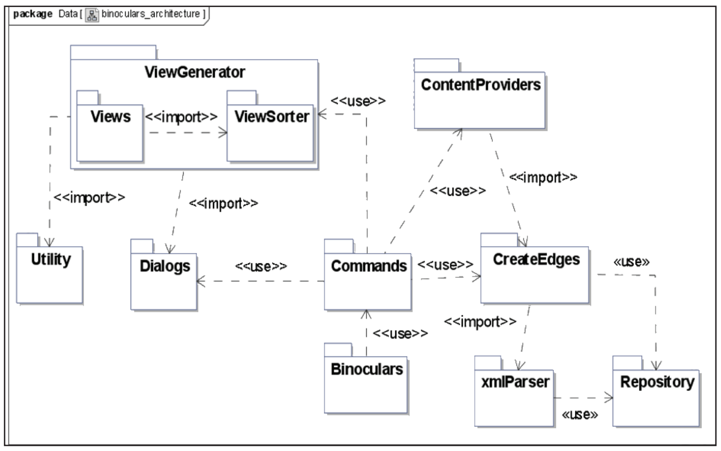


Figure 12. Overall architecture of Binoculars



Libraries and Platform Used

In what follows, the platform and libraries used to implement Binoculars are introduced.

- Eclipse SDK: Eclipse SDK (http://www.eclipse.org) is an open source, multi-language software development environment comprising an IDE and a plug-in system to extend it. Eclipse is used for developing Binoculars.
- The Java SE (version 6) Runtime Environment is used for developing the tool (Oracle, 2011).
- Eclipse RCP (Rich Client Platform): Eclipse RCP (Eclipse, 2010) allows the Eclipse platform to be used to create flexible and extensible desktop applications. Eclipse is built upon a plugin architecture, where a plugin is defined as the smallest deployable and installable software component of Eclipse. This architecture allows developers to use

existing plugins, other third party extensions and own custom-built ones.
- Zest: ZEST is a graph visualization Toolkit built as an Eclipse plug-in. It provides a convenient set of API's for rendering and manipulating graphs (http://www.eclipse.org).
- JFree chart: JFreeChart (Gilbert & Morgner, 2010) is a Java chart library with a set of API's supporting a wide range of chart types. JFreeChart is open source and distributed under the terms of LGPL (GNU Lesser General Public License). To use JFreeChart, the libraries (.jar) of JFreeChart need to be added to the Java classpath.
- Dom XML parser: The XML DOM (W3 Schools, 2010) defines a standard way for accessing and manipulating XML documents. It provides a convenient set of API's in Java for creating and manipulating XML documents.
- SWT (Standard Widget Toolkit): SWT is an open source widget toolkit available as an Eclipse plug-in and used to design efficient, portable user-interfaces in Java.

Features

The features of Binoculars are implemented to support the analysis of open source projects from the perspective of both community and code base. Figure 13 shows the main interface of Binoculars.

As shown in Figure 13, Binoculars has three panels (left, top and bottom), each of which is used to hold and display multiple views. Views are used to hold and display different representations of the data upon request. Depending on the purposes, Binoculars has six views. They are, project views, tabular graph data view, graph view, tabular chart data view, chart view and project data view.

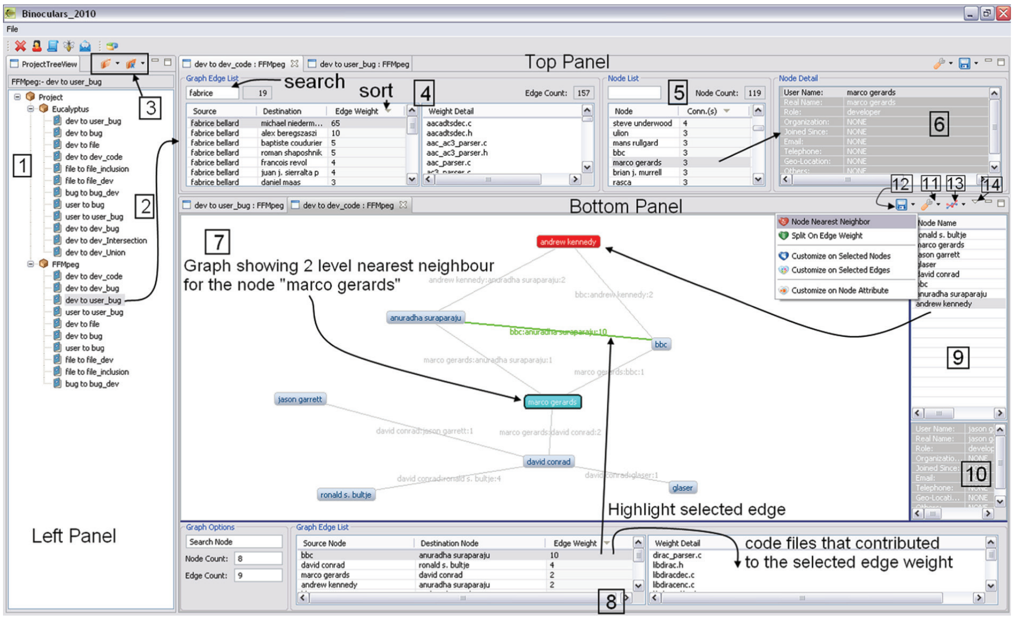
The left panel is used to hold and display the project view, the top panel is used to hold tabular graph data view, project data view and tabular chart data view, and the bottom view holds and displays the graph view and the chart view. What follows describes the functionalities of each of these views and their intercommunication.

***Project View*:** This view lists the projects and the graphs that are generated under each project. This list is organized in a tree structure as shown in Figure 13, item 1. Clicking on a graph name under a project, would display the tabular representation of the graph in the top panel (Figure 13, item 2). Project view also has menu level commands (Figure 13, item 3) that allows users (a) to add new projects and graphs under existing projects and (b) to remove existing graphs and projects. The list of graphs that can be generated under a project are listed in Table 1. Each graph in this table is described by its unique name, purpose and graph operation that is applied to generate the graph.

***Graph Data View [Tabular]*:** This view displays graphs as a list of edges and nodes. It contains the following, (a) An edge list table consisting of graph edges. Each edge is described by source node, destination node, edge weight and weight detail. (b) A node list table which lists all nodes in the graph with the number of connections for each node. (c) A node description table which shows information about a selected node (Figure 13, items 4, 5, 6 respectively). Table data can be searched and sorted based on a selected column. For example, the edge list table (Figure 13, item 4) is sorted in descending order based on edge weight and a search is made with the word *fabrice*.

***Graph View*:** This view displays graphs in a graphical way with detailed graph information. This view consists of the following components, (a) a graph display pane which displays the graphs consisting of nodes and edges. Edges might have weights depending on the graph; (b) an edge detail

Figure 13. Binoculars user interface



Graph Edge List: fabrice — 19

Source	Destination	Edge Weight
fabrice bellard	michael niederm...	65
fabrice bellard	alex beregszaszi	10
fabrice bellard	baptiste coudurier	5
fabrice bellard	roman shaposhnik	5
fabrice bellard	francois revol	4
fabrice bellard	juan j. sierralta p	4
fabrice bellard	daniel maas	3

Weight Detail
aacadtsdec.c
aacadtsdec.h
aac_ac3_parser.c
aac_ac3_parser.h
aac_parser.c

Edge Count: 157

Node List — Node Count: 119

Node	Conn.(s)
steve underwood	4
ulion	3
mans rullgard	3
bbc	3
marco gerards	3
brian j. murrell	3
rasca	3

Node Detail

User Name:	marco gerards
Real Name:	marco gerards
Role:	developer
Organization:	NONE
Joined Since:	NONE
Email:	NONE
Telephone:	NONE
Geo-Location:	NONE
Others:	NONE

Node Name
onald s. bultje
narco gerards
ason garrett
glaser
david conrad
bbc
anuradha suraparaju
andrew kennedy

User Name:	jason g
Real Name:	jason g
Role:	develop
Organizatio...	NONE
Joined Since:	NONE
Email:	NONE
Telephone:	NONE
Geo-Locati...	NONE
Others:	NONE

Graph Options
Search Node
Node Count: 8
Edge Count: 9

Graph Edge List

Source Node	Destination Node	Edge Weight
bbc	anuradha suraparaju	10
david conrad	ronald s. bultje	4
marco gerards	david conrad	2
andrew kennedy	anuradha suraparaju	2

Weight Detail
dirac_parser.c
libdirac.h
libdiracdec.c
libdiracenc.c

pane which consists of an edge list table, a weight detail table and a graph option table. The weight detail table displays weight detail when an edge in the graph is selected and the graph option provides the graph summary data and searching facility; (c) a node list pane which lists all the nodes of the displayed graph; and (d) a node detail pane which shows information about a node selected from the displayed graph. All these components are shown in Figure 13, items 7, 8, 9, 10 respectively.

Both graph data view and graph view consist of a comprehensive set of menu level options to customize graph data and its visualization. These options are discussed bellow.

***Graph customization*:** Displayed graphs can be customized in five different ways (Figure 13, item 11). These options are as follows, (a) node nearest neighbor: *n* level nearest neighbor of a selected graph node can be viewed. For a given node this option shows all the nodes that are nearest to it by the selected level. The graph in Figure 13 shows the 2 level nearest neighbor for the selected node *marco gerards*; (b) split on edge weight: a graph can be customized based on a given range of edge weights. The edges satisfying the weight range are displayed; (c) customize on selected nodes: a list of nodes from the displayed graph can be selected to draw a customized graph with those nodes, provided they have relations in the original graph; (d) customize on selected edges: a customized graph can be constructed by selecting a list of edges from the original graph; (e) customize on node attributes: this option gives the opportunity to customize a graph based on a given attribute value. For example, the graph in Figure 14 shows those developers and their relationships in the Eucalyptus community who live in the United States of America. All the component panes in the graph view are updated according to the customization.

***Save graph*:** The original graph or it's customized version can be saved in XML files

(Figure 13, item 12). It is also possible to save the graph nodes and their associated detail in XML files.

***Layout Change and zooming*:** Four layout options (namely, spring, grid, tree, radial) are provided for visualizing graphs (Figure 13, item 13). The displayed graph can also be zoomed between 50% and 400% (Figure 13, item 14).

***Chart View*:** Charts are used to show summary data of the project. Two charts are currently available, a pie chart and a bar chart. Charts can be generated based on selected attributes. For example, Figure 15 shows a pie chart revealing bugzilla distribution in the Eucalyptus project based on their priority level.

Apart from these, Binoculars has three more views for visualizing the project repositories. These are the project personnel view which shows project personnel information under their respective projects. Similarly project code base view and project bugzilla view respectively provide codebase information and an issue history log of a project.

Feasibility and Scalability of Binoculars

Binoculars requires repository information to create a graph. Repository information is collected from projects data management systems as discussed in section Data Collection. Customized parsers are built to fully automate this data collection process. But due to the diverse nature of data management systems of OSS projects, it is really hard to have a generalized parser for all the projects. Thus parsers are not the part of Binoculars.

Both the repository information and the created graphs are stored in XML files as discussed in section Architecture. For creating a new graph, Binoculars extract information from the related repositories and create and store the graph in a XML file. For rendering an existing graph, it simply uses the corresponding graph XML file. Thus creating and rendering a graph requires normal processing time.

One can extend an existing repository by simply adding information in the XML files with defined tags. Similarly, a new repository can be added using the existing XML template. Again, Binoculars operates on the repository information, thus it is not dependent on the programming language used in a project.

CASE STUDY

In this section, we demonstrate the use and applicability of our approach and the tool Binoculars by analyzing the selected open source projects, FFMpeg (2010) and Eucalyptus (2011).

Short Description of FFMpeg

FFmpeg is a complete, cross-platform solution to record, convert and stream audio and video in numerous formats (FFmpeg, 2010). There are more than 140 projects listed in the FFMpeg official website which use programs from the FFMpeg project (FFmpeg, 2010).

The name of the project comes from the MPEG video standards group, together with "FF" for "fast forward" (Bellard, 2006). Fabrice Bellard is the originator of this project and FFmpeg is his trademark.

FFmpeg is written in C and developed under Linux. It is free software and is licensed under the GNU Lesser General Public License (LGPL) version 2.1 (FFmpeg, 2010) or later. FFmpeg also incorporates several optional parts and optimizations that are covered by the GNU General Public License (GPL) version 2 or later.

In the FFmpeg project, information is mainly maintained in a version control system, a bug reporting system and a registered user information system. These are the basic sources of information for the analysis.

Short Description of Eucalyptus

Eucalyptus is a software platform which implements scalable IaaS (Infrastructure as a Service)

Table 1. Graph visualization supported by binoculars

Graph Name	Description
dev to file	This graph shows the relationships between code files and the developers who are responsible for those code files. Thus, this graph reveals how responsibilities are distributed over the developer community. To draw this graph, an $interConnect(File^F, developer)$ operation is performed. A detailed description of this operation is presented in the appendix, The interConnect operation.
Dev to issue	This graph shows relationships between developers and user issues answered by the developers. From this graph, one can identify which developers are engaged in solving user issues and to what extent. To draw this graph, an $interConnect(Issue^I, developer)$ operation is performed.
User to issue	This graph shows relationships between issues and the users who posted those issues. From this graph one can easily figure out the interest areas of users and their contributions, as subject areas of the issues should be related to the purpose of use. To draw this graph, an $interConnect(Issue^I, user)$ operation is performed.
file to file_inclusion	This graph shows relationships between code files of the project based on the inclusion structure. Thus it can reveal how code files are dependent on each other. To draw this graph, an $interConnect(File^F, inclFile)$ operation is performed.
dev to dev_code	This graph shows relationships between developers based on a common code file use. The edge weight between two developer nodes specifies how many code files they share in common. To draw this graph an $intraConnect(G_{File-Developer}, File)$ operation is performed. A detail description of this operation is presented in the appendix, section The intraConnect operation.
dev to dev_issue	This graph shows how developers are interacting with each other when working on issues. A relationship exists between two developers if they worked on a common issue. The edge weight between two nodes shows how many common issues exists between two developers. To draw this graph an $intraConnect(G_{Issue-Developer}, Issue)$ operation is performed.
user to user_issue	This graph shows relationships between users based on a common issue. Generally issues (such as bug reporting, feature request) are posted by users and it is quite common that many users have the same issue in common. Thus, there are relationships between them. The edge weight in this graph again shows the multiplicity of such occurrences. To draw this graph an $intraConnect(G_{Issue-User}, Issue)$ operation is performed.
file to file_dev	This graph shows relationships between code files based on a common developer involvement. That means if two code files are written or maintained by the same developer then there exists a relationship between them. To draw this graph an $intraConnect(G_{File-Developer}, Developer)$ operation is performed.
issue to issue_dev	This graph shows relationships among issues based on common developers' involvement, which in turn reveals what subject area of these issues actually motivates developers' in solving them. To draw this graph an $intraConnect(G_{Issue-Developer}, Developer)$ operation is performed.

continued on following page

Table 1. continued

dev to dev_Intersection	This graph shows the group of developers in the community who work as active developers as well as issue solvers. Edge weights in this graph shows how frequently two developers interact with each other while performing both the roles. To draw this graph an $intersection(G^{F}_{developer}, G^{I}_{developer})$ operation is performed. Details of this graph operation are presented in the appendix, section Set operations.
dev to user_issue	This graph shows relationships between developers and users of the software based on a common issue, as in a typical OSS project one of the major media of communication among the developers and users is the issue tracking system. To draw this graph an $intersection(G^{F}_{developer}, G^{I}_{developer})$ operation is performed.
file to file_Intersection	This graph shows to what extent the inclusion architecture of the code base matches that of the file relationship due to the developers' involvement. In other words it reveals whether the community structure matches that of the product architecture, an application of Conways law (Conway, 1968). To draw this graph an $intersection(G^{Dev}_{File}, G^{Inc}_{File})$ operation is performed.
dev to dev_Union	This graph shows relationships among all the developers in the community based on their working domain. So, this graph would answer questions like, who is doing what, to what extent and in doing their jobs who are the other people they are interacting with. To draw this graph a $union\left(G^{F}_{developer}, G^{I}_{developer}\right)$ operation is performed. Details of this graph operation are presented in the appendix, section Set operations.
dev to dev_Difference	This graph shows the relationships of that part of the developer community who work only as developers. To draw this graph a $difference(G^{F}_{developer}, G^{I}_{developer})$ operation is performed. Details of this graph operation are presented in the appendix, section Set operations.
file to file_Difference	This graph shows those parts of the code base that do not match with the file relationships due to community involvement. To draw this graph a $difference(G^{Inc}_{File}, G^{Dev}_{File})$ operation is performed.
dev to dev_SymmDiff	This graph portrays two groups (if exists) in the developer community, one of which performs specifically as developers and the other only as issue solvers. To draw this graph a s $ymmdifference(G^{F}_{developer}, G^{I}_{developer})$ operation is performed. Details of this graph operation are presented in the appendix, section Set operations.

style private and hybrid cloud computing (Eucalyptus, 2011). The platform provides a single interface that lets users access computing infrastructure resources (e.g., machines, network, and storage). These resources are available in private clouds implemented by Eucalyptus inside an organization's existing data center as well as externally in public cloud services. The software is designed to be modular and the extensible web-services based architecture enables Eucalyptus to export a variety of APIs toward users via client tools. Currently Eucalyptus implements the Amazon Web Service (AWS) API (Eucalyptus, 2011), which allows interoperability with existing AWS-compatible services and tools.

There are the enterprise edition and the open-source edition of Eucalyptus. Version 1.5.2 was the first release as an open source. This open source version of Eucalyptus is currently available with most of the Linux distributions including Ubuntu, Red Hat Enterprise Linux (RHEL), CentOS, SUSE Linux Enterprise Server (SLES), OpenSUSE, Debian and Fedora.

As cloud computing is an emerging phenomenon and the Eucalyptus OSS community makes a large contribution in this field, we are thus particularly interested in the open source version of Eucalyptus. For our purposes we explore all the possible sources of information provided by the open source version of Eucalyptus. The sources are mainly available in the following categories: the user information system, the version control system and the bug reporting system.

Figure 14. Graph customization on the attribute geo location with value "America"

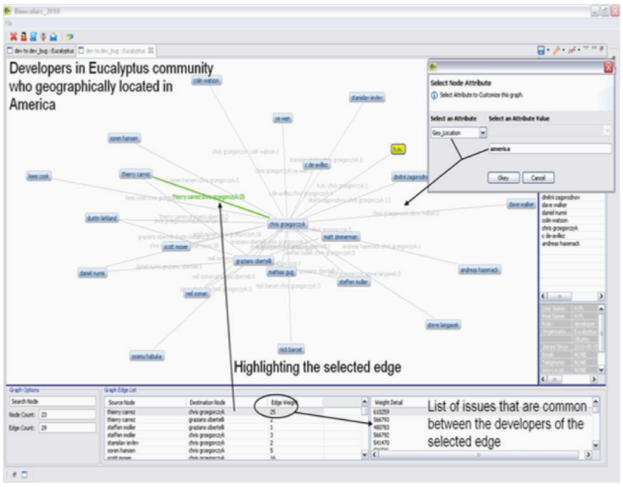
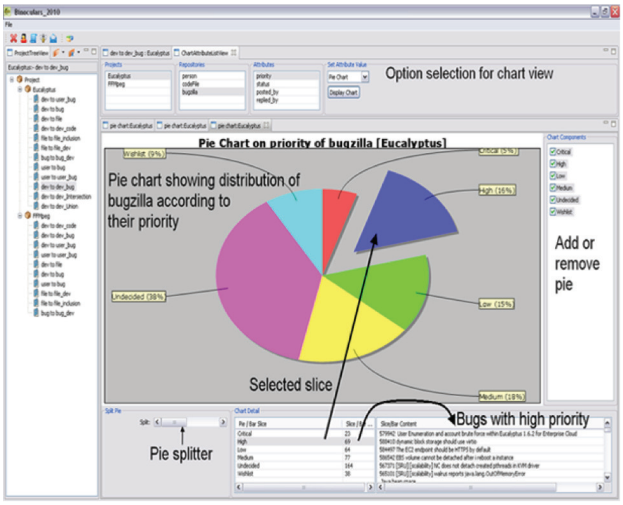

Developers in Eucalyptus community who geographically located in America
Highlighting the selected edge
List of issues that are common between the developers of the selected edge
Select Node Attribute
Select an Attribute
Select an Attribute Value
Geo_Location
america
Okay
Cancel

Figure 15. Pie chart showing bugzilla distribution according to their priority in Eucalyptus project



Research Questions

Following are the queries that are investigated and answered using Binoculars. Primary focuses of these queries are the community aspects, the code base and their relationships in an OSS project.

Q1. Where does the expertise lay within the developer community?

Q2. Whom (developer) users should contact to solve an issue?

Q3. How many people work on each software component? Who are they?

Q4. Does the inclusion structure of the code base conform to the organizational structure (i.e., do Conway's law apply)?

Q5. What are the critical issues related to the performance of the software?

These queries are motivated by the fact that- software development is no longer a single handed job (Rosso, 2009). Rather, it is a collaborative and distributed work in both form of software development (closed source and open source) process. Thus cultural, time zone, and language differences among the community members are obvious. Improving software development in this context requires better understanding and improved coordination and communication (Mockus & Herbsleb, 2003; Herbsleb, Moitra, & Lucent Technol., 2001). This coordination and collaboration issues for improving software knowledge are studied in software engineering for quite a long time (Cockburn, 2001; Schwaber & Beedle, 2001) and was empirically validated (Bellini, Canfora, Garcia, Piattini, & Visaggio, 2005). Likewise, exploiting and understanding this collaboration and coordination, and its relation with the underlying software architecture has profound impact both on software architecture evolution (CYB, MacCormack, & Rusnak, 2008) and on software quality (Ye, 2006; Herbsleb, 2007). The queries stated are derived from these needs. In Rosso (2009) some of these queries are explored and answered from community perspective using SNA measures and metrics. But, for in-depth understanding on this issue requires exploration of relationship between community and the code base of an OSS project.

In what follows, the data collection process and answer to the research questions using FFMpeg and Eucalyptus. In answering the queries, first a brief discussion on the context and focus of the query is given.

Data Collection

The main sources of information for the two projects, FFMpeg and Eucalyptus are, code repository, bug tracking system, registered user information system and mailing list. We take a snapshot of the SVN code repository of FFMpeg project on 20-09-2010 and use the code base of Eucalyptus-1.6.2. Other information sources (bug tracking system, user information system and mailing lists) are extracted from the first entry up to 20-09-2010. For extracting data from the information sources described above, we build parsers in java and represent the extracted data in XML files which constitute the repositories for Binoculars.

As discussed, Binoculars contains three repositories to store extracted information, namely, Person, File and Issue repositories. Person repository contains project personnel information including, unique user name; real name; role performed in the community (e.g., developer, user, active developer); date of joining to the project; project or part of the project working for; other detail such as, geographical location, email address, communication medium used, preferred language(s) and phone number. File repository contains information about the code files, which consists of file name; its path to the svn; extension of the file; developer(s) who implemented or maintain the file; inclusion information, and copyright information. Finally, Issue repository contains information extracted from the bug repository as well as from the mailing list. This repository consists of issue id; title of the issue (subject of a mail or a bug report); its priority level, and current status; name of the persons who posted and replied to the issue along with date and time of posting, and the description of the issue.

To identify and distinguish personnel (either developer or user) in a project, we adopted the following approach; first we collected the person information from the code repository for each code file. Then we searched the registered user information system to identify personnel with their role defined as developer. These two sources provide the developers list in a project. Other personnel whose responsibility was mentioned as user in the registered user information system or who are not in the developer list were considered as users of the project. Also for code files, it might be possible that two files have the same name. To resolve this conflict, we used absolute path for each code file.

Case Study: 1 FFMpeg

In this section we discuss how selected queries are answered when analyzing an FFMpeg project using Binoculars.

Q1. Where does the expertise lay within the developer community?

Description: Developers might have different areas of interest and expertise with respect to the code base and responsibilities should be distributed accordingly. Thus, it is obvious that developers having the same interest area would share knowledge more frequently and they are the main sources of information for that portion of the code base. The main focus of this query is to find out the expertise groups within the developer community.

To answer this query an *intraConnect* ($G_{File\text{-}Developer}$, *File*) operation is performed. The resultant graph is shown in Figure 16.

According to this graph, developers MN and FB share 65 code files between them. However, developer FB shares about 49 code files with 19 other developers. M.N. also shares 40 code files with another 15 developers. Thus, it is quite clear that both MN and FB are the central developers in FFMpeg and make a significant contribution to different areas of the code base.

Q2. Whom (developer) users should contact to solve an issue?

Description: In an open source project developers play two types of roles: (a) developing and maintaining the software and (b) dealing with user issues, which may directly or indirectly relate to the product. Depending on the area of interest, developers may choose either or both of these roles. Developers playing both roles and exchanging ideas with each other form the core of the developer community. This community of developers has the in-depth knowledge of the software and the user community as they both deploy the software and solve user polled issues. A user having some problem in hand may find the proper person in the developer community whom he should contact. This question was also polled in Anvik, Hiew, and Murphy (2006) where a machine learning technique was used to suggest a small number of developers suitable to resolve a reported issue.

To answer this query an:

$$intersection(G^{F}_{developer}, G^{I}_{developer})$$

operation is performed. The resultant graph contains only 13 developers but they have no connection between them. This means that the communication network in the developer community which evolved due to the code development is not the same as the one evolved due to bug or issue solving. In other words, developers who are responsible for a particular portion of the code base are not necessarily the same group of developers who are responsible for solving related issues.

Q3. How many people work on each software component? Who are they?

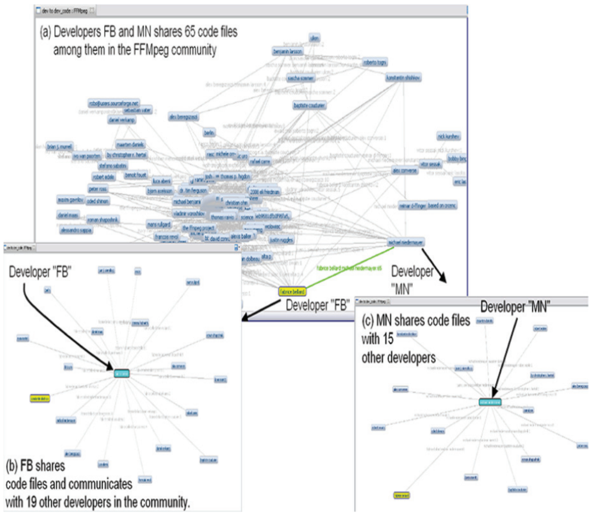
Description: The goal is to construct a graph that actually reveals the relationships between developers and the code files to which they have contributed. This graph would help to identify how responsibilities are distributed over the community.

To answer this query an *interConnect* (*File*^{F}, *developer*) operation is performed. The resultant graph reveals that the distribution of the files in not uniform in the FFmpeg project. Rather, responsibilities are disseminated according to seniority and experience. For example, as shown in Figure 17, the top four developers contributed to around 550 code files of which the developer MN contributed to 227 code files alone.

Q4. Does the inclusion structure of the code base conform to the organizational structure (i.e., do Conway's law apply)?

Description: The inclusion structure shows the logical relationships and interdependencies among the code files. Changes to such files might affect other related ones. Thus distribution of files among developers plays an important role in software evolution. If developers working on the same or related code files make changes to those files without the

Figure 16. Developer relationship graph based on code file sharing in FFMpeg with top two developers communication domain


(a) Developers FB and MN shares 65 code files among them in the FFMpeg community
Developer "FB"
Developer "MN"
Developer "FB"
(b) FB shares code files and communicates with 19 other developers in the community.
Developer "MN"
(c) MN shares code files with 15 other developers

other developers involved being aware, then conflicting changes must occur. Due to this lack of communication among the developers working on such inter-related files, these conflicts are detected and resolved only during commits. But this increases the cost of resolving those conflicts.

To carry out this query, an *intersection* $(G_{File-File},\ G^{F}_{File})$ operation is performed. This operation takes two graphs as argument. One is a $G_{File\text{-}File}$ graph which is generated by employing the *interConnect*(*File*^{F}, *includeFile*) operation, and the other is a G^{F}_{File} graph generated by applying the *intraConnect*($G_{File\text{-}Developer}$, *Developer*) operation. The resultant graph contains nodes and edges that are common to both the graphs. This graph is shown in Figure 18.

The FFMpeg project consists of 1490 code files, but the resultant graph (Figure 18) contains only 765 code files and their relationships. Thus, around 50% of the code files distributions are concerned with the organizational structure in the FFMpeg project.

Q5. What are the critical issues related to the performance of the software?

Description: The main focus of this query is to categorize the issues based on a particular condition. As an example, we categorize the issues based on the assumption that developers invest their time on those issues that are closely related to the performance of the software. It is also desirable that these issues are related to the code files for which these developers are responsible. Again, if enough information is provided by the repository, then these issues would directly lead to the sections of the code base that raise these issues.

To answer this query, an *intraconnect* $(G_{Developer-Issue}, Developer)$ operation is performed. The resultant graph is shown in Figure 19. The edge weight in this graph shows how many developers are involved when solving two connected issues. A close look at this graph reveals that there are a few issues in the FFMpeg project that are referenced frequently by the developers with other issues. For example, *issue1322* is referenced with 258 other issues (Figure 19). Similarly, *issue763* and *issue272* are referenced with other 237 and 217 issues respectively. This wide interconnection between these issues and others reveals that they are

Figure 17. Developer and code file relationship graph in FFMpeg

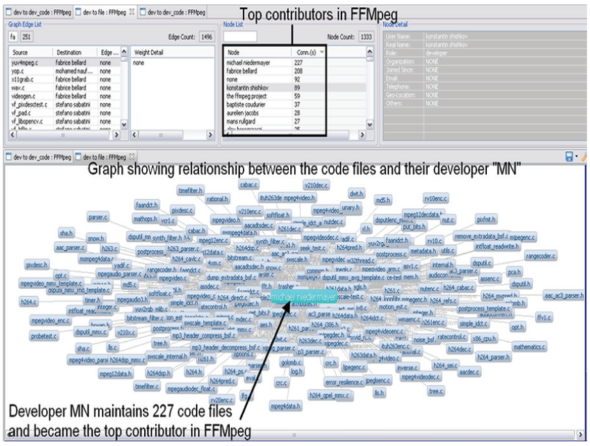
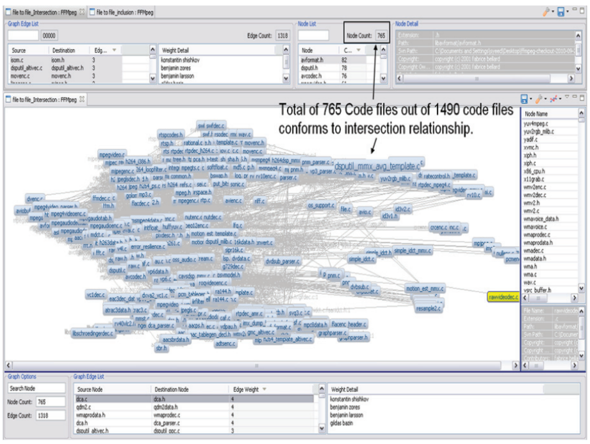

Top contributors in FFMpeg
Graph showing relationship between the code files and their developer "MN"
Developer MN maintains 227 code files and became the top contributor in FFMpeg

Figure 18. File relationship graph generated based on common developer involvement and inclusion structure


Total of 765 Code files out of 1490 code files conforms to intersection relationship.

critical for the project and have a deep impact on the performance of the software.

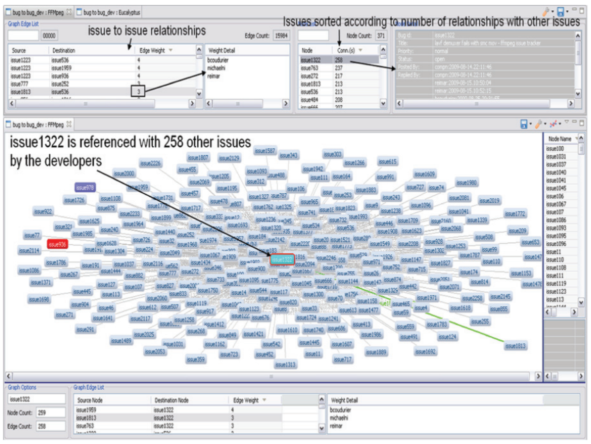
Case Study: 2 Eucalyptus

In this section an analysis report of the Eucalyptus community is presented. The same set of queries as in the FFMpeg project is investigated in this case as well. The contexts of the queries are the same as in the previous analysis.

Q1. Where does the expertise lay within the developer community?

Description: The Eucalyptus community currently has 12 active developers, who participate in the project development. There are also other developers who contribute to the project. But, due to the fact that their contributions are not maintained properly at the code base level, the overall contribution of the developer community cannot be measured. Based on the partial information available, it is apparent that three of the developers in the Eucalyptus community have contributed more than the others.

Figure 19. Issue relationship graph generated based on common developer involvement


issue to issue relationships
Issues sorted according to number of relationships with other issues
issue1322 is referenced with 258 other issues by the developers

Q2. Whom (developer) users should contact to solve an issue?

Description: The outcome of this query is quite similar to that of the FFMpeg, because there is no distinct group of developers who communicate while working on the project as well as dealing with user issues. That is developers who implement certain code files are not necessarily the same ones who solve issues related to those code files.

Q3. How many people work on each software component? Who are they?

Description: The result of this query is also similar to that of the FFMpeg project, where responsibilities are not evenly distributed within the developer community. For example, developer CZ alone contributed to 202 (out of 676) code files in the Eucalyptus project.

Q4. Does the inclusion structure of the code base conform to the organizational structure (i.e., do Conway's law apply)?

Description: It is found that around 39% (266 out of 676) of the code files are distributed according to the communication structure of the Eucalyptus developer community. This is significantly lower than that of the FFMpeg community. This might be due to the fact that the

Eucalyptus community is a new and growing one which is still converging towards homogeneity.

Q5. What are the critical issues related to the performance of the software?

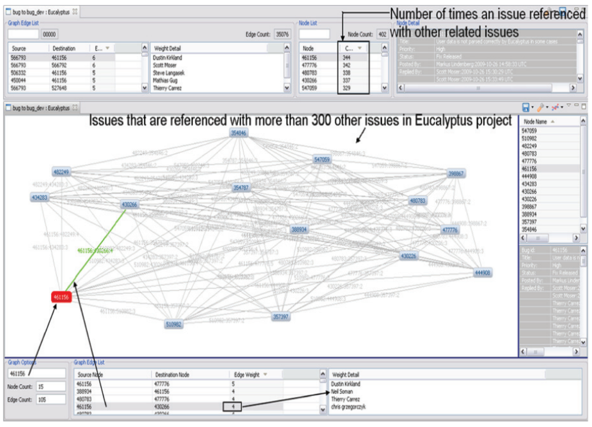
Description: The result of this query in the Eucalyptus project shows that while working on one issue, developers the project consists of 402 issues, of which 350 issues are referenced with more than 50 other issues. In other words, in around 87% cases when developers show an interest in one issue they have also worked on at least 50 other issues. The graph in Figure 20 shows only those issues and their relationships that were referenced with more than 300 other issues by the developers.

This result is quite different from the FFMpeg project where the rate of this cross referencing is significantly lower.

RELATED WORK

The open source research community has proposed several techniques and tools to study open source projects. These tools, and the approaches behind them, can be classified along

Figure 20. Issue relationship graph generated based on common developer involvement


Number of times an issue referenced with other related issues
Issues that are referenced with more than 300 other issues in Eucalyptus project
bug to bug_dev : Eucalyptus
Graph Edge List
Node List
Node Detail
Graph Options

Edge Count: 35076

Source	Destination	E...
566793	461156	6
566793	566792	6
506332	461156	5
450044	461156	5
566793	527648	5

Weight Detail
Dustin Kirkland
Scott Moser
Steve Langasek
Mathias Gug
Thierry Carrez

Node Count: 402

Node	C...
461156	344
477776	342
480783	338
430266	337
547059	329

Title:	User data is not parsed correctly by Eucalyptus in some cases
Priority:	High
Status:	Fix Released
Posted By:	Markus Lindenberg:2009-10-26 14:58:33 UTC
Replied By:	Scott Moser:2009-10-26 15:30:29 UTC
	Scott Moser:2009-10-26 15:33:49 UTC

461156

Node Count: 15

Edge Count: 105

Source Node	Destination Node	Edge Weight
461156	477776	5
388934	461156	4
480783	477776	4
461156	430266	4

Weight Detail
Dustin Kirkland
Neil Soman
Thierry Carrez
chris grzegorczyk

two main dimensions as shown in Table 2). The first dimension is the purpose of the tool (why), examples of which include social network analysis, code analysis, licensing investigation, project exploration, and conflict management. The second dimension is the data source that is used to extract information (what), examples of which include the source code, revision history, and mailing lists.

We chose to group the data sources into different categories. The software category refers to software artifacts like the source code and configuration files. Communication includes data sources such as mailing lists, and chat entries. Configuration refers to elements like revision history and bug tracks. Knowledge includes data sources such as user forums and Wiki entries. Statistics cover data concerning the number of downloads or web hits. Finally, contribution covers patches and feature requests.

These categories are not orthogonal, for example, communication overlaps with contribution as mailing list entries are often referred to feature requests. As can be seen in Table 2, every tool focuses on a specific purpose and uses a set of data sources. For example, in Beaver, Cui, St Charles, and Potok (2009) communication and knowledge sources are used to predict the success of open source projects. Also in Mockus and Herbsleb (2002), configuration and contribution are explored to identify expertise within the project group, whereas in Biehl, Czerwinski, Smith, and Robertson (2007) and Sarma, Noroozi, and Van der Hoek (2003), source code and configuration files were mined to increase the developer' awareness of each others' contribution within the project and to effectively identify and resolve conflicts when working on shared artifacts, respectively. This gives the impression that (i) the purposes are unrelated and (ii) certain data sources are applicable for certain purposes only.

We argue that the tool purposes shown in Table 2 are in fact different functionalities of a generic OSS analysis tool. We also argue that the data sources are in fact complementary and that one data source could be relevant for several purposes. For example, source code is used to perform traditional code analysis (Knab, Pinzger, & Bernstein, 2006), social network analysis (Martinez-Romo, Robles, Ortuo-Perez, & Gonzalez-Barahona, 2008), licensing investigation (Di Penta & German, 2009) and so on.

The tool that comes closest to our approach is the Tesseract (Sarma, Maccherone, Wagstrom, & Herbsleb, 2009). This tool shows simultaneously the social and technical aspects of the relevant project, and cross links the two. It also takes into account project evolution by allowing interactive exploration of the data in a selected time period. And it highlights matches and mismatches among the technical dependencies and communication patterns of the developers. Still Tesseract has its limitations. First, the main focus of this tool the commercial software development projects and it tries to accommodate the aspects that are similar to open source software projects. Second, this tool does not consider the user base of OSS projects, which constitutes a large part of the OSS community and contributes to the project in many ways, such as bug reporting, asking for new features, feedback, mail communication and so on. Third, data collection for this tool is more manual than automated. It requires cross-checking and validation with project personnel. This process might take months to prepare data, which would hinder the usefulness of the tool.

Compared to these approaches, we proposed a compact and extensible metamodel (see section TOWARDS A GENERIC OSS ANALYSIS TOOL), since particularly at a more detailed level, the types of artifacts and their relations to be analyzed differ from project to project.

Correspondingly, support for the other dimensions discussed above can be provided either by extending the repositories or by extending the metamodel. For example, to integrate fault prediction of OSS projects, it is sufficient to incorporate the measured metrics in the corresponding repositories. On the other hand to incorporate licensing we need to extend the metamodel. It can be achieved by attaching licensing details to FileSystemItem class shown in Figure 1.

Our approach is currently aimed at the software and community sides of OSS projects. Each of these dimensions are modeled and implemented through repositories as discussed in section Repository Structure and in the appendix. As SNA traditionally employs mathematical graphs and metrics (Carrington, Scott, & Wasserman, 2005) to render and analyze relationships, we further define a set of operations (see section Graph Operations and appendix) which help support such graph construction and analysis. For example, the traditional 2-mode network in SNA can be easily constructed using the *interConnect* operation (section Graph Operations and in appendix, section The interConnect operation). This 2-mode network could then be easily transformed into two 1-mode SNA networks through the *intra-Connect* operation (section Graph Operations and in appendix section The intraConnect operation). Also for further querying, a group of set operations are defined (section Graph Operations and in appendix, section Set operations) and implemented. These operations are reusable in the sense that if other dimensions of OSS projects are mapped through repositories, then these operations can be directly applied to them as well. Thus our approach is more formal, procedural and flexible for exploring the socio-technical aspects of an OSS project while keeping perfect alignment with SNA approaches as well as reverse engineering.

DISCUSSION

Conclusion

The research problem addressed in this paper is what kind of infrastructure is needed to meet the requirements of a generic OSS analysis tool. Our approach in addressing this challenge is as follows.

First, we proposed a meta-model based approach for OSS analysis. We started the development of the metamodel, given in section II, focusing on two purposes, namely code analysis and social network analysis, and in particular combination of the two. The metamodel includes concepts for the community (Person), the software (FileSystemItem), and communication (Issue). It is not complete, as it could be augmented, for instance, with *legality* issues (like License) and *economical issues*.

Second, we formally defined graph operations (e.g., intra-Connect, interConnect, intersection, union, difference and symmetric difference), due to the fact that the abstract metamodel is a mathematical graph. These operations are discussed in section Graph Operations and in the appendix. The analysis is done by applying these operations.

Third, as a proof of the concept we developed the analysis tool Binoculars. This tool incorporates the described metamodel, and the analysis is carried out by applying the graph operations on the model. The architecture and features of this tool are discussed in TOOL SUPPORT. Binoculars were used to analyze two projects, FFMpeg and the open source version of Eucalyptus (presented in section CASE STUDY). We were able to answer questions like "Whom should a user contact to solve an issue" and "Who are the developers working on each component". During the trial the tool showed promising functionality; we were able to find answers to the enumerated questions. Despite the fact that FFMpeg can be considered as a large open source project, Binoculars worked well and smoothly. The developed metamodel proved to be excellent.

Finally, we gave a two-dimensional (purpose of the tool and used data source) classification for open source software analysis tools (section RELATED WORK). Then we argued that the tool purposes are actually different functionalities of a generic OSS tool. Moreover, the data sources are complementary and can be used for different purposes. We also discuss how our approach can be applied to fill this gap.

Ultimately, the kind of questions which the Binoculars approach addresses is not limited to open source projects. Other software development settings such as globally distributed projects may involve similar concerns such as the impact of team distribution on software architecture (Avritzer, Paulish, & Cai, 2008). The only assumption which our approach makes is that the subject project comes with relevant development data such as team communication and revision history. Even in single site development setting, the binoculars approach can be used for knowledge combination (Nonaka & Takeuchi, 1995) by combining together various elements of explicit knowledge (codified in the development artifacts) in order to build a bigger system of knowledge. A simple example of combination is the merging of the results of two developers' development history to identify overlapping efforts.

On the negative side: The GUI of Binoculars is challenging. Namely, the size of large projects tends to lead to huge graphs, which are complex and slow to render on the screen. This is a well-known challenge in the development and application of reverse engineering and program analysis tools, and is often tackled with abstraction and slicing techniques. We are studying such graph abstractions that would preserve the interesting features, but produce small enough graphs to be visualized.

Also, it is possible that two persons having the same name in a project have different responsibilities (e.g., one as a developer and the other as a user). In such case, Binoculars would not be able to distinguish between them and may lead to inconsistent result.

FUTURE WORK

The most widely accepted reverse engineering metamodel is The Dagstuhl Middle Metamodel (DMM) (Lethbridge, Tichelaar, & Ploedereder, 2004), which is illustrated in Figure 21. DMM includes various viewpoints to the system under reverse engineering. Figure 21 illustrates the file-oriented portion of the actual class diagram of DMM. The idea behind the class structure is to separate the abstract model from the concrete source code. Therefore, the model has both SourceObject and ModelObject.

Our goal is to augment DMM with open source community related concerns. Figure 22 illustrates our current understanding of the augmentation of DMM. Remember that the Binoculars' metamodel was given in Figure 1. FileSystemItem has been left out from the Binoculars metamodel. Due to the fact that open source development is very code-oriented activity, we have chosen the subclasses of SourceObject to points of augmentation. The left-hand side of the figure shows the Binoculars metamodel and the righthand side illustrates the DMM. The four relations between the models show how they are related.

Persons are either users or developers of the system. Developers work with many SourceUnits and one SourceUnit may be coded by several developers. Therefore, Person

Table 2. A two dimensional classification of open source analysis tools

-	Software	Communication	Configuration	Knowledge	Statistics	Contribution
Code analysis	(Knab, Pinzger, & Bernstein, 2006; Sarma, Maccherone, Wagstrom, & Herbsleb, 2009)(Knab, Pinzger, & Bernstein, 2006; Sarma, Maccherone, Wagstrom, & Herbsleb, 2009)		(Sarma, Maccherone, Wagstrom, & Herbsleb, 2009; Zhou & Davis, 2005)(Sarma, Maccherone, Wagstrom, & Herbsleb, 2009; Zhou & Davis, 2005)			(Chacon, 2010)
SNA	(Martinez-Romo, Robles, Ortuo-Perez, & Gonzalez-Barahona, 2008; Sarma, Maccherone, Wagstrom, & Herbsleb, 2009)(Martinez-Romo, Robles, Ortuo-Perez, & Gonzalez-Barahona, 2008; Sarma, Maccherone, Wagstrom, & Herbsleb, 2009)	(Kamei, Matsumoto, Maeshima, Onishi, Ohira, & Matsumoto, 2008; Crowston & Howison, 2005)	(Porruvecchio, Uras, & Quaresima, 2008)	(Mller, Meuthrath, & Baumgra, 2008)	(Wiggins, Howison, & Crowston, 2009)	
Licensing	(Di Penta & German, 2009; Tuunanen, Koskinen, & Karkkainen, 2006)					
Prediction	(English, Exton, Rigon, & Cleary, 2009)	(Beaver, Cui, St Charles, & Potok, 2009)	(English, Exton, Rigon, & Cleary, 2009)	(Beaver, Cui, St Charles, & Potok, 2009)		
Evolution	(Bouktif, Antoniol, Merlo, & Neteler, 2006; Capiluppi & Fernandez-Ramil, 2007)	(Bouktif, Antoniol, Merlo, & Neteler, 2006)	(Bouktif, Antoniol, Merlo, & Neteler, 2006; Capiluppi & Fernandez-Ramil, 2007)		(Koch & Stix, 2008)	(Deshpande & Riehle, 2008)
Project Exploration	(Souza, Quirk, Trainer, & Redmiles, 2007; DeLine, Khella, Czerwinski, & Robertson, 2005)		(Mockus & Herbsleb, 2002; DeLine, Khella, Czerwinski, & Robertson, 2005)			(Mockus & Herbsleb, 2002)
Awareness	(Biehl, Czerwinski, Smith, & Robertson, 2007; Froehlich & Dourish, 2004)	(Froehlich & Dourish, 2004)				
Conflict management	(Sarma, Noroozi, & Van der Hoek, 2003; Schmmer & Haake, 2001)					

Figure 21. The Dagstuhl Middle Metamodel (DMM)

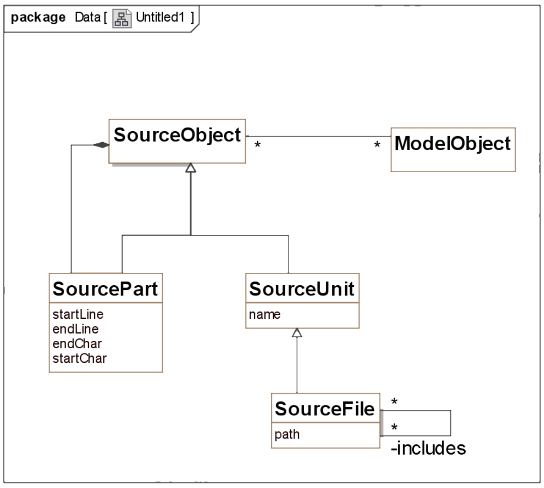
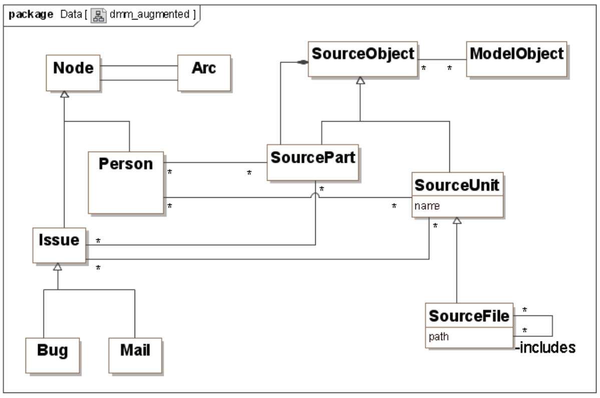


Figure 22. The DMM augmented with the community dimension



and SourceUnit are in many-to-many relation. The SourcePart is used to model the code snippets crafted by the developers. Therefore, User and SourcePart are in a many-to-many relation. Similarly, issues are related to SourceUnits. Namely, most of the issues deal with the code-oriented questions, like bugs in the system. Therefore, Issue has a many-to-many relation with both SourcePart and SourceUnit.

The described augmentation is our next step in developing Binoculars. The relation of the current Binoculars metamodel and the more abstract parts of the DMM is worth a more comprehensive study. Is it enough just to relate our model to the code?

Current results of the tool support show very promising results and thus are subject to further study and research. As this current tool is a prototype and mainly developed to study how far it matches up to our expectations, extensions to the tool would be a topic of future work. Hence, the proposal is to develop a generic OSS analysis tool that must conform to the following.

It must support the traditional reverse engineering concept along with a community analysis mechanism.

- Customized visualization and rendering of graphs along with different levels of abstraction of data should be supported.
- From the architectural point of view: the tool should have an architecture that allows its extension, customization, and tailoring for different needs. It should be flexible enough to be tailored for other purposes relevant to the analysis of open source projects.
- The tool itself will be open source, which further supports its easy customization. The use needs of the tool should facilate the following.
- Users of open source projects: the tool can be used to get an understanding of an open source project in order to assess its relevance for the needs of the user.
- Developers of open source projects: since the tool itself will be open source and customizable, the developers can publish a tailored version of the tool together with the software itself which would better serve the end users.

REFRENCES

Anvik, J., Hiew, L., & Murphy, G. (2006). *Who should fix this bug? 28th international conference on Software engineering* (pp. 361–370). New York, NY, USA: ACM.

Avritzer, A., Paulish, D., & Cai, Y. (2008). Coordination implications of software architecture in a global software development project. *Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)* (pp. 107–116). Washington, DC, USA: IEEE Computer Society.

Beaver, J., Cui, X., St Charles, J., & Potok, T. (2009). Modeling success in floss project groups. *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, (pp. 1-8).

Bellard, F. (2006). *Ffmpeg naming and logo*. Retrieved from FFmpeg mailing list: lists.mplayerhq.hu/pipermail/ffmpeg-devel/2006-February/

Bellini, E., Canfora, G., Garcia, F., Piattini, M., & Visaggio, C. (2005). Pair designing as practice for enforcing and diffusion design software. *Journal of Software Maintenance and Evolution: Research and Practice*, *17*(6), 401–423. doi:10.1002/smr.322

Biehl, J., Czerwinski, M., Smith, G., & Robertson, G. (2007). Fastdash: A visual dashboard for fostering awareness in software teams. *SIGCHI conference on Human Factors in computing systems*, (pp. 1313–1322). San Jose, California, USA.

Bouktif, S., Antoniol, G., Merlo, E., & Neteler, M. (2006). A feedback based quality assessment to support open source software evolution: the grass case study. *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, (pp. 155–165).

Capiluppi, A., & Fernandez-Ramil, J. (2007). A model to predict anti-regressive effort in open source software. *Proceedings of the IEEE International Conference on Software Maintenance*, (pp. 194–203).

Carrington, P., Scott, J., & Wasserman, S. (2005). *Models and methods in social network analysis*. Cambridge University press.

Cockburn, A. (2001). *Agile software development*. Indianapolis, IN: Addison-Wesley Professional.

Conway, M. E. (1968). *How do committees invent?* F. D. Thompson Publications, Inc. Reprinted by permission of Datamation magazine.

Crowston, K., & Howison, J. (2005). *The social structure of free and open source software development.* First Monday.

CYB. A., MacCormack, D., & Rusnak, J. (2008). Exploring the duality between product and organizational architectures: A test of the mirroring hypothesis. *Working Papers, Harvard Business School.*

DeLine, R., Khella, A., Czerwinski, M., & Robertson, G. (2005). Towards understanding programs through wear-based filtering. *ACM Symposium on Software Visualization*, (pp. 183–192). St. Louis, Missouri.

Deshpande, A., & Riehle, D. (2008). Continuous integration in open source software development. *Open Source Development, Communities and Quality, IFIP 20th World Computer Congress, Working Group 2.3 on Open Source Software*, (pp. 273–280).

Di Penta, & German, D. (2009). Who are source code contributors and how do they change? *Proceedings of 16th Working Conference on Reverse Engineering*, (pp. 11–20).

DOM XML parser. (2010). Retrieved from DOM XML parser: www.w3schools.com

Eclipse project. (2010). Retrieved from Eclipse project: www.eclipse.org

English, M., Exton, C., Rigon, I., & Cleary, B. (2009). Fault detection and prediction in an open-source software project. *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, (pp. 17-27).

Eucalyptus. (2011). Retrieved from The open source cloud platform: http://open.eucalyptus.com

FFmpeg. (2010). Retrieved from FFmpeg project: www.ffmpeg.org

Froehlich, J., & Dourish, P. (2004). Unifying artifacts and activities in a visual tool for distributed software development teams. *International Conference on Software Engineering*, (pp. 387–396). Edinburgh, UK. *git version control system*. (2010). Retrieved from git version control system: www.git-scm.com

Guttag, J., & Horning, J. (1993). *Larch: Languages and tools for formal specification*. New York, NY: Springer-Verlag, New york, Inc.

Herbsleb, J. (2007). *Global software engineering: The future of socio-technical coordination. Future of Software Engineering* (pp. 188–198). Washington, DC, U.S.A: IEEE Computer Society.

Herbsleb, J., Moitra, D., & Lucent Technol, I. (2001). Global software development. *IEEE Software*, *18*(2), 16–20. doi:10.1109/52.914732

Java runtime enviornment. (2011). Retrieved from Java runtime enviornment: http://www.oracle.com/

JFreeChart. (2010). Retrieved from JFreeChart: www.jfree.org

Kamei, Y., Matsumoto, S., Maeshima, H., Onishi, Y., Ohira, M., & Matsumoto, K. (2008). Analysis of coordination between developers and users in the apache community. *Proceedings of the Fourth Conference on Open Source Systems*, (pp. 81–92).

Knab, P., Pinzger, M., & Bernstein, A. (2006). Predicting defect densities in source code files with decision tree learners. *Proceedings of the International workshop on Mining software repositories*, (pp. 119–125).

Koch, S., & Stix, V. (2008). Open source project categorization based on growth rate analysis and portfolio planning methods. *Open Source Development, Communities and Quality, IFIP 20th World Computer Congress, Working Group 2.3 on Open Source Software*, (pp. 375–380).

Lethbridge, T. C., Tichelaar, S., & Ploedereder, E. (2004). The dagstuhl middle metamodel: A schema for reverse engineering. *Electronic Notes in Theoretical Computer Science*, 7–18. doi:10.1016/j.entcs.2004.01.008

Martinez-Romo, J., Robles, G., Ortuo-Perez, M., & Gonzalez-Barahona, J. M. (2008). Using social network analysis techniques to study collaboration between a floss community and a company. *Proceedings of the Fourth Conference on Open Source Systems*, (pp. 171–186).

Mller, C., Meuthrath, B., & Baumgra, A. (2008). Analyzing wiki based networks to improve knowledge processes in organizations. *Journal of Universal Computer Science*, *14*(4), 526–545.

Mockus, A., & Herbsleb, J. (2002). Expertise browser: A quantitative approach to identifying expertise. *Proceedings of International Conference on Software Engineering*, (pp. 503–512). Orlando.

Mockus, A., & Herbsleb, J. (2003). An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering*, *29*(6), 481–494. doi:10.1109/TSE.2003.1205177

Nonaka, I., & Takeuchi, H. (1995). *The knowledge-creating company:How japanese companies create the dynamics of innovation*. New York: Oxford University.

Porruvecchio, G., Uras, S., & Quaresima, R. (2008). Social network analysis of communication in open source projects. *9*, pp. 220–221. Proceedings of 9th International Conference on Agile Processes in Software Engineering and Extreme Programming.

Rich Client Platform. (2010). Retrieved from Rich Client Platform: wiki.eclipse.org

Rosso, C. (2006). Continuous evolution through software architecture evaluation. *Journal of Software Maintenance and Evolution: Research and Practice*, *18*(5), 351–383. doi:10.1002/smr.337

Rosso, C. (2009). Comprehend and analyze knowledge networks to improve software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, *21*, 189–215. doi:10.1002/smr.408

Sarma, A. Maccherone, Wagstrom, & Herbsleb. (2009). Tesseract: Interactive visual exploration of socio-technical relationships in software development. *Proceedings of the 31st International Conference on Software Engineering*, (pp. 23–33).

Sarma, A., Noroozi, Z., & Van der Hoek, A. (2003). Palantr: Raising awareness among configuration management workspaces. *Twenty-fifth International Conference on Software Engineering*, (pp. 444–454). Portland, Oregon, USA.

Schmmer, T., & Haake, J. M. (2001). Supporting distributed software development by modes of collaboration. *Seventh European Conference on Computer Supported Cooperative Work*, (pp. 79–98).

Schwaber, K., & Beedle, M. (2001). *Agile software development with Scrum*. Englewood Cliffs, NJ: Prentice-Hall.

Souza, C. d., Quirk, S., Trainer, E., & Redmiles, D. (2007). Supporting collaborative software development through the visualization of socio-technical dependencies. *International ACM SIGGROUP Conference on Supporting Group Work*, (pp. 147–156). Sanibel Island, FL.

Tuunanen, T., Koskinen, J., & Karkkainen, T. (2006). Asla: reverse engineering approach for software license information retrieval. *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, (pp. 291–294).

Wiggins, A., Howison, J., & Crowston, K. (2009). Heartbeat: Measuring active user base and potential user interes in floss projects. [IFIP Advances in Information and Communication Technology.]. *Open Source Ecosystems: Diverse Communities Interacting.*, *299/2009*, 94–104. doi:10.1007/978-3-642-02032-2_10

Ye, Y. (2006). Supporting software development as knowledgeintensive and collaborative activity. *International Workshop on Interdisciplinary Software Engineering Research*, (pp. 15–22). New York NY, U.S.A.

Zhou, Y., & Davis, J. (2005). Open source software reliability model: an empirical approach. *Proceedings of the fifth workshop on Open source software engineering, 30*, pp. 1-6.

M. M. Mahbubul Syeed is a PhD student and working as a researcher in the department of Software Systems, Tampere University of Technology. He obtained M.Sc degree in Information Technology from the same department in 2010. His research area includes analysis of open source systems and projects.

Timo Aaltonen is a principal researcher in Nokia Research Center. He got the doctoral degree in 2005 while studying formal methods. After the dissertation he initiated the open source research in the department of software systems at Tampere University of Technology together with Imed Hammouda. Aaltonen managed several open source related research projects during the years. In 2010 Aaltonen left the university and joined Nokia Research Center.

Imed Hammouda is currently an adjunct professor at Tampere University of Technology (TUT) where he is heading the international master's programme at the department of software systems. He got his PhD in software engineering from TUT in 2005. Dr. Hammouda's research interests include open source software, community-driven software development, and software architecture. He is co-leading TUTOpen - TUT research group on open source software. He has been leading and involved in several research projects related to various open initiatives.

Tarja Systä is a professor at Tampere University of Technology, department of software systems. Her current research interests include software maintenance and analysis, software architectures, model-driven software development, and development and management of service-oriented systems.

APPENDIX

Repository Structure

Let, $\rho = \{R^1, R^2, \cdots, R^n\}$ be the set of repositories. Where each repository, $R^i = \{r^i_1, r^i_2, \cdots, r^i_m\}$ consists of repository elements r^i_j. Each repository element $r^i_j = \{c^i_{j1}, c^i_{j2}, \cdots, c^i_{jx}\}$ consists of attributes c^i_{jk} describing it. Each attribute element $c^i_{jk} = \{c^i_{jkp}\}$ consists of a number of attribute values c^i_{jkp} as shown in Figure 2.

For example, consider a set of repositories, $\rho = \{File^F, Person^P, Issue^I\}$ where the $File^F$ repository contains a description of the code files, the $Person^P$ repository contains detail of each person (either user or developer) involved in the project and $Issue^I$ repository contains information about the issues (e.g., reported bugs, feature requests) raised by the project personnel. For this illustration consider only the $File^F$ repository. The repository $File^F = \{file^F_1, file^F_2, \cdots, file^F_m\}$ consists of code files as repository elements. Each code file,

$$File^F_j = \{f^F_{name\,j1}, path^F_{j2}, extension^F_{j3}, developer^F_{j4}, includes^F_{j5}, copyright^F_{j6}\}$$

consists of six attributes referring to the name of the file, its path, extension, a list of the names of developers, included files and copyright information. This repository structure is shown in Figure 3.

The InterConnect Operation

The interConnect operation denoted by $interConnect(R^i, attributeName)$ is performed between the repository elements and corresponding attribute values. Let, $R^1 = \{r^1_1, r^1_2, \cdots, r^1_m\}$ be a repository with its repository elements r^1_j and for each repository element r^1_j, there is an attribute $c^1_{j4} = \{c^1_{j4p}\}$ with its attribute values c^1_{j4p}.

$interConnect\left(R^1, c^1_{j4}\right)$ results in a graph with the following vertex and edge sets.

Vertices: Each r^1_j is added to vertex list $vSetTypeA$ and each attribute value c^1_{j4i} is added to the vertex list $vSetTypeB$ if not already added.

Edges: For each r^1_j in R^1, create edges between r^1_j and each attribute value c^1_{j4i}. Edges are then added to edge list E.

The resultant graph will be a bipartite graph, $G_{\text{vSetTypeA}-\text{vSetTypeB}} = (vSetTypeA \bigcup vSetTypeB, E)$. For example, consider the file repository where $File^F = \{file^F_1, file^F_2, \cdots, file^F_m\}$. For each repository element $File^F_j$, the attribute $developer^F_{j4}$ consists of one or more developer names, $developer^F_{j4} = \{dev^F_{j41}, dev^F_{j42}, \cdots, dev^F_{j4z}$ as attribute values.

Then $interConnect(File^F, developer^F_{j4})$ operation results in a graph with the following vertex and edge sets.

Vertices: $file_j^F$ is added to vertex list *File* and each attribute value dev_{j4i}^F is added to the vertex list *Developer* if not already added.

Edges: For each $file_j^F$ in $File^F$, create edges between $file_j^F$ and each attribute value dev_{j4i}^F. Edges are then added to edge list *E*.

The resultant bipartite graph $G_{File-Developer} = (File \bigcup Developer, E)$ represents a 2-mode SNA network. It portrays the relationship between two sets of vertices *File* and *Developer*. In other words, this graph shows the relationship between the code files in the repository and the developers who are responsible for those code files. Thus, the interconnect operation can be used to generate graphs that reveal how people (i.e., developers or users) in an OSS project are related to different project artifacts (i.e., code files, issues, mail list).

The IntraConnect Operation

The intraConnect operation denoted by $intraConnect(G, relationship\,VertexSet)$ is performed on graphs generated by the interConnect operation. Thus *G* is a 2-mode SNA network and is a bipartite graph. The second argument in this operation denotes one of the two vertices sets in the input bipartite graph. This operation is defined as follows.

Let $G_{\mathrm{vSetTypeA-vSetTypeB}} = \left(vSetTypeA \bigcup vSetTypeB, E\right)$ be a graph generated by the interConnect operation. The vertex set *vSetTypeA* = {vA_1, vA_2, ..., vA_m} and vertex set $vSetTypeB = \{vB_1, vB_2, \cdots, vB_n\}$. Then, $intraConnect\ (G_{vSetTypeA-vSetTypeB},\ vSetTypeA)$ results in a graph $G_{vSetTypeB}^A = (V_{vSetTypeB}^A, E_{vSetTypeB}^A)$ with the following vertex and edge set, $V_{vSetTypeB}^A = \{vB_1, vB_2, \cdots, vB_n\}$ and:

$$E_{vSetTypeB}^A = \left\{\left(vB_j, vB_q\right) | \left(vB_j, vA_i\right) \in E \wedge \left(vB_q, vA_i\right) \in E,\ j \neq q\right\}$$

And the weight for each edge $\left(vB_j, vB_q\right)$ is calculated as follows,

$W\left(vB_j, vB_q\right) = \left|\{vA_i \middle| \left(vB_j, vA_i\right) \in E \wedge \left(vB_q, vA_i\right) \in E, j \neq q\}\right|$, that is the weight for each edge $\left(vB_j, vB_q\right)$ is the number of nodes vA_i in $vSetTypeA$ that are common between vB_j and vB_q in the input graph.

For example, consider that the intraConnect operation will be performed on the graph $G_{\mathrm{File-Developer}} = (File \bigcup Developer, E)$. The vertex set $File^F = \{file_1^F, file_2^F, \cdots, file_m^F\}$ consists of code file nodes, and vertex set $Developer = \{dev_1, dev_2, \cdots, dev_n\}$ consists of developer nodes.

Then, the application of $intraConnect\ (G_{File-Developer}\ , File)$ operation results in a graph $G_{developer}^F = (V_{developer}^F, E_{developer}^F)$ where, $V_{developer}^F = \left\{dev_1, dev_2, \cdots, dev_n\right\}$ and:

$$E_{developer}^F = \{(dev_j, dev_q) \mid (dev_j, file_i) \in E \wedge (dev_q, file_i) \in E, j \neq q\}\ .$$

The resultant graph $G_{developer}^F$ shows the relationship between developer nodes that have edgestoatleastonecommonfilenodeinthe $G_{File-Developer}$ .Inotherwords, $G_{developer}^F = (V_{developer}^F, E_{developer}^F)$ graph shows the relationship between developers who contribute or are responsible for the same code files. Hence, the intraConnect operation generates graphs that show how people (i.e., developers or users) are related to each other through project artifacts (i.e., code files, mailing list, bug repository) and vice versa.

The weight for each edge is calculated as

$$W\left(dev_j, dev_q\right) = \mid \{file_i \mid (dev_j, file_i) \in E \wedge (dev_q, file_i) \in E, j \neq q\} \mid$$

Thus edge weight in the $G^F_{developer}$ graph shows how many code files each pair of developers share in common. This in turn reflects how strongly a group of developers are related to each other and share information and views while developing the project.

Set Operations

In this section, set operations are defined. As set operations are performed on graphs, we will use the following two graphs generated by the *intraConnect* operation,

$G^A_{vSetTypeB} = (V^A_{vSetTypeB}, E^A_{vSetTypeB})$ and $G^C_{vSetTypeB} = (V^C_{vSetTypeB}, E^C_{vSetTypeB})$.

These graphs draw relationship among vertices in $vSetTypeB$ based on vertex sets $vSetTypeA$ and $vSetTypeC$, respectively. As an example consider the following two graphs, $G^F_{developer} = (V^F_{developer}, E^F_{developer})$ and $G^I_{developer} = (V^I_{developer}, E^I_{developer})$. These graphs are generated by the *intraConnect* operation and show the relationships between developers based on common code file sharing and common user issues answered, respectively.

In what follows the definition of set operations is in terms of the above mentioned graphs.

***Intersection*:** The intersection operation denoted by, $intersection(G^A_{vSetTypeB}, G^C_{vSetTypeB})$ keeps only those vertices and edges that are common to both $G^A_{vSetTypeB}$ and $G^C_{vSetTypeB}$ graphs. That is, the resultant graph is $G^{intersect}_{vSetTypeB} = (V_{vSetTypeB}, E_{vSetTypeB})$, where $V_{vSetTypeB} = V^A_{vSetTypeB} \bigcap V^C_{vSetTypeB}$ $E_{vSetTypeB} = E^A_{vSetTypeB} \bigcap E^C_{vSetTypeB}$.

The weights for the edges are calculated as follows: let, $W^A_{vSetTypeB}$ and $W^C_{vSetTypeB}$ be the weight sets for $G^A_{vSetTypeB}$ and $G^C_{vSetTypeB}$ respectively. Then the weight set $W_{vSetTypeB}$ for $G^{intersect}_{vSetTypeB}$ is calculated as $w\left(e_i\right) = \{\left(w\left(e^A_j\right) + w\left(e^C_k\right)\right), e_i = e^A_j = e^C_k\}$ where, $w\left(e^A_j\right) \in W^A_{vSetTypeB}$, $w\left(e^C_k\right) \in$ $W^C_{vSetTypeB}$ and $w\left(e_i\right) \in W_{vSetTypeB}$.

That is, the weight of each edge in the $G^{intersect}_{vSetTypeB}$ graph is the summation of weights of the same edge in the $G^A_{vSetTypeB}$ and $G^C_{vSetTypeB}$ graphs.

A careful look at the input graphs would show that each input graph reveals the relationships among nodes for a specific domain. Thus the weights in the input graph reflect relationship strengths for that domain. The resultant graph considers only those nodes and relationships that are present in both the input graphs (i.e., present in both domains). Thus, the edge weight in this resultant graph must show the overall impact of a relation in both the domains. This is why the weights for the edges in the resultant graph are calculated by adding up weights from both the input graphs.

For example, the $intersection(G^F_{developer}, G^I_{developer})$ operation keeps only those developer vertices and edges that are common to both the $G^F_{developer}$ and $G^I_{developer}$ graphs. That is, the resultant graph is $G^{intersect}_{developer} = (V_{developer}, E_{developer})$, where

$$V_{developer} = V^F_{developer} \bigcap V^I_{developer}$$
$$E_{developer} = E^F_{developer} \bigcap E^I_{developer}.$$

The graph $G^{intersect}_{developer}$ shows those developers and their relationships that are responsible for both code files and solving user issues. Thus, an intersection operation generates graphs showing the multidimensional activities of people (i.e., users and developers) within the OSS project.

For the weight calculation, let $W^F{}_{developer} = \{w\left(e^F_1\right), w\left(e^F_2\right), \cdots, w(e^F_n)\}$ and $W^I{}_{developer} = \{w\left(e^I_1\right), w\left(e^I_2\right), \cdots, w(e^I_m)\}$ be the weights for $G^F_{developer}$ and $G^I_{developer}$ respectively.

Then the weight $W_{developer} = \{w(e_1), w(e_2), \cdots, w(e_z)\}$ for the graph $G^{intersect}_{developer}$ is calculated as

$$w\left(e_i\right) = \{\left(w\left(e^F_j\right) + w\left(e^I_k\right)\right), e_i = e^F_j = e^I_k\}$$

Thus, the weights in $G^{intersect}_{developer}$ show the overall interactions between developers during development of the project as well as in solving issues.

Union: The union operation denoted by $union(G^A_{vSetTypeB}, G^C_{vSetTypeB})$ includes all the nodes and edgesthatarepresentinbothinputgraphs.Thatis,theresultantgraphis $G^{union}_{vSetTypeB} = (V_{vSetTypeB}, E_{vSetTypeB})$ where,

$$V_{vSetTypeB} = V^A_{vSetTypeB} \bigcup V^C_{vSetTypeB},$$
$$E_{vSetTypeB} = E^A_{vSetTypeB} \bigcup E^C_{vSetTypeB}.$$

Thus this graph shows three regions, one that is common to both the input graphs and the two that are present in either one or the other of the input graphs.

To calculate the edge weights, Let $W^A_{vSetTypeB}$ and $W^C_{vSetTypeB}$ be the weight sets for $G^A_{vSetTypeB}$ and $G^C_{vSetTypeB}$ respectively. Then the weight set $W_{vSetTypeB}$ for $G^{union}_{vSetTypeB}$ is calculated as follows,

Weights for the common region of the graph are:

$$w\left(e_i\right) = \{\left(w\left(e^A_j\right) + w\left(e^C_k\right)\right), e_i = e^A_j = e^C_k\}$$

and weights for other two regions are

$w\left(e_i\right) = \{w(e_j) \mid w(e_j) \in W^A_{vSetTypeB} \vee w(e_j) \in W^C_{vSetTypeB}, e_i = e_j\}$, where, $w\left(e^A_j\right) \in W^A_{vSetTypeB}$, $w\left(e^C_k\right) \in W^C_{vSetTypeB}$ and $w\left(e_i\right) \in W_{vSetTypeB}$.

Thus the weights for the common region of the graph are calculated the same way as the intersection operation and have the same rationale as the intersection operation. But the weights for the other two regions of the graph retain their source graph weights. Because each of these regions represents the relationships for a specific domain the weight should reflect the relationship strength for that domain.

Now, applying the union operation to our example graphs would generate a graph $G_{developer}^{union} = (V_{developer}, E_{developer})$, where

$$V_{developer} = V_{developer}^{F} \bigcup V_{developer}^{I},$$
$$E_{developer} = E_{developer}^{F} \bigcup E_{developer}^{I}.$$

This graph $G_{developer}^{union}$ shows three regions within the developer community. The common region shows group of developers who are interested in playing both roles (as developers and issue solvers) and the other two regions show groups of developers who either work as developers or take part only as bug or issue solvers. This graph also shows the relationships among these groups of people within the developer community. Thus, a comprehensive illustration of people's activity, interest area and contribution within the community can be visualized through this operation.

For calculating the weights, let $W^{F}{}_{developer} = \{w\left(e_1^F\right), w\left(e_2^F\right), \cdots, w(e_n^F)\}$ and $W^{I}{}_{developer} = \{w\left(e_1^I\right), w\left(e_2^I\right), \cdots, w(e_m^I)\}$ be the weights for $G_{developer}^{F}$ and $G_{developer}^{I}$ respectively.

Then the weight $W_{developer} = \{w(e_1), w(e_2), \cdots, w(e_z)\}$ for the graph $G_{developer}^{union}$ can be calculated as follows. The weights for the common region of the graph are

$w\left(e_i\right) = \{\left(w\left(e_j^F\right) + w\left(e_k^I\right)\right), e_i = e_j^F = e_k^I\}$ and the weights for the other regions are

$$w\left(e_i\right) = \{w(e_j) \mid w(e_j) \in W_{developer}^{F} \vee w(e_j) \in W_{developer}^{I}, e_i = e_j\}$$

That is, the edge weight for the common region is calculated as the sum of their source weights. Therefore this would present the overall strength of the relationship between developers performing both roles (as developer and as issue solver). However, the edges that are from either of the input graphs retain their source weights, showing their communication frequency either as developers or as bug solvers.

Difference: The difference operation denoted by, $difference(G_{vSetTypeB}^{A}, G_{vSetTypeB}^{C})$ results in a graph which will contain only those nodes and edges that are present in $G_{vSetTypeB}^{A}$ but not in $G_{vSetTypeB}^{C}$. That is the resultant graph is $G_{vSetTypeB}^{diff} = (V_{vSetTypeB}, E_{vSetTypeB})$, where

$$V_{vSetTypeB} = V_{vSetTypeB}^{A} - V_{vSetTypeB}^{C} = \{v_{vSetTypeB} \mid (v_{vSetTypeB} \in V_{vSetTypeB}^{A}) \otimes (v_{vSetTypeB} \notin V_{vSetTypeB}^{C})\}$$

and

$$E_{vSetTypeB} = E_{vSetTypeB}^{A} - E_{vSetTypeB}^{C} = \{e_{vSetTypeB} \mid (e_{vSetTypeB} \in E_{vSetTypeB}^{A}) \otimes (e_{vSetTypeB} \notin E_{vSetTypeB}^{C})\}.$$

Edge weight is calculated as follows. Let $W_{vSetTypeB}^{A}$ and $W_{vSetTypeB}^{C}$ be the weight sets for $G_{vSetTypeB}^{A}$ and $G_{vSetTypeB}^{C}$ respectively. Then the weight set $W_{vSetTypeB}$ for $G_{vSetTypeB}^{diff}$ would be,

$w(e_i) = \{w(e_j) \mid (w(e_j) \in W^{A}_{vSetTypeB}), e_i = e_j\}$ where $w(e_i) \in W_{vSetTypeB}$. That is, the weight of each edge in the $G^{diff}_{vSetTypeB}$ graph retains its source graph weight. The reason is obvious. The resultant graph contains edges from one of the input graphs which are not present in the other.

Thus they should retain their source weights.

For example, the *difference*$(G^{F}_{developer}, G^{I}_{developer})$ operation would result in the $G^{diff}_{developer} = V$ $(V_{developer}, E_{developer})$ graph, where:

$$V_{developer} = V^{F}_{developer} - V^{I}_{developer} = \{v_{developer} \mid (v_{developer} \in V^{F}_{developer}) \otimes (v_{developer} \notin V^{I}_{developer})\} \text{ and}$$

$$E_{developer} = E^{F}_{developer} - E^{I}_{developer} = \{e_{developer} \mid (e_{developer} \in E^{F}_{developer}) \otimes (e_{developer} \notin E^{I}_{developer})\}.$$

Thus the graph $G^{diff}_{developer}$ contains only those nodes and edges that are in $G^{F}_{developer}$, excluding all the nodes and edges that are common between the two input graphs or in the $G^{I}_{developer}$ graph. In other words, this graph shows the group of people within the developer community who only concentrate on development.

For the weight calculation, let $W^{F}_{developer} = \{w(e^{F}_1), w(e^{F}_2), \cdots, w(e^{F}_n)\}$ and $W^{I}_{developer} = \{w(e^{I}_1), w(e^{I}_2), \cdots, w(e^{I}_m)\}$ be the weights for $G^{F}_{developer}$ and $G^{I}_{developer}$ respectively.

Then the weight $W_{developer} = \{w(e_1), w(e_2), \cdots, w(e_z)\}$ for the graph $G^{diff}_{developer}$ can be calculated as $w(e_i) = \{w(e_j) \mid (w(e_j) \in W^{F}_{developer}), e_i = e_j\}$. That is, the weight of each edge is taken from source graph $G^{F}_{developer}$. These weights show how strongly the developers are tied together during development.

Symmetric Difference: The Symmetric Difference operation denoted by:

$$symmDifference(G^{A}_{vSetTypeB}, G^{C}_{vSetTypeB})$$

results in a graph which contains all the nodes and edges that are either in $G^{A}_{vSetTypeB}$ or in $G^{C}_{vSetTypeB}$, excluding the common nodes and edges. That is, the resultant graph is $G^{symmDiff}_{vSetTypeB} = (V_{vSetTypeB}, E_{vSetTypeB})$, where

$$V_{vSetTypeB} = V^{A}_{vSetTypeB} \otimes V^{C}_{vSetTypeB} = \{v_{vSetTypeB} \mid (v_{vSetTypeB} \in V^{A}_{vSetTypeB}) \otimes (v_{vSetTypeB} \notin V^{C}_{vSetTypeB})\}$$

and

$$E_{vSetTypeB} = E^{A}_{vSetTypeB} \otimes E^{C}_{vSetTypeB} = \{e_{vSetTypeB} \mid (e_{vSetTypeB} \in E^{A}_{vSetTypeB}) \otimes (e_{vSetTypeB} \notin E^{C}_{vSetTypeB})\}.$$

And the weights for the edges are calculated as:

$$w(e_i) = \{w(e_j) \mid (w(e_j) \in W^{A}_{vSetTypeB}) \vee (w(e_j) \in W^{C}_{vSetTypeB}), e_i = e_j\} \text{ where } w(e_i) \in W_{vSetTypeB}.$$

So, the edges in $G^{symmDiff}_{vSetTypeB}$ retain the source weights to which they actually belong because the edges in this graph are from either of the input graphs which in turn represent distinct domain of activities. Thus the edge weights should also reflect the relationship strength for that particular domain.

For example, the $symmDifference(G^{F}_{developer}, G^{I}_{developer})$ operation would result in $G^{symmDiff}_{developer} = (V_{developer}, E_{developer})$ graph, where:

$V_{developer} = V^{F}_{developer} \otimes V^{I}_{developer} = \{v_{developer} \mid (v_{developer} \in V^{F}_{developer}) \otimes (v_{developer} \notin V^{I}_{developer})\}$ and

$E_{developer} = E^{F}_{developer} \otimes E^{I}_{developer} = \{e_{developer} \mid (e_{developer} \in E^{F}_{developer}) \otimes (e_{developer} \notin E^{I}_{developer})\}$.

Thus, for this example, graph $G^{symmDiff}_{developer}$ shows two groups within the developer community, one performing as developers and the other only dealing with bugs or issues. Hence, this operation can be used to find out specialized groups within the community.

For calculating the weights Let, $W^{F}_{developer} = \{w\left(e^{F}_{1}\right), w\left(e^{F}_{2}\right), \cdots, w(e^{F}_{n})\}$ and:

$W^{I}_{developer} = \{w\left(e^{I}_{1}\right), w\left(e^{I}_{2}\right), \cdots, w(e^{I}_{m})\}$ be the weights for $G^{F}_{developer}$ and $G^{I}_{developer}$ respectively.

Then the weight $W_{developer} = \{w(e_1), w(e_2), \cdots, w(e_z)\}$ for the graph $G^{symmDiff}_{developer}$ would be, $w\left(e_i\right) = \{w(e_j) \mid (w(e_j) \in W^{F}_{vSetTypeB}) \vee (w(e_j) \in W^{I}_{vSetTypeB}), e_i = e_j\}$ .Thatis,theedgesin $G^{symmDiff}_{developer}$ retain their source weights.

[III] M.M. Syeed, and I. Hammouda. Socio-technical Congruence in OSS Projects: Exploring Conway's Law in FreeBSD. In *Proceedings of the 9th IFIP WG 2.13 International Conference of Open Source Systems (OSS'2013)*, pages 109–126. Springer, June, 2013.

Socio-technical Congruence in OSS Projects: Exploring Conway's Law in FreeBSD

M.M. Mahbubul Syeed and Imed Hammouda

Tampere University of Technology, Finland
{mm.syeed,imed.hammouda}@tut.fi

Abstract. Software development requires effective communication, coordination and collaboration among developers working on interdependent modules of the same project. The need for coordination is even more evident in open source projects where development is often more dispersed and distributed. In this paper, we study the match between the coordination needs established by the technical domain (i.e. source code) and the actual coordination activities carried out by the development team, such hypothetical match is also known as socio-technical congruence. We carry out our study by empirically examining Conway's law in FreeBSD project. Our study shows that the congruence measure is significantly high in FreeBSD and that the congruence value remains stable as the project matured.

1 Introduction

Investigating socio-technical congruence in software development projects has become an active research area in the last decade [10]. Socio-technical congruence can be defined as the match between the coordination needs established by the technical domain (i.e., the architectural dependency in the software) and the actual coordination activities carried out by project members (i.e., within the members of the development team) [6]. This coordination need can be determined by analyzing the assignments of people to a technical entity such as a source code module, and the technical dependencies among the technical entities [6]. Socio-technical congruence not only has been used as a measure for a number of project properties such as software build success [8] but also as a means for software engineering tasks like architecture recovery [23].

In most of research on socio-technical congruence, Conway's law [4] is often presented as a guide and a basis for the underlying study. Conway's Law in its purest form states that "the organizations that design systems are constrained to produce systems which are copies of the communication structures of these organizations" [1]. In other words, the product architecture often reflects the organizational structure of the development team [1][6]. In [7], Conway's law is considered homomorphic and thus claimed to be true in reverse as well. This means the communication pattern within developer community should reflect the architectural dependency in the software. Thus, the notion of socio-technical congruence is actually a conceptualization of Conway's law.

Studying socio-technical congruence in open source software has its own questions and hypotheses, for those products are peer-produced, crowdsourced systems where centralized planning and control of architecture is difficult. Furthermore, the evolution of such systems, and their underlying architecture, is not bound to any pre-defined plans [5]. Surprisingly, this research area has not been given much attention among open source researchers [34].

In this paper, we study Conway's Law and Reverse Conway's Law, as a lens for socio-technical congruence, in the context of open source development projects by proposing a novel evaluation and measurement technique. We further explore the significance of socio-technical congruence as the project matures. To investigate our research problem, we focused on a popular open source project, FreeBSD, which is an advanced operating systems for computing platforms. We carried out our study empirically by analyzing the software repository of the project in a semi-automatic way.

The remaining of the paper is organized as follows. Section 2 introduces a number of key concepts that this study uses. Motivation and related work is presented in Section 3. We then introduce the research questions explored in this paper and our study design in Section 4 and 5 respectively. Results are reported and discussed in Section 6. Possible limitations and threats to validity are highlighted in Section 7. Finally, Section 8 concludes the papers and shed light on future research.

2 Definitions

In this paper we interpreted Conway's law (and reverse Conway's law) as a measure to verify the extent to which the communication pattern of the contributing members is due to the communication needs established by the concrete architecture and vice versa. In examining this the following concepts are defined.

2.1 Developer Contribution

Developer contribution to the software can be defined as the code contribution or any form of commit made to the code base.

2.2 Concrete Architecture

Concrete architecture of a software presents the relationship among components of a software (e.g., modules, files or packages) based on the actual design and implementation. In this work, header file inclusions dependency at code file level were used to derive the concrete architecture. In this architecture two files, and their corresponding packages, were linked if the two files have an inclusion dependency.

2.3 Concrete Coordination Network

Concrete Coordination Network is a social network in which two developers have a relationship if they have communication history as seen by the mailing archives representing the social and technical interactions among the developers.

2.4 Derived Architecture

Derived architecture defines an architecture of the software where any two components (e.g., packages or code files) are related if there are developer(s) who have either (a) contributed to both the components, or (b) have communication at organizational level (e.g., through email). For instance consider that developer D1 has contributed to packages P1 and P2, and developer D2 has contributed to package P3. Also consider that both developers has communication at organizational level as shown in Fig. 1(a). Thus according to the definition, packages P1, P2 and P3 are linked to each others in the Derived Architecture (Fig. 1(b)).

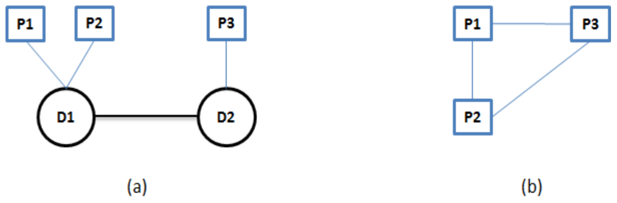


Fig. 1. (a) Concrete Coordination Network with contribution to code base (b) Corresponding Derived Architecture

2.5 Derived Coordination Network

Derived Coordination Network is the developer relationship network in which two developers have a relationship if they have contributed either (a) to a common code file or (b) to the code files that have relationships in the concrete architecture. For instance consider that developer D1 and D2 has contributed to package P1 and developer D3 has contributed to package P2. Also consider that P1 and P2 have an inclusion dependency as shown in Fig.2(a). Then, according to the definition developers D1, D2 and D3 will have relationships in Derived Coordination Network as shown in Fig. 2(b).

3 Motivation and Related Work

In this section we discuss the significance of Conway's law and socio-technical congruence in the field of software engineering citing related works from existing literature.

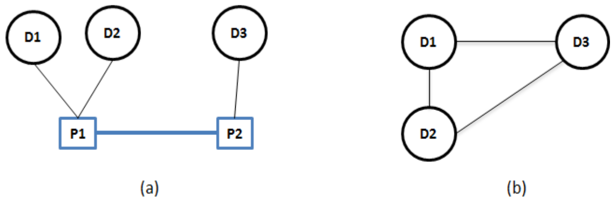


Fig. 2. (a) Concrete architecture with contributing developers (b) Corresponding Derived Coordination Network

3.1 Need for Conway's Law and Socio-technical Congruence

Conway's law was stated 30 years ago by Melvin Conway as the means to emphasize the need of coordination in software development [1]. Until then researchers have long argued that the mirroring effect of the software architecture and the communication pattern of the developer community plays a pivotal role in coordinating the development work [9]. A central question arises "Does Conway's law matter in the modern era of software development?". As reported in [10], Conway's law still matters in the domain of software development and the product quality is strongly affected by the organizational structure [11]. Also with the advent of global software engineering where development teams are distributed across the world, the impact of organizational structure on Conways law and its implications on quality is significant [12].

In the domain of software engineering, socio-technical congruence as a conceptualization of Conway's law was first coined in [13]. In this paper, socio-technical congruence was employed as a fine-grained measure of coordination that can be used to diagnose coordination problems in the development team. Also, evidence from studies [13] shows that higher congruence leads to faster completion of modification requests. This concept has been identified as an important element for product design in the field of engineering [14] and management science [15] as well. Researchers also argued that such congruence is a natural consequence and a desired property for collaborative development activities [14], such as software engineering.

3.2 Socio-technical Congruence and OSS Development

In the epoch of OSS projects, there exists a significant number of successful software systems whose volume and complexity are as compound as that of their proprietary counterparts. Such large and successful OSS projects often consist of hundreds and even thousands of developers contributing to the development of the project. Developers working in such projects are not strictly bound to any organizational rules, regulation and structure. Rather they voluntarily join and contribute to the project. OSS developers belong to discrete geographical locations of significant background, timezone, language and cultural distances. Often an OSS project is developed and evolved through the collaboration among the developers using simple communication media like email, wiki, chat [31] as well as revision tracking systems (such as CVS, SVN) to store and access the software code they produce [19].

This unconventional organizational structure and practices of OSS projects combing with the inherent complexity of the software development (as discussed above) brings forth the question, "why such projects succeed and can socio-technical congruence be conceived as an implicit driving force for such success?". Surprisingly, socio-technical congruence as a research area has not been given much attention among open source researchers. In this paper, we infer that socio-technical congruence is an endogenous driver for successful OSS projects, and examines its existence in OSS project though empirical evidences.

4 Research Questions

Our research objective in this paper is to study the applicability of Conway's law as a means of verifying the socio-technical congruence on large, distributed and evolving OSS projects. In great part because of the specific characteristics of the OSS projects and yet having revolutionary success, it is most likely that the OSS development process implicitly encompasses the notion of socio-technical congruence. Thus, we targeted the following research questions.

(a) How does the communication patterns of OSS developer community resemble the architecture of the software?
Conway's law can be interpreted as the first explicit recognition that the communication pattern of the organization has an inevitable impact on the product [20]. Thus there must exist a correspondence between the community structure and the software structure. Our point of interest here is that if such homomorphic force between the software model and its development organization exists in OSS projects, then it could be effectively used to conceptualize the architecture of legacy systems. It can also be used as a metric to assess the quality and maintainability of the system as well.

(b) How does the architecture of the software resemble the collaboration patterns of OSS developer community?
Conway's law pointed out [1] that only the organizational arrangements can be optimized with respect to the system concept. This observation enforces the need for a stable and modular design of the system in order to facilitate effective communication, coordination among the developers. With geographically distributed development setup, as in OSS projects, co-ordination becomes more challenging due to location, time, and cultural differences, and the need for a stable design becomes even more evident [9]. For this kind of organizational setup, a clearly separable and stable architectural design would be the basis for assigning task to the developers [9] and a key to the success of the project. In other words, the collaboration pattern of the developer community should mirror the architecture of the software which should remain stable during the evolution of the project.

(c) Does the socio-technical congruence evolve as the project matures?
Our intension is to examine how the socio-technical congruence evolves as the project matures. We are particularly interested in deriving any visible pattern of such congruence during the evolution. But relating such trends in congruence in OSS projects with its success parameters are out of the scope of this study.

5 Study Design

5.1 Case and Subject Selection

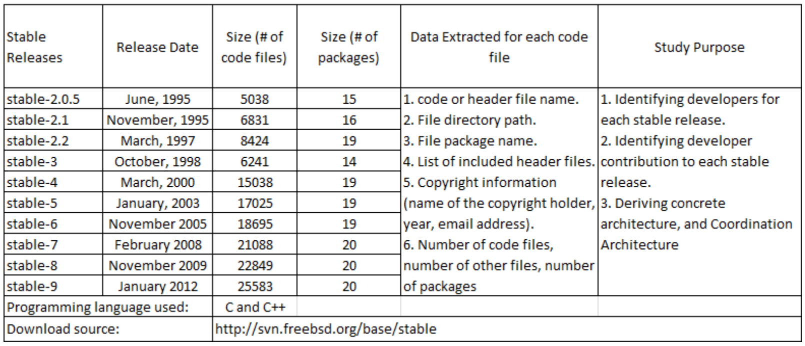
To investigate our research questions, we focused on a popular open source project, FreeBSD, which is an advanced operating system for modern server, desktop, and embedded computer platforms [24]. It is derived from BSD, the version of UNIX developed at the University of California, Berkeley. Selection of this project as a case study was influenced by the facts that FreeBSD's code base has undergone continuous development, improvement, and optimization for thirty years [24]. The project has been developed and maintained by a large team of individuals. Also, FreeBSD has gained considerable attention in earlier research on the evolution of OSS projects [16] [17] [18].

5.2 Data Sources

In literature [10] a great emphasis was given to leveraging software repositories along with the communication data for deriving technical dependencies as well as developers coordination patterns. In OSS projects, repository data is stored and maintained through different data management systems, e.g., source code repository (SVN or CVS), change logs, mailing archive, bug reporting systems, and communication channels. These data sources are highly accepted and utilized medium for empirical studies on OSS projects [25][26][27]. In FreeBSD project, the development and communication history is maintained through SVN repositories and mailing archives. Following is the description of these sources and the data extracted for this study.

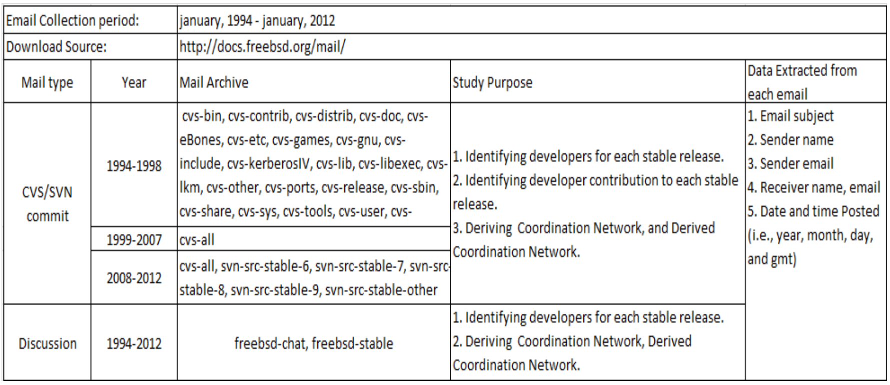
1. Source code repository: FreeBSD has two release branches, stable and production releases. In this study, SVN repositories of the stable releases were collected. Fig. 3 provides detail of these stable releases and the data collected from each release.
2. Mailing list archive: In OSS projects, email archives provide a useful trace of the task-oriented communication and co-ordination activities of the developers during the evolution of the project [28]. In the FreeBSD project, email archives are categorized according to their purpose. For instance, there are email for project development (e.g., commits), stable release planning, chat, user emails, bug reports. These archives contain all mailing lists since 1994 and are updated every week [24]. Fig. 4 provides detail on the FreeBSD email archives that were extracted for this study. These email archives contain the

detail of the commit records made for each stable release and are only used by the developer community. These archives thus give more accurate history of contributions than the bug reporting system which is often used by the the passive users for different purposes.



Stable Releases	Release Date	Size (# of code files)	Size (# of packages)	Data Extracted for each code file	Study Purpose
stable-2.0.5	June, 1995	5038	15	1. code or header file name. 2. File directory path. 3. File package name. 4. List of included header files. 5. Copyright information (name of the copyright holder, year, email address). 6. Number of code files, number of other files, number of packages	1. Identifying developers for each stable release. 2. Identifying developer contribution to each stable release. 3. Deriving concrete architecture, and Coordination Architecture
stable-2.1	November, 1995	6831	16		
stable-2.2	March, 1997	8424	19		
stable-3	October, 1998	6241	14		
stable-4	March, 2000	15038	19		
stable-5	January, 2003	17025	19		
stable-6	November 2005	18695	19		
stable-7	February 2008	21088	20		
stable-8	November 2009	22849	20		
stable-9	January 2012	25583	20		
Programming language used:		C and C++			
Download source:		http://svn.freebsd.org/base/stable			

Fig. 3. Stable Releases of FreeBSD



Email Collection period:		january, 1994 - january, 2012		
Download Source:		http://docs.freebsd.org/mail/		
Mail type	Year	Mail Archive	Study Purpose	Data Extracted from each email
CVS/SVN commit	1994-1998	cvs-bin, cvs-contrib, cvs-distrib, cvs-doc, cvs-eBones, cvs-etc, cvs-games, cvs-gnu, cvs-include, cvs-kerberosIV, cvs-lib, cvs-libexec, cvs-lkm, cvs-other, cvs-ports, cvs-release, cvs-sbin, cvs-share, cvs-sys, cvs-tools, cvs-user, cvs-	1. Identifying developers for each stable release. 2. Identifying developer contribution to each stable release. 3. Deriving Coordination Network, and Derived Coordination Network.	1. Email subject 2. Sender name 3. Sender email 4. Receiver name, email 5. Date and time Posted (i.e., year, month, day, and gmt)
	1999-2007	cvs-all		
	2008-2012	cvs-all, svn-src-stable-6, svn-src-stable-7, svn-src-stable-8, svn-src-stable-9, svn-src-stable-other		
Discussion	1994-2012	freebsd-chat, freebsd-stable	1. Identifying developers for each stable release. 2. Deriving Coordination Network, Derived Coordination Network.	

Fig. 4. FreeBSD email archives used for this study

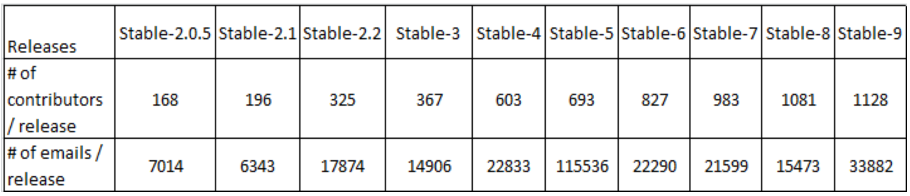
5.3 Data Collection Procedure

Data Collection from Source Code Repositories: Each stable release of the FreeBSD project was downloaded from the SVN repository to the local directory. Fig. 3 lists the stable releases and associated detail of each release. To extract data from each of these releases, a parser was written in Java. This parser searched through each directory of a stable release, read through the files in a directory and parsed relevant data. The data that were parsed from each file were listed in Fig. 3. As FreeBSD was written in C and C++, included header files were identified from the #include directive in each code file. The parser excluded all the library header files and kept only the user defined header files. Also the copyright directive in a file contains information of the developer name, email and the copyright year. The parser extracted each of these information from the copyright directive. The developers that were found in this process were considered as the contributors to that file for that stable release.

The parsed data for each stable release was stored in excel files. To read/write excel files Apache POI [29] was used. The number of code files and packages read from each stable release was shown in Fig. 3.

Data Collection from Email Archives: The email archives that concern CVS/SVN commits and general discussions (e.g., on stable releases and the chat entries) were extracted from FreeBSD email archive as shown in Fig. 4. For extracting data from each email entry, a data extraction program was written in Java. This data extractor used the web interface of the email archives (link is provided in Fig. 4). Thus each email was read as an HTML page and the data was extracted using Jsoup html parser [30]. Data that was extracted from each email entry is listed in Fig. 4. This data was then stored in excel files according to the archive name and year. Then the email data was sorted according to each stable release as follows: (a) emails were categorized into a specific release if the release number was mentioned in email subject (e.g., SVN commit emails provide release number in email subject) and (b) other emails (e.g., freeBSD-stable, freeBSD-chat and most of the CVS commit emails) for which the release numbers were not mentioned, the posting dates were checked. In this case, for instance, an email was categorized to stable release 3 if its posting date fall between the release date of stable release 2 and 3. The rationale here is that developers would commit to the code base and discuss on its release strategy before it is officially released.

After categorizing emails to each stable release, the subject of a CVS/SVN commit email was parsed. This subject mentioned the path to the repository to which the commit was made. From this subject, the directory path, package name, and if provided, the name of the modified code file and the stable release number were identified. Sender name for each of these CVS/SVN commit email was considered as a contributor to the code base for that release. Contributors found in this process were combined with the contributors found from the code base to get list of developers who contributed to each stable release. Fig. 5 lists release wise distribution of the number of developers and email entries identified through this process.



Releases	Stable-2.0.5	Stable-2.1	Stable-2.2	Stable-3	Stable-4	Stable-5	Stable-6	Stable-7	Stable-8	Stable-9
# of contributors / release	168	196	325	367	603	693	827	983	1081	1128
# of emails / release	7014	6343	17874	14906	22833	115536	22290	21599	15473	33882

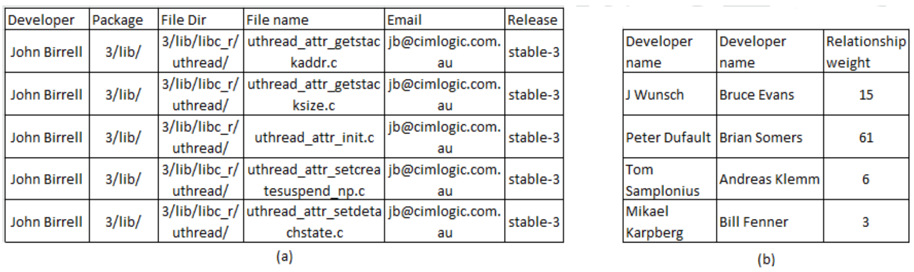
Fig. 5. Number of contributors and emails identified for stable releases

Data preprocessing: Data that was extracted and parsed following the above process contained anomalies data in many cases. For instance, developer names and email addresses might contain punctuation characters like, semi-colons, inverted comas, brackets, unnecessary white space, and hyphens. Furthermore, parsers may parsed data inappropriately in some cases. For example, copyright text "All rights reserved" can be treated as part of developer name while parsing copyright directive from a code file. To clean such anomalies data and punctuation characters, data cleaning programs were written in Java. To ensure the correctness of this process, we performed a manual checking on a randomly selected data to verify their correctness.

5.4 Analysis Procedure

This section discussed the methods used to construct the communication networks, architectures and to measure socio-technical congruence (defined in section 2) utilizing the data collected from the FreeBSD project.

Developer Contribution: Release-wise developer contribution was measured in two ways, (a) from the copyright information provided in each source code file of a release and (b) from the commits made by a developer for a release. Fig. 6(a) shows a sample contribution made by developer *John Birrell* in stable release 3.



Developer	Package	File Dir	File name	Email	Release
John Birrell	3/lib/	3/lib/libc_r/uthread/	uthread_attr_getstackaddr.c	jb@cimlogic.com.au	stable-3
John Birrell	3/lib/	3/lib/libc_r/uthread/	uthread_attr_getstacksize.c	jb@cimlogic.com.au	stable-3
John Birrell	3/lib/	3/lib/libc_r/uthread/	uthread_attr_init.c	jb@cimlogic.com.au	stable-3
John Birrell	3/lib/	3/lib/libc_r/uthread/	uthread_attr_setcreatesuspend_np.c	jb@cimlogic.com.au	stable-3
John Birrell	3/lib/	3/lib/libc_r/uthread/	uthread_attr_setdetachstate.c	jb@cimlogic.com.au	stable-3

(a)

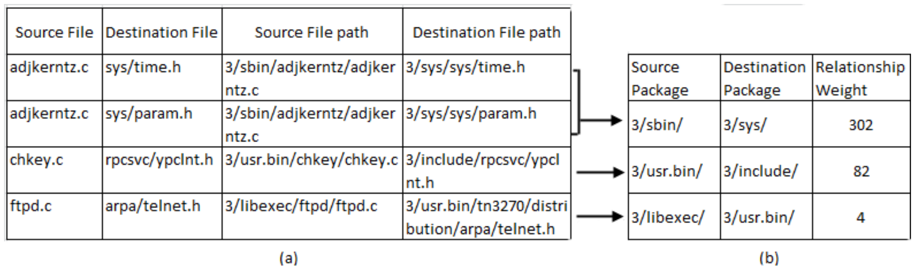
Developer name	Developer name	Relationship weight
J Wunsch	Bruce Evans	15
Peter Dufault	Brian Somers	61
Tom Samplonius	Andreas Klemm	6
Mikael Karpberg	Bill Fenner	3

(b)

Fig. 6. (a) Sample contributions made by developer John Birrell (b) Sample relationships in Concrete Coordination Network

Concrete Architecture: The concrete architecture of a stable release was constructed based on header file inclusion dependency as defined in section 2.2. This inclusion dependency relation was used in earlier works [32] [35] to construct architecture of legacy C/C++ software systems.

Then higher level abstraction of this architecture was built to get the package level concrete architecture. In this architecture, package p1 and p2 had a relationship if file f1 in package p1 had an inclusion dependency with a file f2 in package p2 or vice-versa. Relationship between packages were weighted which was the total number of inclusion relationships that exists between the files in two packages. An example of file level concrete architecture and corresponding package level architecture of stable release 3 is shown in Fig. 7(a) and (b) respectively. Package level concrete architecture was constructed for each stable release.



(a)

Source File	Destination File	Source File path	Destination File path
adjkerntz.c	sys/time.h	3/sbin/adjkerntz/adjkerntz.c	3/sys/sys/time.h
adjkerntz.c	sys/param.h	3/sbin/adjkerntz/adjkerntz.c	3/sys/sys/param.h
chkey.c	rpcsvc/ypclnt.h	3/usr.bin/chkey/chkey.c	3/include/rpcsvc/ypclnt.h
ftpd.c	arpa/telnet.h	3/libexec/ftpd/ftpd.c	3/usr.bin/tn3270/distribution/arpa/telnet.h

(b)

Source Package	Destination Package	Relationship Weight
3/sbin/	3/sys/	302
3/usr.bin/	3/include/	82
3/libexec/	3/usr.bin/	4

Fig. 7. (a) Code file level Concrete Architecture (b) Package level Concrete Architecture

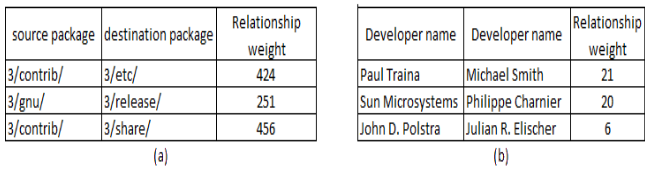
Concrete Coordination Network: Common email conversation in FreeBSD can appear in any of the email archives listed in Fig. 4. This relationship was weighted, meaning that for each new instance of email conversation between the same developers, the relationship weight would be increased by one. For each stable release of FreeBSD, one such Concrete Coordination Network was constructed. Fig. 6(b) shows example relationships in the Concrete Coordination Network of stable release 3. Weight column in this figure shows the number of emails common between the two developers.

Derived Architecture: Derived Architecture was generated based on the definition in Section 2.4. Each package relationship in this architecture was weighted and would increase if either of the two criteria hold for other developers. Both package level and code file leave derived architectures were constructed for each stable release. Fig. 8(a) shows the package level Derived Architecture for stable release 3.

Derived Coordination Network: Derived Coordination Network was generated based on the package level Concrete Architecture and for each stable release following the definition presented in Section 2.5. Each relationship was weighted. Fig. 8(b) shows an example of this network for stable release 3.

Thus this network shows the actual communication need among the developers which is based on the design of the software (i.e., the concrete architecture). This network is essential due to the fact that if two subsystems are to communicate, it is likely to have communication between the developers of the two subsystems [1].

Socio-technical Congruence: To measure socio-technical congruence the following method was applied: common relationships are identified that exists between (a) Concrete Architecture and Derived Architecture, and between (b)



source package	destination package	Relationship weight
3/contrib/	3/etc/	424
3/gnu/	3/release/	251
3/contrib/	3/share/	456

(a)

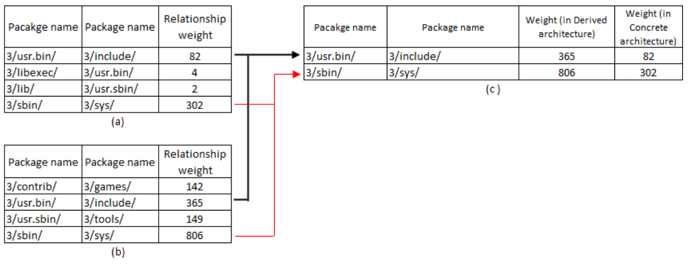
Developer name	Developer name	Relationship weight
Paul Traina	Michael Smith	21
Sun Microsystems	Philippe Charnier	20
John D. Polstra	Julian R. Elischer	6

(b)

Fig. 8. (a) Derived Architecture (b) Derived Coordination Network

Concrete Coordination Network and Derived Coordination Network. The two sets of relationships identified in this process is termed as congruence.

The former congruence illustrates the match between the architectural dependency and the architecture produced due to the communication structure of the community. And the latter congruence depicts the match between the actual coordination activities in the community and the coordination need established by the architectural dependency of the software. To be precise, these congruences verify Conway's law and the reverse Conway's law, respectively. Both form of congruences were determined for each stable release. A partial snapshot of the congruence between Concrete and Derived Architectures of stable release 3 is shown in Fig. 9.



Pacakge name	Package name	Relationship weight
3/usr.bin/	3/include/	82
3/libexec/	3/usr.bin/	4
3/lib/	3/usr.sbin/	2
3/sbin/	3/sys/	302

(a)

Package name	Package name	Relationship weight
3/contrib/	3/games/	142
3/usr.bin/	3/include/	365
3/usr.sbin/	3/tools/	149
3/sbin/	3/sys/	806

(b)

Pacakge name	Package name	Weight (in Derived architecture)	Weight (in Concrete architecture)
3/usr.bin/	3/include/	365	82
3/sbin/	3/sys/	806	302

(c)

Fig. 9. (a) Concrete Architecture (b) Derived Architecture (c) Congruence

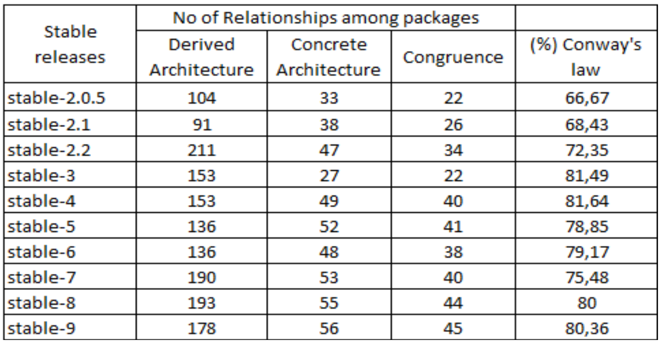
To construct the networks and architectures and to measure socio-technical congruence and Conway's law, Java programs were written.

Result Interpretation: To measure the extent to which Conway's law holds for each stable release of the FreeBSD project, percentile value was calculated for (a) the congruence value and the corresponding Concrete Architecture, and (b) the congruence value and corresponding Coordination Network. These percentile values were plotted in a graph against the stable releases to conceptualize the evolution of the Conway's law and socio-technical congruence in the FreeBSD project.

6 Results

6.1 Resemblance of Communication Pattern to Software Architecture

Our study target in this work was to verify the extent to which the communication patterns of the members of the developer community resembles the actual architectural dependencies. To achieve this, we collected historical data from the FreeBSD project and measured such resemblance for each stable release. The resemblance process consists of determining the concrete architecture and derived architecture for each stable release. Then, the congruence between the two architectures and corresponding percentile measure of Conway's law for each release was measured. The result of this process is reported in Fig. 10. Column 2 and 3 in this figure show the number of relations between packages identified by the respective architectures. The congruence relationships and the percentile value for each release indicate the extent to which the two architectures of the software overlap with each other. Here the percentile value is measured between the congruence and the concrete architecture, because the intension is to measure the extent to which the derived architecture approximates the concrete architecture.



Stable releases	No of Relationships among packages			
	Derived Architecture	Concrete Architecture	Congruence	(%) Conway's law
stable-2.0.5	104	33	22	66,67
stable-2.1	91	38	26	68,43
stable-2.2	211	47	34	72,35
stable-3	153	27	22	81,49
stable-4	153	49	40	81,64
stable-5	136	52	41	78,85
stable-6	136	48	38	79,17
stable-7	190	53	40	75,48
stable-8	193	55	44	80
stable-9	178	56	45	80,36

Fig. 10. Resemblance of communication pattern to software architecture

The percentile values suggest that around 76% to 82% overlap between the two architecutres are found from stable release 3 onward. Whereas for the earlier three releases it is between 66% to 73%. It is worth to mention here that we noted considerable drift in the congruence value for the first three releases of the FreeBSD project. Thus we excluded them from our observation, considering this period as a restructuring and reformation period of FreeBSD after being forked. However, this observation is a positive sign towards the socio-technical congrucence, and can be an implicit charactersitcs of OSS development process. From this result, we can safely say that to a considerable extent the communication of the contributing developers in the community is actually due to the coordination need as identified by the architectural dependency.

Now the question is, does the derived architecture actually resembles the concrete architecture, and can it be used to recover the architecture of the legacy software? By examining Fig. 10 it is evident that the derived architecture is overestimating the concrete architecture for all the releases. For instance, in stable release 4, the derived architecture identifies 153 relations among the 19 packages, whereas concrete architecture shows only 49. Thus, it is not possible to resemble the concrete architecture from the derived architecture. Yet, the following observation can be offered.

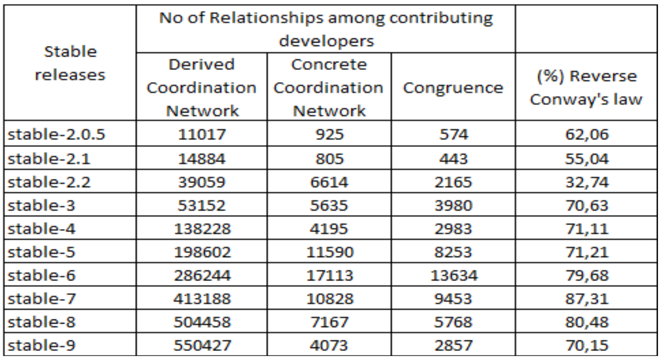
To a great part, the communication pattern of the contributing members within the community is due to the communication needs established by the concrete architecture.

The interdependency among the packages in the concrete architecture influences their contributors to communicate at community level. This supports the existence of Conway's law within the FreeBSD development process. None-theless, both the architectures overlap considerably. Thus the derived architecture can be used in authenticating the architecture recovered by traditional reverse engineering process.

This over estimation by the derived architecture can be justified in a way that this architecture was derived from the communication record of their contributing developers. And the developers in the community can communicate and collaborate on issues that might be outside of the scope of the actual code implementation and commits. For instance, FreeBSD-chat archive collects only the email threads related to the general discussion. These discussions generate additional relationships among the contributing developers which were not due to only common contribution. In the derived architecture these relationships create additional links between the packages. Yet it can also be possible that some of the links between packages in the derived architecture reflect valid relationships for the release, as the concrete architecture might not identify all the links that actually exists for that release.

6.2 Resemblance of Software Architecture to Communication Pattern

Conventional wisdom supports that the socio-technical congruence measured in reverse (i.e., reverse Conway's law) should hold as well. That is the communication pattern identified by the concrete architecture should simulate the actual communication pattern of the contributing community members. To measure this we constructed the coordination network and the derived coordination network. These networks show the communication among the contributing developers identified by the actual communication through email and by the concrete architecture, respectively. Then the congruence between these two networks and corresponding percentile value was calculated for each stable release. The percentile value indicates the extent to which both networks overlap. Relationships identified by each of these networks are presented in Fig.11.



Stable releases	No of Relationships among contributing developers			
	Derived Coordination Network	Concrete Coordination Network	Congruence	(%) Reverse Conway's law
stable-2.0.5	11017	925	574	62,06
stable-2.1	14884	805	443	55,04
stable-2.2	39059	6614	2165	32,74
stable-3	53152	5635	3980	70,63
stable-4	138228	4195	2983	71,11
stable-5	198602	11590	8253	71,21
stable-6	286244	17113	13634	79,68
stable-7	413188	10828	9453	87,31
stable-8	504458	7167	5768	80,48
stable-9	550427	4073	2857	70,15

Fig. 11. Resemblance of software architecture to communication pattern

It can be noted that the architecture of FreeBSD remains quite stable in terms of number of packages (Fig.3) and their interdependency as identified by the concrete architecture (column 3 of Fig.10). Thus it can be ascertained that

The architectural design of the software remains stable during its evolution.

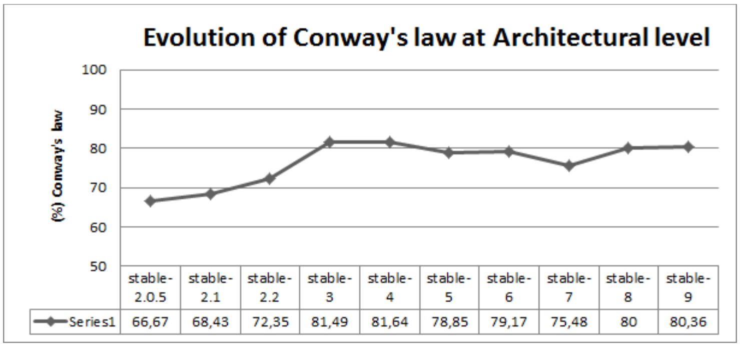
As can be seen from Fig. 11 the congruence values remain in-between 71% to 88% from stable release 3 to 9 (ignoring the first three releases as discussed in section 6.1). This result is closely tied with the results reported in Fig. 10. This implies that socio-technical congruence in reverse also holds for FreeBSD. And the communication pattern of contributing developer community simulates the underlying architectural dependency of the software to a gereat extent. Yet the derived coordination network is not able to estimate the coordination network, the former network overestimates the later one, similar to the derived architecture. Instead the following observatioin can be offered.

The communication need defined by the architectural design and interdependency among the modules of the software is effectively embedded within the communication pattern of the developer community of the FreeBSD project.

Again the over estimation by the derived coordination network can be vindicated as follows, this network was derived from package level concrete architecture. Thus developers contributing to a package or to the related packages were considered to have communication among them. This lead to a number of relationships among developers in the derived coordination architecture who might not have contributed to the same code files, but to the same packages. Yet these relationships among contributing developers should be taken as a suggestion to improve the coordination further in the community.

6.3 Evolution of Socio-technical Congruence in FreeBSD Project

To conceptualize the evolution of socio-technical congruence in FreeBSD we plotted the congruence values in graph against the stable releases. As shown in



	stable-2.0.5	stable-2.1	stable-2.2	stable-3	stable-4	stable-5	stable-6	stable-7	stable-8	stable-9
Series1	66,67	68,43	72,35	81,49	81,64	78,85	79,17	75,48	80	80,36

Fig. 12. Evolution of Conway's law at architectural level

Fig. 12 the congruence values remains stable around 76% to 82% starting from stable release 3.

Similar trend can be notified in Fig. 13 where the congruence value remains stable within the range 71% to 88% starting from stable release 3. Based on these observations it can be stated that with the maturation of the project, the communication pattern among developers get more structured following the need of communication based on their tasks. This socio-technical congruence pattern of communication among the community memebrs becomes a traditional practice as the project evolves. Thus it can be affirmed that

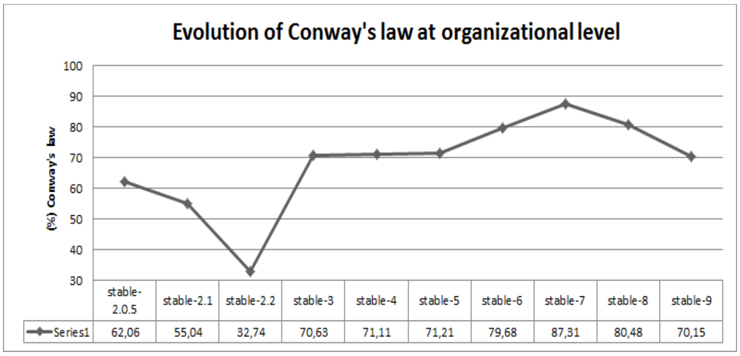
The socio-technical congruence in FreeBSD project remains stable during its evolution.

7 Threats to Validity

The following aspects have been identified which could lead to threats to validity of this study.

External validity (how results can be generalized): As case study subject, FreeBSD project was selected, which has been used popularly in the OSS evolution studies. Also, FreeBSD is a large and well established OSS project with over thirty years of evolution history. Yet the findings may not generalize to other OSS projects. This threat can only be countered by doing additional case studies with projects from different domains, which is a part of our future work.

Internal validity (confounding factors can influence the findings): Missing historical data - the study has been able to make use only of available data. It is possible, for instance, that there are commit records and developer chat entries other than that recorded in the emails. Also other email archives may contain



	stable-2.0.5	stable-2.1	stable-2.2	stable-3	stable-4	stable-5	stable-6	stable-7	stable-8	stable-9
Series1	62,06	55,04	32,74	70,63	71,11	71,21	79,68	87,31	80,48	70,15

Fig. 13. Evolution of Conway's law at organizational level

relevant data. Thus, we make no claim on the completeness of the email entries with relevance to this study target.

Construct validity (relationship between theory and observation): In this study, part of the email entries were categorized to specific stable release according to their date of post. The reasoning here is that developers commit and discuss on release planning before the product is officially released. Yet, we do not claim the perfection of this approach.

8 Conclusions

In this paper we evaluated the concept of Conway's law as a means of verifying the socio-technical congruence within the context of FreeBSD project. According to our judgment the congruence measure is significantly high in FreeBSD project which has a stable evolution history for the last seven releases of the project. In other words, the communication pattern of the contributing members within FreeBSD community are due to the communication need established by the concrete architecture. Having such high congruence between the code and the community would resolve many contemporary questions that cannot be answered otherwise. For instance, in identifying exact community member(s) to contract for an specific issue [36], it is required to trace which community members collaborates and exchange knowledge based on their task level responsibility.

This congruence is a desired property for collaborative development activities [14] and the reported result conforms the same for collaborative open source community. This result also creates a favorable basis to explore its implications on the OSS projects concerning the quality, success, and sustainability, which were already examined and confirmed in closed source projects.

Also, the results reported in this work hold for a specific OSS project, hence the study suffers from lack of generalizability. Thus a part of our future work is to extend and examine the findings in other OSS project, preferably to the BSD group and operating system domain.

References

1. Conway, M.E.: How Do Committees Invent? Datamation 14(4), 28–31 (1968)
2. Kwan, I., Schröter, A., Damian, D.: Does Socio-Technical Congruence Have an Effect on Software Build Success? A Study of Coordination in a Software Project. TSE 37(3), 307–324 (2011)
3. Ovaska, P., Rossi, M., Marttiin, P.: Architecture as a coordination tool in multi-site software development. Soft. Process: Improvement & Prac., 233–247 (2003)
4. de Souza, C.R.B., Quirk, S., Trainer, E., Redmiles, D.F.: Supporting collaborative software development through the visualization of socio-technical dependencies. In: International ACM Conference on Supporting Group Work, pp. 147–156 (2007)
5. Scacchi, W.: Understanding the requirements for developing open source software systems. IEE Proceedings Software 149(1), 24–39 (2002)
6. Bendifallah, S., Scacchi, W.: Work Structures and Shifts: An Empirical Analysis of Software Specification Teamwork. In: 11th ICSE, pp. 260–270 (1989)
7. Jongdae, H., Chisu, W., Byungjeong, L.: Extracting Development Organization from Open Source Software. In: 16th APSEC, pp. 441–448. IEEE (2009)
8. Raymond, E.S.: The new hacker's dictionary, 3rd edn. MIT Press, Cambridge (1996)
9. Nagappan, N., Murphy, B., Basili, V.R.: Architectures, Coordination, and Distance: Conway's Law and Beyond. Journal IEEE Software 16(5), 63–70 (1999)
10. Kwan, I., Cataldo, M., Damian, D.: Conway's Law Revisited: The Evidence for a Task-Based Perspective. IEEE Software 29(1), 90–93 (2012)
11. Brooks, F.P.: The Mythical Man-Month, Anniversary Edition. Addison-Wesley Publishing Company (1995)
12. Nagappan, N., Murphy, B., Basili, V.R.: The influence of organizational structure on software quality: an empirical case study. In: ICSE 2008, pp. 521–530 (2008)
13. Cataldo, M., Wagstrom, P.A., Herbsleb, J.D., Carley, K.M.: Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In: CSCW, Banff, Canada (2006)
14. Browning, T.: Applying the design structure matrix to system decomposition and integration problems: a review and new directions. IEEE Transactions on Engineering Management 48(3), 292–306 (2001)
15. Sosa, M.E., Eppinger, S.D., Rowles, C.M.: The misalignment of product architecture and organizational structure in complex product development. Management Science 50(12), 1674–1689 (2004)
16. Jingwei, W., Holt, R.C., Hassan, A.E.: Empirical Evidence for SOC Dynamics in Software Evolution. In: ICSM, pp. 244–254 (2007)
17. Herraiz, I.: A statistical examination of the evolution and properties of libre software. In: ICSM, pp. 439–442 (2009)
18. Herraiz, I., Gonzalez-Barahona, J.M., Robles, G., German, D.M.: On the prediction of the evolution of libre software projects. In: ICSM, pp. 405–414 (2007)
19. Fogel, K., Bar, M.: Open Source Development with CVS: Learn How to Work With Open Source Software. The Coriolis Group (1999)
20. Herbsleb, J.D., Grinter, R.E.: Splitting the Organization and Integrating the Code: Conway's Law Revisited. In: ICSE, pp. 85–95. ACM Press, New York (1999)
21. Baldwin, C.Y., Clark, K.B.: Design Rules: The Power of Modularity. MIT (2000)
22. Colfer, L., Baldwin, C.Y.: The Mirroring Hypothesis: Theory, Evidence and Exceptions, Working paper. Harvard Business School (2010)

23. Bowman, I.T., Holt, R.C.: Software Architecture Recovery Using Conway's Law. In: CASCON 1998, pp. 123–133 (1998)
24. FreeBSD (2013), `http://www.freebsd.org/`
25. Mathieu, G., Mens, T.: A framework for analyzing and visualizing open source software ecosystems. In: IWPSE-EVOL, pp. 42–47 (2010)
26. Daniel, M.G.: Using software trails to reconstruct the evolution of software. Journal of Software Maintenance and Evolution 16(6), 367–384 (2004)
27. Wang, Y., Guo, D., Shi, H.: Measuring the Evolution of Open Source Software Systems with their Communities. ACM SIGSOFT Notes 32(6) (2007)
28. Zhang, W., Yang, Y., Wang, Q.: Network Analysis of OSS Evolution: An Empirical Study on ArgoUML Project. In: IWPSE-EVOL, pp. 71–80 (2011)
29. Apache POI-Java API for Microsoft Documents (2013), `http://poi.apache.org/`
30. jsoup: Java HTML Parser (2013), `http://jsoup.org/`
31. Yamauchi, Y., Yokozawa, M., Shinohara, T., Ishida, T.: Collaboration with Lean Media: How Open-source Software Succeeds. In: CSCW, pp. 329–338 (2000)
32. Dayani-Fard, H., Yu, Y., Mylopoulos, J., Andritsos, P.: Improving the build architecture of legacy C/C++ software systems. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 96–110. Springer, Heidelberg (2005)
33. Mahbubul Syeed, M.M.: Binoculars: Comprehending Open Source Projects through graphs. In: Hammouda, I., Lundell, B., Mikkonen, T., Scacchi, W. (eds.) OSS 2012. IFIP AICT, vol. 378, pp. 350–355. Springer, Heidelberg (2012)
34. Bolici, F., Howison, J., Crowston, K.: Coordination without discussion? Socio-technical congruence and Stigmergy in Free and Open Source Software projects. In: 2nd STC, ICSE (2009)
35. Kazman, R., Carrière, S.J.: Playing Detective: Reconstructing Software Architecture from Available Evidence, Technical Re- p ort CMU/SEI-97-TR-010, Carnegie Mellon University (1997)
36. Syeed, M.M., Altonen, T., Hammouda, I., Systä, T.: Tool Assisted Analysis of Open Source Projects: A Multi-facet Challenge. IJOSSP 3(2) (2011)

[IV] M.M. Syeed, and I. Hammouda. Socio-Technical Dependencies in Forked OSS Projects: Evidence from BSD Family. *Journal of Software, vol. 9, no. 11*, pages 2895–2909. May, 2014.

Socio-Technical Dependencies in Forked OSS Projects: Evidence from the BSD Family

M.M. Mahbubul Syeed[a], Imed Hammouda[b]
[a] Department of of Pervasive Computing, Tampere University of Technology, Finland.
Email: mm.syeed@tut.fi
[b] Chalmers and University of Gothenburg, Sweden.
Email: imed.hammouda@cse.gu.se

Abstract— **Existing studies show that open source projects may enjoy high level of socio-technical congruence despite their open and distributed character. Such observation is yet to be confirmed in the case of forking, where projects originating from the same root evolve in parallel and are typically lead by different development teams. In this paper, we empirically investigate the endogenous and exogenous characteristics of BSD family projects related to socio-technical congruence. Our motivation is that BSD family, as a representative example of forked projects, share a common development ground for both the code-base and the development community, which may influence their evolution from a socio-technical perspective. Our study results show that the BSD family maintain a certain level of collaboration throughout the project history, mainly due to a shared portion of the community. This partly explains the relative harmony of socio-technical congruence levels in the BSD projects.**

Index Terms— **Open Source Software, Evolution, Conway's Law, Socio-Technical Congruence, Forking**

I. Introduction

SOFTWARE development requires effective communication, coordination, and collaboration among developers working on interdependent modules of the same project. The need for coordination is even more evident in Open Source Software (OSS) projects where development is often more dispersed and distributed [1]. As argued in the literature, such coordination and communication may be influenced and guided by the cooperation needs devised by the design of the software [2]. This suggests that there might exist a two way mapping between the communication patterns of the developer community and the architectural dependencies among the components of the software, in which one can be used to approximate the other.

This collaboration can effectively be examined and verified through the notion of socio-technical congruence which defines the match between the coordination needs established by the technical domain (i.e., the architectural dependencies in the software) and the actual coordination activities carried out by project members (i.e., within the members of the development team) [3].

This work was supported in part by TiSE Graduate School, Tampere, Finland and Nokia Foundation Grant, Finland.

In fact, socio-technical congruence provides an empirical verification of a well-known but insufficiently understood phenomenon known as Conway's Law [4] and describes to which extent the law is enforced in a given software development project [3] [5]. Such empirical verification has been a primary motivation for many research efforts in the realm of socio-technical congruence [6] [7]. However, research on the topic has always assumed that development of a software project is performed by the same organization or group of developers. In the case of open source, projects may evolve in parallel, lead by different development teams. This is known as "forking" [8]. To the knowledge of the authors, no research has been performed yet on socio-technical congruence in the context of forked projects.

In an earlier work we have studied socio-technical congruence, and the significance of Conway's Law, in the FreeBSD open source project [9]. Our previous study showed that the congruence measure is significantly high in FreeBSD and that the congruence value remains stable as the project matured. In this work, we extend our earlier study to cover the BSD project family, empirically investigating the endogenous and exogenous characteristics of BSD projects. BSD projects are popularly known in the research community. For instance, in [10], change history information is extracted from BSD projects for the visualization of change dependencies. Similarly, FreeBSD project has been studied to verify the viability of incremental development approach, and to identify the common characteristics of successful OSS development process in relation to their quality, in [11] and [12], respectively.

Within the endogenous characteristics we investigated the notion of socio-technical dependency through the measure of socio-technical congruence in the individual projects. In the technical domain, the architectural dependencies have been constructed out of source code syntactic information such as functional dependency, attribute referencing and header file inclusion dependency. On the social side, the coordination network has been built out of email conversations between developers.

Among the exogenous characteristics, we examined to what extent the forked projects collaborate and communicate with each other. As a measuring criteria of such collaboration, we quantitatively measured the alignment among the source code and the developer community of the forked projects. The rationale here is that forked projects hold the same root for both the code-base and the community, thus sharing a common development ground. Hereof it is worth to empirically investigate the extent to which such common ground is maintained during projects evolution.

The remaining of the paper is organized as follows. Section II introduces a number of key concepts that this study uses. Section III introduces the research questions explored and Section IV presents our study design. Results are reported and discussed in Section V, followed by final discussion and related work in Section VI. The overall impact of missing data on the reported results and the replication guidelines are presented in Section VII. Possible limitations and threats to validity are highlighted in Section VIII. Finally, Section IX concludes the paper and sheds light on future research.

II. Definitions

In this section we define a set of concepts used in this study.

A. *Conway's Law*

Conway's Law in its purest form states that "organizations which design systems are constrained to produce systems which are copies of the communication structures of these organizations" [4]. In other words, the software product architecture reflects the organizational structure of its development team [4] [3]. In [13], Conway's Law is considered homomorphic and thus claimed to be true in reverse as well. This means the communication pattern within a developer community should reflect the architectural dependency in the developed software. Thus, Conway's Law can effectively be interpreted as the basis for studying the social and technical interdependency within a software project [14].

B. *Socio-technical congruence*

The contemporary phenomenon "Socio-technical congruence" is actually the conceptualization of Conway's Law. Socio-technical congruence can be defined as the match between the coordination needs established by the technical domain (i.e., the architectural dependency in the software) and the actual coordination activities carried out by project members (i.e., within the members of the developer community) [3]. This coordination need can be determined by analyzing the assignments of people to a technical entity such as a source code module, and the technical dependencies among the technical entities [3]. Accordingly, developers within the community should communicate if there exists a communication need. For example, developers working on the same module or on the interdependent modules should be coordinating.

C. *Developer Contribution*

In this work, developer contribution to a software project can be defined as code contribution or any form of commit made to the code base.

D. *Explicit Architecture*

The explicit architecture of a software presents the relationship among components of a software (e.g., modules, files or packages) based on the actual design and implementation. For this work, functional dependency, attribute referencing and header file inclusion dependency at code file level are used to derive the Explicit Architecture of a software product.

E. *Explicit Coordination Network*

The explicit coordination network is a social network in which two developers have a relationship if they have direct communication history as seen by the mailing archives representing the social and technical interactions among the developers.

F. *Implicit Architecture*

The implicit architecture defines an architecture of the software where any two components (e.g., packages or code files) are related if there are developers who have either (a) contributed to both components, or (b) have direct communication at organizational level (e.g., a one to one email conversation). For instance, consider that developer D1 has contributed to packages P1 and P2, and developer D2 has contributed to package P3. Also consider that both developers have direct communication at organizational level as shown in Fig. 1(a). Thus according to the definition, packages P1, P2 and P3 are linked to each other in the Implicit Architecture (Fig. 1(b)).

G. *Implicit Coordination Network*

The implicit coordination network is the developer relationship network in which two developers have a relationship if they have contributed either (a) to a common code file or (b) to the code files that have direct relationships in the Explicit Architecture. For instance consider that developers D1 and D2 have contributed to package P1 and developer D3 has contributed to package P2. Also consider that P1 and P2 have a functional dependency (i.e., a direct relationship) as shown in Fig. 2(a). Then, according to the definition, developers D1, D2 and D3 are linked to each other in the Implicit Coordination Network as shown in Fig. 2(b).

H. *Forking*

In the context of open source development, *forking* occurs when a part of a development community (or a third party not related to the original project) starts a completely independent line of development based on the source code of the original project [8] [15]. To be considered as a fork, a project should have:

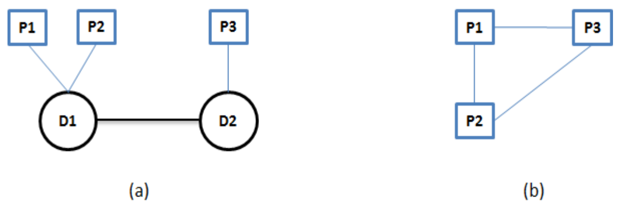


Figure 1. (a) Explicit Coordination Network with contribution to code base (b) Corresponding Implicit Architecture

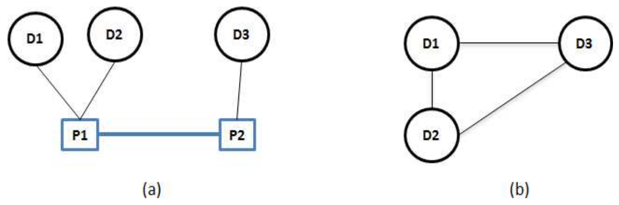


Figure 2. (a) Explicit Architecture with contributing developers (b) Corresponding Implicit Coordination Network

- A new project name.
- A branch of the software.
- A parallel infrastructure (web site, version control system, mailing lists, etc.).
- And a new developer community.

Based on this definition, we propose the following set of relationships within a pair of forked projects: (a) parent-child, in which one project is forked from the other, (b) siblings, if two projects are forked from the same parent project, and (c) lineages, for all descendant relationships in which (a) and (b) do not hold. For example, in Fig. 3, NetBSD and OpenBSD have a parent-child relationship, FreeBSD and NetBSD are sibling projects, whereas FreeBSD, and OpenBSD are the lineages of 386BSD.

III. Research Questions

Our choice of research questions is motivated by our agenda to measure the exogenous and endogenous characteristics of forked OSS projects related to socio-technical congruence.

(RQ1) How does the software architecture compare and evolve across forked OSS projects?

When a project is forked, the source code of the parent project is copied [8]. Thus it is natural that at the initial stage the source code, and hence the architecture, of both systems are similar. Based on this observation, we are interested in exploring the extent to which the forked projects share common architectural structure during their evolution. In doing so, we calculated and compared the architectures of the forked projects at three abstraction levels. Namely, at package level, at first directory level and at n^{th} directory level (i.e., the last directory where the files reside). We argue that this three level comparison can provide a holistic view of the architectural overlapping. For instance, it might be possible that forked projects maintain homogeneous architectural design at higher level of abstraction (e.g., in package level), yet getting liberated at detailed architectural level (e.g., n^{th} directory level).

(RQ2) How does the community compare and evolve across forked OSS projects?

Traditionally, the developer community divides when a project is forked [8]. This is typically followed by a community rebuild and restructuring process in both projects. Community members in both projects might communicate and coordinate in such circumstances in making both projects survive. Thus, our intention here is to examine how these fragmented communities act in building the projects: do they contribute to both projects? Does such collaboration sustain during the evolution of the projects?

(RQ3) How does the socio-technical congruence evolve within the forked OSS projects?

Socio-technical congruence is a natural consequence and a desired property for collaborative development activities, like OSS projects [16]. Conventional wisdom suggests that correspondence between the social and technical domain of a project may reduce the communication overhead and may increase productivity [7]. Furthermore, lack of collaboration is classified as a negative stimuli to performance [17] and has an influence on lowering productivity [5]. Consequently, socio-technical congruence can be a decisive property of a successful project [3] [5] [16]. With strong congruence measure projects can get more cohesive, organized, and self-dependent with higher productivity. Thus our intention here is to stress these reported observations in forked OSS projects by examining the extent to which Socio-Technical Congruence holds in forked projects.

IV. Study Design

This section presents in detail our study design, covering discussion on the case study selection, required data sets, data acquisition, cleaning, and analysis process.

A. Case and Subject Selection

To explore the three research questions, we performed a case study with three large (more than 1,414,641 LOC [18]), long-lived (around 20 years of evolution history) OSS projects that were forked from their predecessors. These case study projects are FreeBSD [19], NetBSD [20] and OpenBSD [21]. All three projects originate from the 386BSD project, which is the version of UNIX developed at the University of California, Berkeley. FreeBSD and NetBSD were directly forked from 386BSD during late 1993, and therefore have a sibling relationship. OpenBSD was forked from NetBSD in 1995, thus having a parent-child forking relationship. Whereas, FreeBSD and OpenBSD are lineages of 386BSD. As a consequence, the core of these projects encompass the code base of 386BSD. The forked relationship among these projects are shown in Fig. 3 and the lifetime of these projects till 2013 are shown in Fig. 4.

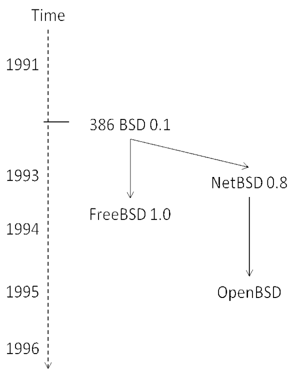


Figure 3. Rough time line of the forked BSD projects

Our selection of the BSD project family was influenced by the following factors: (a) the code base of these projects have undergone continuous development, improvement, and optimization for twenty years [19], (b) these projects have been developed and maintained by a large team of individuals [20], (c) the properties of a forked project hold for these projects, (d) these projects have extensively been used in earlier research on the evolution of OSS projects [22] [23] [18], and (e) results reported in this study can be stressed to OSS projects having similar properties, e.g., forking history, domain, community structure, and size.

B. Data Sets

OSS projects often consist of a number of software development repositories. These repositories contain a plethora of information on both the underlying software and the associated communication and development process [24] [25]. In the literature [26] a great

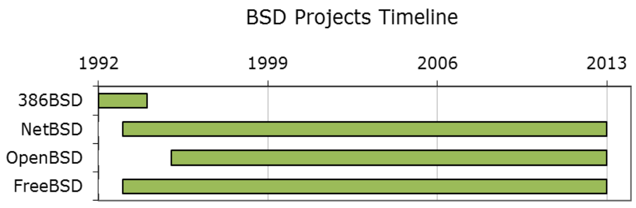


Figure 4. Life time of the BSD projects

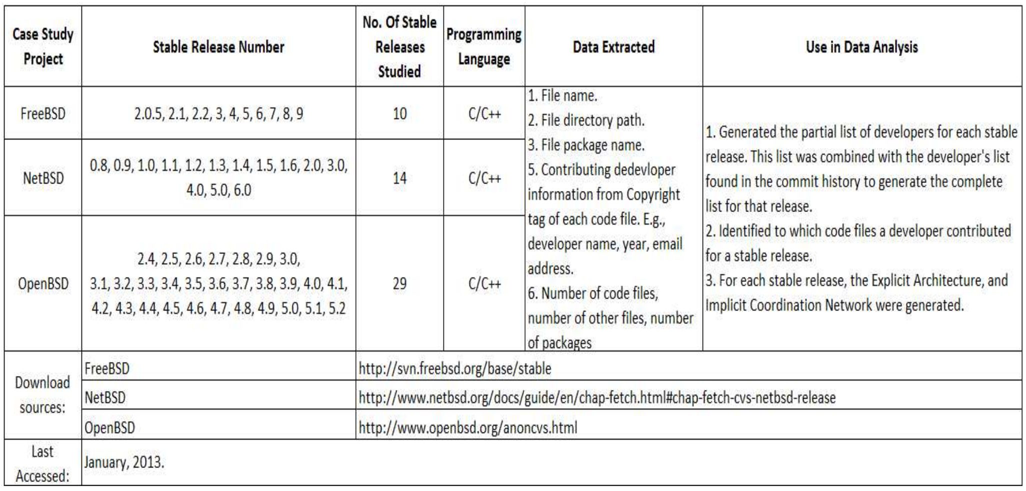
emphasis was given to leveraging these repositories for deriving technical dependencies as well as developers' coordination patterns. The repository data are often longitudinal, allowing for analysis along the whole project evolution phases. Such data sources are highly accepted and utilized medium for empirical studies on OSS projects [27] [28] [29]. In this study we utilized the following repositories.

Source code repository: We downloaded the source code of each stable release of the three projects. FreeBSD maintains its source code in Subversion version control system, whereas NetBSD and OpenBSD use CVS. In Fig. 5 we provide the details of the stable releases, the data collected from each release, and the corresponding download sources.

Mailing list archive: In OSS projects, email archives provide a useful trace of task-oriented communication and co-ordination activities of the developers during project evolution [30]. In the studied projects, email archives are categorized according to their purpose including commit records, stable release planning, chat, user emails, and bug reports. The archives contain the commit history and the email conversations since the initiation of the projects. In this study we used a complete list of commit records and email conversations from the beginning of each studied project. Consequently, data from relevant email archives was extracted and refined from each project, detail of which is presented in Fig. 6.

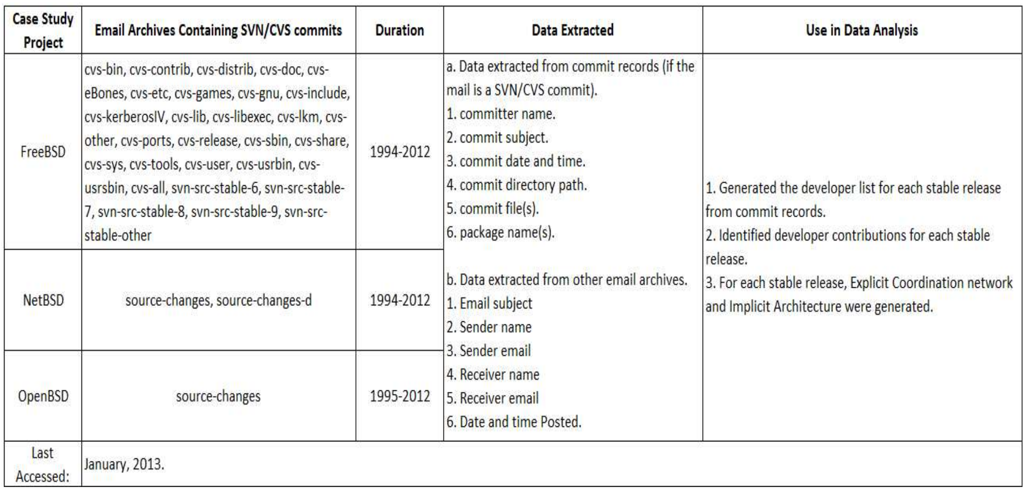
C. Data Collection

From source code repositories: The source code of each stable release of the selected projects was downloaded to a local directory. Fig. 5 lists the stable releases that were downloaded for each project. To extract data from each of the releases, a parser was written in Java. The parser searched through each directory of a stable release, read through the files in a directory and parsed relevant data. Each code file in a release contains a copyright directive. Under this directive the contributing developer name, email, and the copyright year is mentioned. The developers that were found in the process were considered as the initial contributors to that file. To get a complete list of contributors for a stable release, developers names were extracted from the commit history log and were merged with this contributor list. This process is described in following



Case Study Project	Stable Release Number	No. Of Stable Releases Studied	Programming Language	Data Extracted	Use in Data Analysis
FreeBSD	2.0.5, 2.1, 2.2, 3, 4, 5, 6, 7, 8, 9	10	C/C++	1. File name. 2. File directory path. 3. File package name. 5. Contributing dedevloper information from Copyright tag of each code file. E.g., developer name, year, email address. 6. Number of code files, number of other files, number of packages	1. Generated the partial list of developers for each stable release. This list was combined with the developer's list found in the commit history to generate the complete list for that release. 2. Identified to which code files a developer contributed for a stable release. 3. For each stable release, the Explicit Architecture, and Implicit Coordination Network were generated.
NetBSD	0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 3.0, 4.0, 5.0, 6.0	14	C/C++		
OpenBSD	2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 5.0, 5.1, 5.2	29	C/C++		
Download sources:	FreeBSD	http://svn.freebsd.org/base/stable			
	NetBSD	http://www.netbsd.org/docs/guide/en/chap-fetch.html#chap-fetch-cvs-netbsd-release			
	OpenBSD	http://www.openbsd.org/anoncvs.html			
Last Accessed:	January, 2013.				

Figure 5. Stable Releases of BSD Projects (FreeBSD, NetBSD and OpenBSD)



Case Study Project	Email Archives Containing SVN/CVS commits	Duration	Data Extracted	Use in Data Analysis
FreeBSD	cvs-bin, cvs-contrib, cvs-distrib, cvs-doc, cvs-eBones, cvs-etc, cvs-games, cvs-gnu, cvs-include, cvs-kerberosIV, cvs-lib, cvs-libexec, cvs-lkm, cvs-other, cvs-ports, cvs-release, cvs-sbin, cvs-share, cvs-sys, cvs-tools, cvs-user, cvs-usrbin, cvs-usrsbin, cvs-all, svn-src-stable-6, svn-src-stable-7, svn-src-stable-8, svn-src-stable-9, svn-src-stable-other	1994-2012	a. Data extracted from commit records (if the mail is a SVN/CVS commit). 1. committer name. 2. commit subject. 3. commit date and time. 4. commit directory path. 5. commit file(s). 6. package name(s). b. Data extracted from other email archives. 1. Email subject 2. Sender name 3. Sender email 4. Receiver name 5. Receiver email 6. Date and time Posted.	1. Generated the developer list for each stable release from commit records. 2. Identified developer contributions for each stable release. 3. For each stable release, Explicit Coordination network and Implicit Architecture were generated.
NetBSD	source-changes, source-changes-d	1994-2012		
OpenBSD	source-changes	1995-2012		
Last Accessed:	January, 2013.			

Figure 6. FreeBSD, NetBSD and OpenBSD email archives

sections. Information that was extracted using the parser is listed in Fig. 5, column 5. The parsed data for each stable release was then stored in a spreadsheet for further analysis.

From email archives: Data that is maintained in the email archives can be broadly classified into two groups, (a) email archives that maintain CVS/SVN commit records, and (b) archives that store general community discussions (e.g., on stable release planning, chat entries). Fig. 6 presents the total number of email archives that were extracted for each project along with specific names of archives containing the commit records, data collection period, collected data, and their analysis purpose.

For extracting data from each email entry, a data extraction program was written in Java. This data extractor used the web interface of the email archives. Thus each email was read as an HTML page and the data was extracted using the Jsoup HTML parser [31]. Data extracted from each email entry is listed in Fig. 6, column 4. This data was then stored in spreadsheets according to the archive name and year. After that, email data was sorted according to each stable release as follows: (a) emails and commit records were categorized into a specific release if the release number was mentioned in email subject (e.g., SVN commit emails provide release number in email subject for FreeBSD) and (b) other emails for which the release numbers were not mentioned (e.g., freeBSD-stable, freeBSD-chat and some of the CVS commit emails), the posting dates were checked. In this case, for instance, an email was categorized to stable release 3 if its posting date falls between the release date of stable release 2 and 3. The rationale here is that developers would commit to the code base and discuss on its release strategy before it is officially released.

For the CVS/SVN commit email, we parsed the commit path to the repository. The commit path was either mentioned in the subject or in the email body (in specified format). We extracted information like the directory path, package name, and if provided, the name of the modified code file(s) and the stable release number. The name of the committer for each of these CVS/SVN commit emails was considered as a contributor to the code base. Contributors found in this process were combined with the contributors found in the code base to get a complete list of contributing developers for each stable release.

Data preprocessing: Data that was extracted and parsed following the above process contained anomalies in many cases. For instance, developer names and email addresses might contain punctuation characters like semi-colons, inverted comas, brackets, unnecessary white space, and hyphens. Furthermore, parsers may have parsed data inappropriately in some cases. For example, the text *copyright rights reserved* can be treated as part of developer name while parsing copyright directive from a code file. To clean such anomalies data and punctuation characters, data cleaning programs were written in Java. To ensure the correctness of this process, we performed a manual checking on a randomly selected data to verify their correctness.

D. Data Analysis

This section is focused on topics related to the construction of the communication networks, architectures, and their use in measuring the socio-technical congruence utilizing the collected data.

Data analysis is restricted to the stable releases of the projects. This means, analysis point of this study is the stable release dates for a project. This choice of analysis point (instead of discrete time stamps) is made due to the following reasons: (a) a stable release reflects clear milestone for a project, which can also be counted as a step towards successful evolution, and (b) the source code for this study is available for stable releases only, which makes it obvious choice to take release dates as analysis points.

Developer Contribution: Developer contributions were measured release-wise in two ways: (a) from the copyright information provided in each source code file of a release and (b) from the commits made by a developer for a release. Fig. 7(a) shows a sample contribution made by developer *John Birrell* in FreeBSD stable release 3.

Explicit Architecture: The Explicit Architecture of a stable release was constructed based on functional dependency, attribute referencing, and header file inclusion dependency at code file level. For doing this, we used a tool named Understand [32]. This tool takes a source code repository as input and generates the corresponding Explicit Architecture. This tool has been used in previous research, e.g., in [33] [34]. The explicit architecture for each stable release of a project was derived at two abstraction levels, e.g., at code file level and at package level. An example of these two architectures for FreeBSD release 3 is shown in Fig. 8.

Explicit Coordination Network: Following the definition in Section II-E, the Explicit Coordination Network was derived for each stable release of a project. Email conversations for each stable release were used for this purpose. Fig. 7(b) shows example relationships in the Explicit Coordination Network of FreeBSD stable release 3. The weight column in this figure shows the number of email conversations that took place between two developers.

Implicit Architecture: The implicit architecture was generated following the definition in Section II-F. A partial snapshot of the package level Implicit Architecture for FreeBSD stable release 3 is shown in Fig. 9(a). In this architecture, a link weight between two packages designates the number of times the conditions (from Section II-F) hold. The significance of this network lays in the fact that developer communication patterns within the community may simulate the actual architectural dependency. That is, two developers should have communication if they are contributing to same or interrelated components of the software.

Implicit Coordination Network: This network was generated according to the definition presented in Section II-G. A snapshot of this network for FreeBSD stable release 3 is shown in Fig. 9(b). The network shows the actual communication need among developers, based on the design of the software (i.e., the Explicit Architecture). This network is essential due to the fact that if two subsystems exchange information, it is likely that communication among the developers of the two subsystems exists [4].

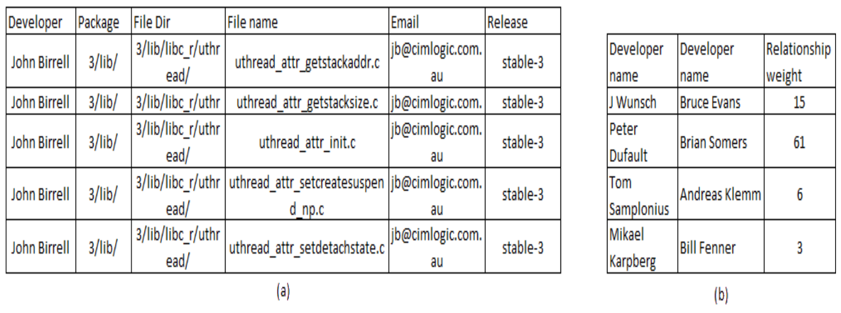
Measuring concurring and congruence among architectures and networks: Comparison among the architectures and communities was measured for two purposes: (a) to measure how the software architectures and communities compare and evolve across forked projects, and (b) to identify how the socio-technical congruence evolve within each forked project.

We applied the following similarity measure to serve both purposes. This approach is analogous to the fit measure used in organizational theory method [5]. An identical approach was applied in [9] for measuring the congruence in FreeBSD project.

$$Concurring/Congruence = \frac{Ref_{A/N} \bigcap Analogous_{A/N}}{|Ref_{A/N}|} \times 100) \quad (1)$$

In the above equation, $Ref_{A/N}$ is the reference architecture or network (either explicit or implicit), and $Analogous_{A/N}$ it the analogous architecture or network (either explicit or implicit) with which concurring or congruence will be measured.

This equation measures concurring between the two architectures or networks with respect to the reference one, $Ref_{A/N}$. Therefore, the numerator of equation (1) identifies the commonalities between the two given ar-



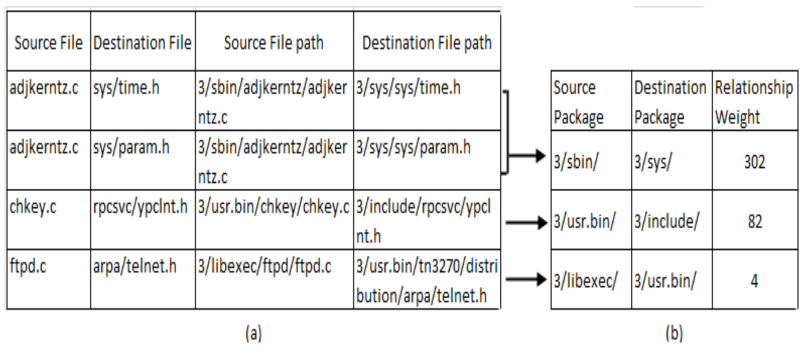
(a)

Developer	Package	File Dir	File name	Email	Release
John Birrell	3/lib/	3/lib/libc_r/uthread/	uthread_attr_getstackaddr.c	jb@cimlogic.com.au	stable-3
John Birrell	3/lib/	3/lib/libc_r/uthr	uthread_attr_getstacksize.c	jb@cimlogic.com.	stable-3
John Birrell	3/lib/	3/lib/libc_r/uthread/	uthread_attr_init.c	jb@cimlogic.com.au	stable-3
John Birrell	3/lib/	3/lib/libc_r/uthread/	uthread_attr_setcreatesuspend_np.c	jb@cimlogic.com.au	stable-3
John Birrell	3/lib/	3/lib/libc_r/uthread/	uthread_attr_setdetachstate.c	jb@cimlogic.com.au	stable-3

(b)

Developer name	Developer name	Relationship weight
J Wunsch	Bruce Evans	15
Peter Dufault	Brian Somers	61
Tom Samplonius	Andreas Klemm	6
Mikael Karpberg	Bill Fenner	3

Figure 7. (a) Sample contributions made by developer John Birrell (b) Sample relationships in Explicit Coordination Network



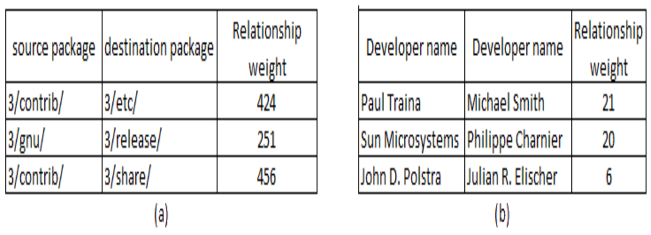
(a)

Source File	Destination File	Source File path	Destination File path
adjkerntz.c	sys/time.h	3/sbin/adjkerntz/adjkerntz.c	3/sys/sys/time.h
adjkerntz.c	sys/param.h	3/sbin/adjkerntz/adjkerntz.c	3/sys/sys/param.h
chkey.c	rpcsvc/ypclnt.h	3/usr.bin/chkey/chkey.c	3/include/rpcsvc/ypclnt.h
ftpd.c	arpa/telnet.h	3/libexec/ftpd/ftpd.c	3/usr.bin/tn3270/distribution/arpa/telnet.h

(b)

Source Package	Destination Package	Relationship Weight
3/sbin/	3/sys/	302
3/usr.bin/	3/include/	82
3/libexec/	3/usr.bin/	4

Figure 8. (a) Code file level Explicit Architecture (b) Package level Explicit Architecture



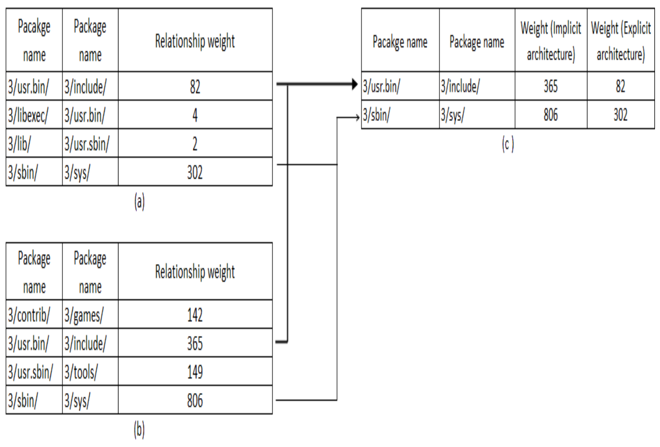
(a)

source package	destination package	Relationship weight
3/contrib/	3/etc/	424
3/gnu/	3/release/	251
3/contrib/	3/share/	456

(b)

Developer name	Developer name	Relationship weight
Paul Traina	Michael Smith	21
Sun Microsystems	Philippe Charnier	20
John D. Polstra	Julian R. Elischer	6

Figure 9. (a) Implicit Architecture (b) Implicit Coordination Network



(a)

Pacakge name	Package name	Relationship weight
3/usr.bin/	3/include/	82
3/libexec/	3/usr.bin/	4
3/lib/	3/usr.sbin/	2
3/sbin/	3/sys/	302

(b)

Package name	Package name	Relationship weight
3/contrib/	3/games/	142
3/usr.bin/	3/include/	365
3/usr.sbin/	3/tools/	149
3/sbin/	3/sys/	806

(c)

Pacakge name	Package name	Weight (Implicit architecture)	Weight (Explicit architecture)
3/usr.bin/	3/include/	365	82
3/sbin/	3/sys/	806	302

Figure 10. (a) Explicit Architecture (b) Implicit Architecture (c) Congruence

chitectures or networks, then is divided by the size of the reference architecture and expressed in a scale of 100.

The application of Equation (1) to specific cases is presented next.

Comparing the Architecture: To measure and compare the architectural concurring among the three projects, we performed a stable release wise comparison of the explicit architectures for each pair of forked projects. Thus in this case, both $Ref_{A/N}$ and $Analogous_{A/N}$ represent two comparable explicit architectures taken from two projects. To be comparable, the stable releases of two projects should be released around the same time period. For instance, consider the stable releases of FreeBSD and NetBSD projects. FreeBSD has 10 stable releases whereas NetBSD has 14 (Fig. 5, column 2). Thus to compare two releases, each taken from the two projects, we determined the release date-wise correspondence. Therefore, FreeBSD release 6 and NetBSD release 3.0 have a correspondence as they were released in November, 2005 and December, 2005, respectively.

The intersection operation in numerator of equation (1) is calculated at three abstraction levels of the explicit architectures, namely, package level (p), first directory level (d_1) and code file directory level (d_n). For package level, the intersection operation results in the number of packages that are common (by comparing the names of the packages) between two releases. On the other hand, for the directory level, e.g., d_1 and d_n, the intersection operation provides the total number of directories that have the complete match in their directory paths. As an illustrative example, consider FreeBSD release 6 and NetBSD release 3.0 which have 19 and 22 packages, respectively. Thus $|FreeBSD - release - 6|$ = 19 and $|NetBSD - release - 3.0|$ = 22. The intersection operation between these two explicit architectures resulted in 16 packages having the same names.

Finally, the concurring value was calculated taking each of these architectures as a reference architecture. This value depicts the extent to which each of these stable releases coincide with the other. In continuation to the above example, FreeBSD release 6 has 84.21% (16/19*100) and NetBSD release 3.0 has 72.72% (16/22*100) concurring with each other. These values were then plotted in a trend chart to visualize how such concurring evolves with the projects. An example of this process is presented in Fig.11 and discussed in Section V-A.

Comparing the Community: To compare the communities among the three forked projects using the similarity measure in Equation (1), we carried out the following: first, the release wise developer list was generated for each project. This step was discussed in section IV-C. Second, for a given pair of releases, the union operation in the numerator identifies the number of contributors in both releases whose names are lexically identical. Finally, for each of the stable releases, concurring value was calculated considering each as a reference network. These values were then plotted in a trend chart. An example of this process is presented in Fig. 14 and discussed in Section V-B.

Socio-technical Congruence: To measure socio-technical congruence using the similarity measure in (1) the following approach was applied: the intersection operation in numerator was carried out between (a) Explicit Architecture and Implicit Architecture, and between (b) Explicit Coordination Network and Implicit Coordination Network. This operation identifies the number of edges (or relationships) that are identical for both the architectures or the networks.

The former measure (in (a)) illustrates the match between the architectural dependency and the architecture produced due to the communication structure of the community. The latter measure (in (b)) in turn depicts the match between the actual coordination activities in the community and the coordination need established by the architectural dependency of the software. These measures verify Conway's Law and the reverse Conway's Law, respectively. Both the measures were determined for each stable release for all three projects. A partial snapshot of the congruence between Explicit and Implicit Architectures of FreeBSD stable release 3 is shown in Fig. 10.

Then to identify the extent to which the implicit architecture and implicit network approximate the corresponding explicit one, we calculated the similarity measure in (1), taking each of the explicit architecture and network as the reference one. The resulting values were plotted in a trend chart for each project to conceptualize their evolution pattern. An example of this analysis is presented in Fig. 17 and discussed in Section V-C.

E. Implementation and Verification

Tools Used In the Study: A number of existing tools and OSS packages were used in this work. For instance, we used the tool *Understand (version: 3.1.659)* [32] to generate the Explicit Architectures. To read/write excel files Apache POI [35] was used. Also, Jsoup HTML parser [31] was used to parse the HTML files.

Implementation and Verification of the Developed Programs: We implemented several data extraction, cleaning, and analysis programs in Java for this work. Data extraction programs were used to extract data from relevant sources and cleaning programs were used for removing the anomalies in the collected data. To verify the correctness of these programs, a two pass evaluation were conducted. First, the programs were tested with a limited number of data samples taken from each of the projects. Notified bugs (e.g., errors in the parsed data for an HTML tag) were fixed accordingly. Second, a manual checking on a random sample of the actual collected data was done. The accuracy of collected data in the second pass was reported to be over 97%.

Additionally, analysis programs were written for generating the architectures, communication networks, release-wise comparisons, and for measuring congruence. These programs in turn were tested following a similar method as stated above.

V. Result Analysis

The target of this study is three-fold. First, we verify the extent to which the forked projects collaborate in both technical and social domain. Second, we measure the socio-technical congruence in each project to conceptualize the socio-technical dependencies. Finally, we study the projects' pattern of evolution during their maturation.

A. Pattern of Architecture Evolution

In this section we present the results of the evolution of the architectural design for each forked project in relation to the other projects. In verifying this, pair-wise comparison of the architectural designs (each taken form the compared projects) were made at three abstraction levels. This action was performed according to the procedure presented in Section IV-D. The result of this comparison is presented in Figs. 11, 12 and 13, one for each pair of projects. These figures show the concurring of architectures (plotted in the Y-axis) for each comparable stable release pair (plotted in the X-axis) of the projects.

Overall architectural evolution revealed similar patterns for all three types of forking relationships, e.g., sibling projects, parent-child projects, and lineages. At higher abstraction level (e.g., package level) the architectures of the forked projects maintain high correspondence between them, which remains consistent as the projects evolve. However, at the detailed architectural level (e.g., at directory levels d_1 and d_n), the design and implementation became more disjoint and independent.

For instance, in Fig. 11, the package level concurring between the architectures of FreeBSD and NetBSD projects remain high throughout their release history. For FreeBSD it remains between 61,9% and 84,21%, whereas for NetBSD it is between 57,69% and 80% with slight drifts between the ranges. Contrary to this, directory level overlapping (d_1 and d_n) point out a different trend. In both of these cases, a consistent decrease in concurring can be noticed. For example, for NetBSD and FreeBSD the overlapping at d_1 directory level begins with 82,81% and 56,1% respectively, which gradually decreases to 37,39% and 41.72% respectively. Likewise, at d_n level, the overlapping goes down to 3,63% and 3,34% from 29,77% and 10,82% respectively.

For the other two cases (Fig. 12 and 13), a similar trend was noticed with minor distinction during the early stages of the projects. For instance, in Fig. 13 the overlapping of all three architectural level starts with a very low ratio, which however had a sharp rise in the next release. For the subsequent releases, the pattern remains similar to the observations stated earlier.

Additionally, at any given point of the comparison, the adherence to common architectural design falls off significantly from abstract to detail level of the design. For instance, in 2012, the FreeBSD package level overlapping is 75%, which is however around 41,72% and 3,34% for directory level overlapping d_1 and d_n, respectively. This observation holds for all the three projects.

These observations indicate that the BSD forked projects preserve a common structure at higher level of design, which are however, get liberated progressively at the detailed architectural design. However, thorough analyses of architectural design need to be conducted to fully affirm this claim.

B. Pattern of Community Evolution

Forking of a project causes a split in the community. The fragmentation of the community is typically followed by a rebuild and restructuring phases in both projects (the original and the fork). However, both projects share the same source of code-base, which could stimulate the development communities of the two projects to contribute to both. This observation lead us to investigate the extent to which the community members (from each project) contribute during the evolution of both projects.

The investigation was done according to the process defined in Section IV-D. The results are presented in Fig. 14, 15, and 16, one for each pair of projects. The findings reveal that the level of participation of the community members in the compared projects remains consistent within a given range. Also, a similar pattern of participation is noticed for the three types of forking, confirming the earlier observation in Section V-A.

Relating these observations to individual cases show that for the FreeBSD and NetBSD projects (Fig. 14), the community overlapping remains between 23,49% and 44,9%, whereas for NetBSD it is between 26,47% and 44,23%. Within this range of participation there exist several drifts. For instance, in 1999 and 2007 (Fig. 14), a decrease in participation can be observed.

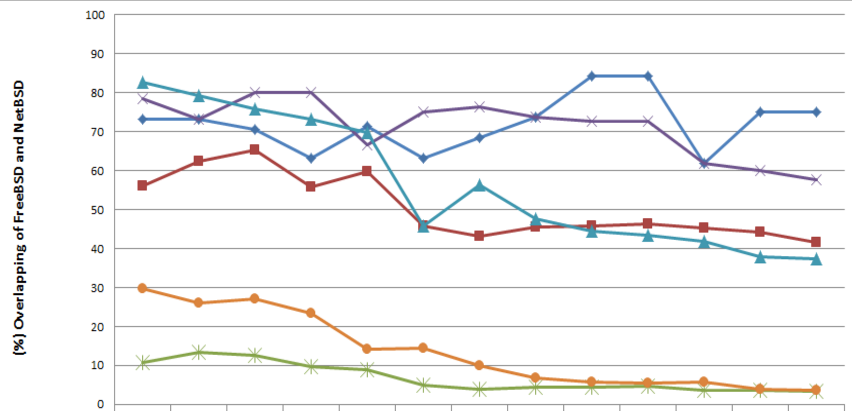
For the other two cases (Fig. 15, and 16), the pattern of overlapping follows a similar trend, except for the first two releases. This observation is similar to that discussed in Section V-A. For instance, the level of contribution rises sharply after having a low participation at the early release. Apart from this, the participation level (in Fig. 15) for NetBSD remains between 42,69% and 50,3%, and for OpenBSD between 34,42% and 38,31%. Similarly, for FreeBSD and OpenBSD (Fig. 16) it is 30,58%-35,05% and 27,18%-30,38%, respectively.

These results lead to the point that a certain group of community members maintain contributions to all the projects. The number of participation also remains stable throughout the evolution.

C. Evolution pattern of Socio-technical Congruence

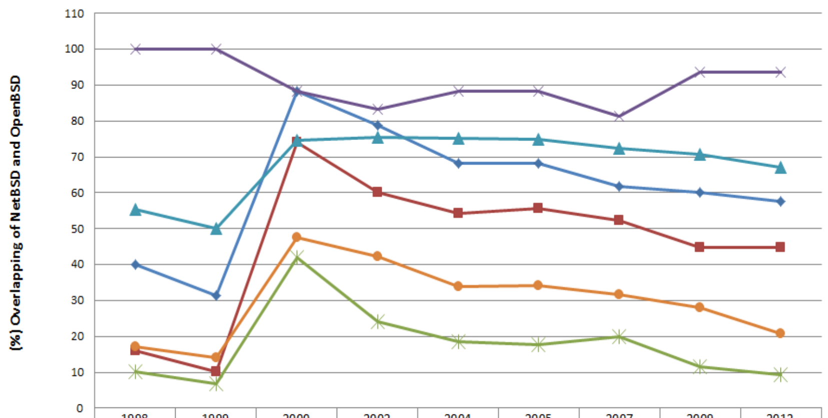
The measurement of Socio-technical Congruence for a project is a two step process. First, the extent to which the communication patterns of the members of the developer community resemble the actual architectural dependencies is verified. And then, the resemblance of the architecture to the community communication is investigated. In doing so, we derived both the implicit and explicit architectures and community collaboration networks, and measured the corresponding congruence. This process was discussed in detail in Section IV-D.

The evolution of congruence at architectural level for the three projects is shown in a trend chart in Fig. 17. In this figure, the congruence approximation is plotted in the



	1993	1994	1995	1996	1998	1999	2000	2002	2004	2005	2007	2008	2012
◆ Package Overlap (FreeBSD)	73,33	73,33	70,59	63,16	71,43	63,16	68,42	73,68	84,21	84,21	61,9	75	75
■ First Directory Overlap (FreeBSD)	56,1	62,37	65,31	55,91	59,66	45,79	43,17	45,47	45,84	46,4	45,31	44,33	41,72
✳ Last Directory Overlap (FreeBSD)	10,82	13,43	12,72	9,84	8,88	4,96	4,03	4,38	4,48	4,7	3,58	3,72	3,34
✕ Package Overlap (NetBSD)	78,57	73,33	80	80	66,67	75	76,47	73,68	72,73	72,73	61,9	60	57,69
▲ First Directory Overlap (NetBSD)	82,81	79,28	75,83	73,32	69,82	45,79	56,38	47,66	44,39	43,56	41,84	37,97	37,39
● Last Directory Overlap (NetBSD)	29,77	26,05	27,25	23,39	14,1	14,49	10,1	6,76	5,81	5,6	5,68	3,94	3,63

Figure 11. Architectural evolution between the sibling forked projects (FreeBSD and NetBSD)



	1998	1999	2000	2002	2004	2005	2007	2009	2012
◆ Package Overlap (NetBSD)	40	31,25	88,24	78,95	68,18	68,18	61,9	60	57,69
■ First Directory Overlap (NetBSD)	15,93	10,11	74,22	60,17	54,2	55,61	52,25	44,73	44,81
✳ Last Directory Overlap (NetBSD)	10,23	6,9	41,91	24,26	18,65	17,61	19,94	11,64	9,48
✕ Package Overlap (OpenBSD)	100	100	88,24	83,33	88,24	88,24	81,25	93,75	93,75
▲ First Directory Overlap (OpenBSD)	55,49	50	74,67	75,42	75,19	74,83	72,32	70,83	67,09
● Last Directory Overlap (OpenBSD)	17,28	14,15	47,67	42,23	34,02	34,16	31,79	28,12	20,91

Figure 12. Architectural evolution between parent-child forked projects (NetBSD and OpenBSD)

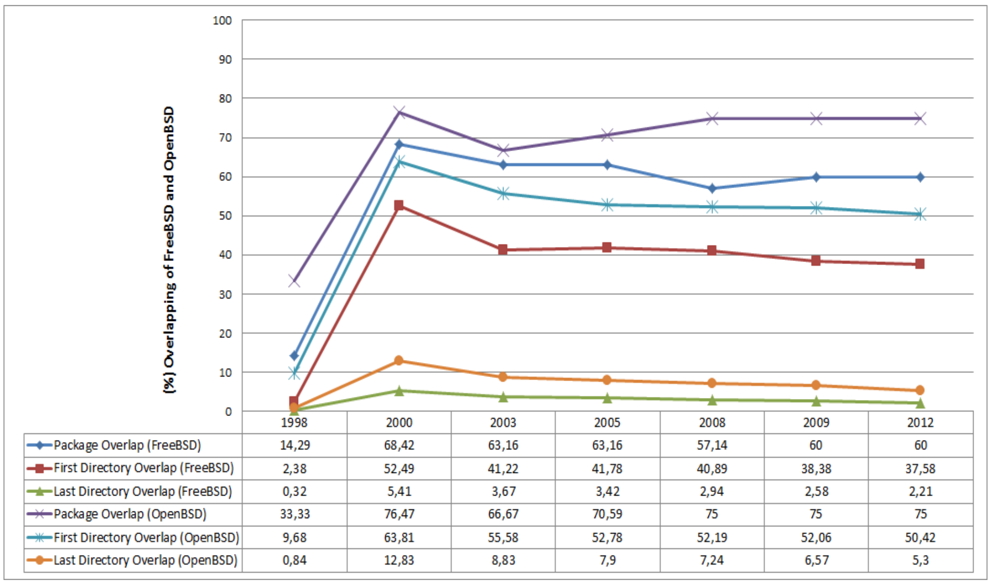
Y-axis (in percentile value) against each stable release of the projects (plotted in the X-axis).

For FreeBSD (the blue line in Fig. 17), the approximation of the congruence consistently has risen starting from 60,5% at the first stable release and has gone up to 89,4%. It had a sharp rise during the early five releases and got stabilized for the later six releases. During this period the congruence level remained between 84,83% and 89,4%. We considered the first four congruence values as outliers as a project usually goes under considerable restructuring and reformation after it is being forked.

For OpenBSD (the green line in Fig. 17) we observed a similar trend of congruence to that of FreeBSD. For the initial two releases the approximation of congruence were around 75%, that increased sharply to 88,38% on the third stable release. Till then onwards it remained stable within the range 85,56% and 88,78%.

In contrast to these two projects, NetBSD (the maroon line in Fig. 17) had a different pattern. In NetBSD the congruence approximation started with 85% and remained stable around 80,77% to 87,5% for the first twelve releases. Nevertheless, for the recent releases (e.g., the last two stable releases), the project experienced a decrease in congruence which has gone bellow 80%.

Accumulation of these results portrays that the approximation of the Explicit Architecture by the congruence is considerably high in all these three projects, which remains stable throughout the evolution. This implies that the architecture derived from the communication pattern of the developer community effectively represents the actual architecture of the software. That is, to a considerable extent the communication of the contributing developers



	1998	2000	2003	2005	2008	2009	2012
Package Overlap (FreeBSD)	14,29	68,42	63,16	63,16	57,14	60	60
First Directory Overlap (FreeBSD)	2,38	52,49	41,22	41,78	40,89	38,38	37,58
Last Directory Overlap (FreeBSD)	0,32	5,41	3,67	3,42	2,94	2,58	2,21
Package Overlap (OpenBSD)	33,33	76,47	66,67	70,59	75	75	75
First Directory Overlap (OpenBSD)	9,68	63,81	55,58	52,78	52,19	52,06	50,42
Last Directory Overlap (OpenBSD)	0,84	12,83	8,83	7,9	7,24	6,57	5,3

Figure 13. Architectural evolution pattern between lineage forked projects (FreeBSD and openBSD)

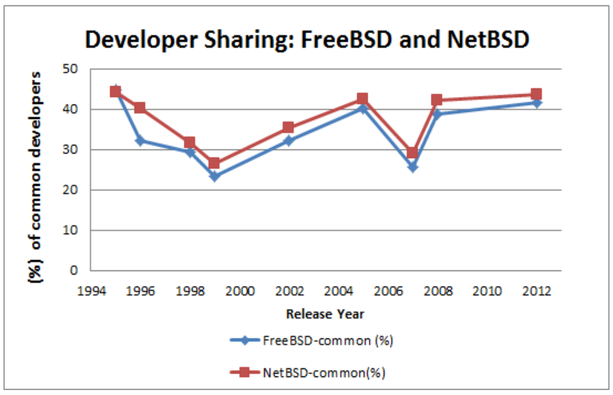


Figure 14. Community concurring pattern between FreeBSD and NetBSD projects

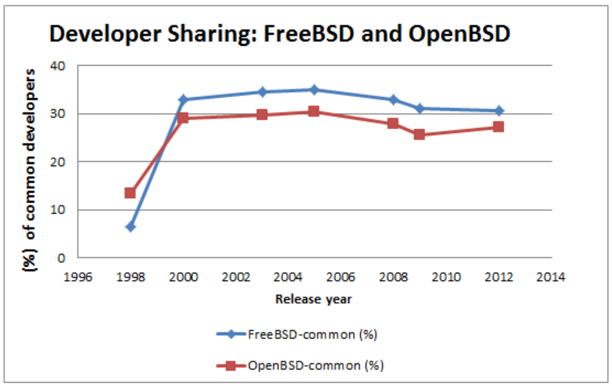


Figure 16. Community concurring pattern between FreeBSD and OpenBSD projects

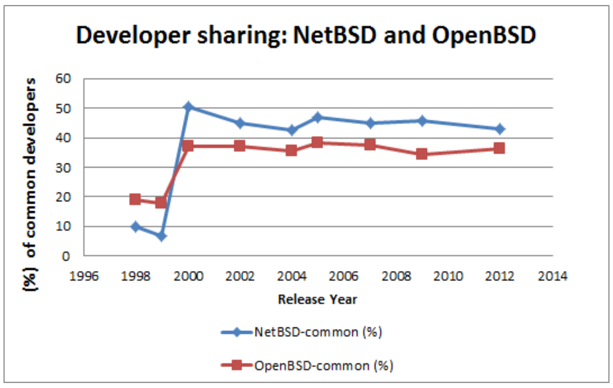


Figure 15. Community concurring pattern between NetBSD and OpenBSD projects

in the community may actually be due to the coordination needs as identified by the architectural dependencies.

On the other hand, the approximation level of the congruence to that of the Explicit Coordination Network reveals a similar pattern for the three projects. Fig. 18 shows the evolution of approximation against each stable release of the projects.

For FreeBSD (the blue line in Fig. 18), the approximation of the congruence remained between 70,63% and 87,31% from the fourth stable release onwards. A few drifts in congruence in the early three releases were noticed, which can be justified with the same reasoning as before. Yet, there was a decreasing trend of congruence noticed for the last two stable releases.

In the case of OpenBSD (the green line in Fig. 18), the approximation of the congruence to that of Explicit Coordination Network started with 80%, and remained stable between the value 73,35% and 87,77% during the entire evolution of the project. Only for the last release the congruence value went down to 39,58%, which is mainly due to missing data.

For NetBSD (the maroon line in Fig. 18) the congruence approximation started with a high value of 98,87% and remained stable between 8139% and 98,87% as the project progressed. Only for the tenth release (May 2005 in the chart) the congruence has gone as bellow as

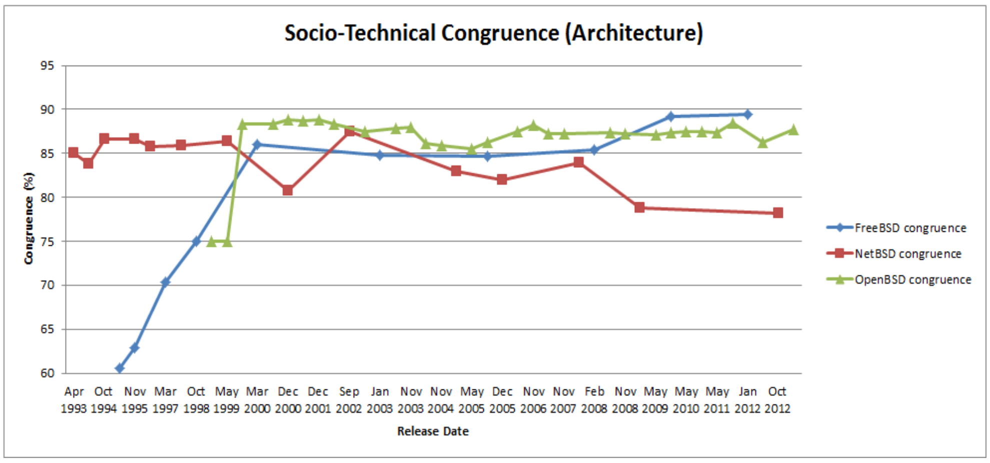


Figure 17. Evolution of Congruence at Architectural Level of the BSD Projects

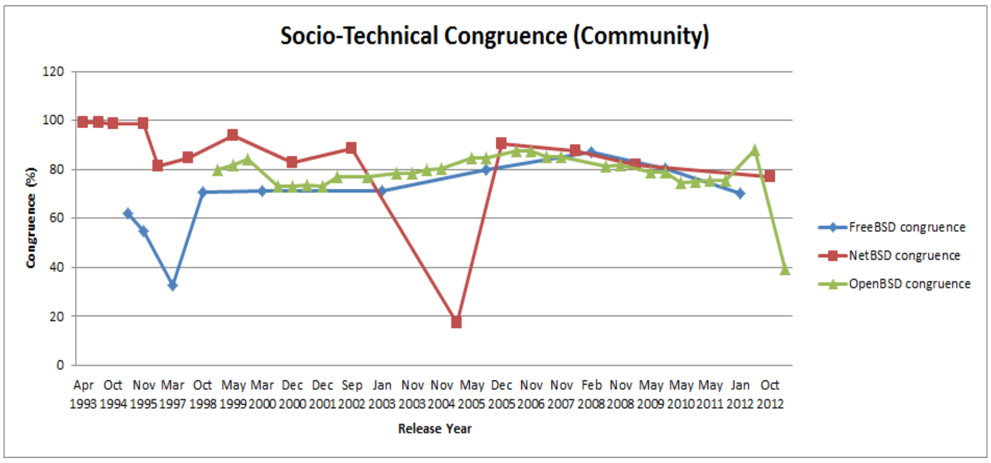


Figure 18. Evolution of Congruence at Community Level of the BSD Projects

17,23%. But it can be treated as an outlier due to missing data. Yet there was a slight decrease noticed for the last three stable releases.

To summarize these results, it can be conceived that the congruence approximation to that of Explicit Coordination Network is considerably high for the three projects. That is, the communication pattern of the developer community derived from the architectural dependency of the components effectively resembles the actual communication pattern. Thus, the communication pattern of contributing developer community can be used to simulate the underlying architectural dependency of the software to a great extent.

VI. Discussion

In this section we hereby summarize the findings of this study and possible implications in relation to prior works.

A. *Research Questions Revisited*

The evidence presented provides a strong indication that each forked project in the BSD family enjoys a high level of Socio-technical congruence throughout their evolution history. Thus, it can be affirmed that to a considerable extent the communication of the contributing developers in the BSD communities might be due to the coordination needs as identified by the technical dependency, and vice-versa. This observation is in-line with the prior work that reported congruence as a desired property and a natural phenomenon of collaborative development works [16] [36].

Alongside these observations, communities of the forked BSD projects have maintained a certain level of collaboration throughout the project history. Our reported model of collaboration shows that a portion of the community is mutual for both the projects. In literature, this group of community members are termed as the bridge between the projects [37], and a means of information flow and collaboration [37] [38].

Moreover, the architectural design at higher abstraction level has remained homogeneous among the forked projects. This might have supported the developer community with better understanding of the overall system designs and have created a common ground for collaboration and contribution. However contrary to this, at detail architectural level these projects are progressively getting liberated. This could be explained by the fact that the developer community of each fork has adopted their own implementation strategies when it comes to fine grained design decisions.

Finally, it was noticed that the pattern of community and architectural evolution for all the three forking relationships (e.g., siblings, parent-child and lineages) have followed similar patterns. This observation highlights the point that forked projects that have originated from the same root project would ideally share a common architectural design and a healthy inter-project collaboration.

B. Implications

It can be argued here that Socio-technical congruence plays a pivotal role in forming cohesive and organized community driven projects, which eventually leads to their successful evolution with high quality. This argument is also affirmed in earlier literature conducted on in-house projects: Higher congruence influences project success [3] [5] [16], with improved productivity [39] [6], maintainability [40], and quality [7].

This measure of socio-technical congruence would better serve the purpose of software development process and organization. Because it provides a quick index of how well the organization is actually aligned with the current and planned sub-division of responsibility in the project [41]. Additionally, the Implicit Architecture can be used as a complementary to the traditional reverse engineering process [42] [43] to derive and validate the recovery of the Explicit Architecture of legacy systems.

The identified pattern of collaboration among the three projects could be one way to explain the sustainability of the forked projects [44], particularly during their early formation stages. Additionally, further study could be initiated to verify the impact of such collaboration on cross project porting and code cloning [45] [46].

Overall, based on our study results, we claim that the traditional perception of forking in OSS projects, which is thought to have negative stimuli for sustainable evolution of the projects [8], can be effectively remedied though (a) maintaining a consistent and cohesive abstract architectural design to form a common ground of collaboration among the forked projects, (b) adopt a collaboration model in which members of a project could participate in other forks, and (c) maintain a consistent and high socio-technical congruence within the project.

None-the-less, this study puts a step forward in reasoning about the successful evolution of forked OSS projects, as this perspective has rarely been studied in current literature on OSS evolution analysis [8] [47].

VII. On the Missing data and Replication of the Study

Data collection process for this study sufferers from some missing data. The missing data constitutes the general communication emails stored in the email archives. Missing email conversations are encountered for NetBSD and OpenBSD projects. To be specific, email conversations during the period of April, 2005 to May, 2005 can not be extracted fully for NetBSD project. Whereas for OpenBSD project, missing emails are noticed during the period of September and October, 2012. In case of NetBSD it is mainly due to broken links to the archives, and for OpenBSD it is probably due to unavailability of the data during that time period.

However, the volume of such missing data is not massive, and thus, have little impact on the overall results. Only at the two points of congruence measure (as discussed in Section V-C), such missing data injected drifts, which however, do not hamper the overall trend of the congruence.

Replication of the study depends on addressing several issues, which includes, (a) data collection from the relevant sources, (b) cleaning and representation of the data and finally, (c) carrying out the analysis. In what follows, a guideline to accomplish these tasks.

Data is collected from two sources, SVN/CVS repositories and email archives. A detail discussion on downloading and extracting data from these sources are presented in Sections IV-B and IV-C. However, to ease this process of data collection for interested researchers, we make available the extracted data in the link given bellow[1]. Further instructions on how to interpret and use the data in replicating this study is discussed in the given link.

Finally, generating the architectures and networks, and carrying out the congruence measure are done thorough the implementation of scripts. There scripts are directly derived from the definitions and analysis methods discussed in Sections II and IV-D, respectively. Tools and packages listed in Section IV-E are used for script implementation. All the packages are open source and are available online for free downloading. However, the scripts used in this study are not made available in the given link. If researchers require assistance in implementing the scripts, we could provide adequate guidelines and the scripts upon request[2].

VIII. Threats to Validity

The following aspects have been identified which could lead to threats to validity of this study.

External validity (how results can be generalized): As case study subject, projects from the BSD family were selected, which are FreeBSD, NetBSD and OpenBSD. All these projects belong to the operating system domain, have large developer and user communities, and have over twenty years of evolution history. Additionally, OSS evolution studies often used these projects as case study. Thus it might be possible to stress the results reported in this article to the population of OSS projects having similar properties, e.g., domain, project size, evolution history. Yet, we cannot claim complete external validity of the results.

[1]http://msyeed.weebly.com/replication-package.html
[2]Contact: *rajit.cit@gmail.com*

Internal validity (confounding factors can influence the findings): Missing historical data - the study has been able to make use only of available data. It is possible, for instance, that there are commit records and developer chat entries other than that recorded in the emails. Additionally, we encountered several broken URL links for emails that could not be retrieved. Thus, we make no claim on the completeness of the email entries with relevance to this study target.

Construct validity (relationship between theory and observation): There exist a few issues that concern the construct validity of the study. First, part of the email entries were categorized to a specific stable release according to their date of post. The reasoning here is that developers commit and discuss on release planning before the product is officially released. Yet, we do not claim the perfection of this approach. Second, the data extraction programs written for this study provided an accuracy of 97%, which was measured with random sample of the collected data. This may affect the construct validity.

IX. CONCLUSIONS

The current study provides empirical evidence that successful OSS forked projects that are lineages of an ancestor project may follow similar evolution patterns in terms of (a) technical and social dependencies and (b) achieving a high level of congruence that sustains throughout their evolution. Though from a technical perspective the forked projects get more and more independent by time, they may enjoy a sustainable level of cross project collaboration. Keeping in line with prior evidence [9], we can argue that congruence is an implicit characteristic of successful forked OSS projects, and combining it with inter project collaboration would portray the reason behind the success of such projects. This claim however needs further empirical evidence. As an alternative to the qualitative argumentation approach taken in our study, one could frame our research questions as hypotheses and perform statistical analysis to evaluate them. This constitutes our future work.

REFERENCES

[1] A. Mockus, R. Fielding, and J. Herbsleb, “Two case studies of open source software development: Apache and mozilla,” *Journal of TOSEM*, vol. 11, no. 3, pp. 309–346, 2002.

[2] G. Valetto, S. Chulani, and C. Williams, “Balancing the value and risk of socio-technical congruence,” *Workshop on Sociotechnical Congruence*, 2008.

[3] I. Kwan, A. Schrter, and D. Damian, “Does socio-technical congruence have an effect on software build success? a study of coordination in a software project,” in *IEEE Trans. Software Eng.*, vol. 37, no. 3, 2011, pp. 307–324.

[4] M. E. Conway, “How do committees invent?” *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[5] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley, “Identification of coordination requirements: Implications for the design of collaboration and awareness tools,” in *ACM CSCW*, 2006, pp. 353–362.

[6] L. Colfer and C. Baldwin, “The mirroring hypothesis: Theory, evidence and exceptions,” in *working paper, Harvard Business School*, 2010.

[7] N. Nagappan, B. Murphy, and V. Basili, “The influence of organizational structure on software quality: an empirical case study,” in *ICSE ’08 Proceedings of the 30th international conference on Software engineering*, 2008, pp. 521–530.

[8] G. Robles and J. Gonzalez-Barahona, “A comprehensive study of software forks: Dates, reasons and outcomes,” in *OSS, IFIP AICT 378*, 2012, pp. 1–14.

[9] M. Syeed and I. Hammouda, “Socio-technical congruence in oss projects: Exploring conways law in freebsd oss evolution,” in *Proceedings of 9th International Conference of Open Source Systems (OSS), Springer*, 2013.

[10] M. Fischer, J. Oberleitner, J. Ratzinger, and H. Gall, “Mining evolution data of a product family,” *ACM SIGSOFT Software Engineering Notes*, vol. 4, no. 30, pp. 1–5, 2005.

[11] J. Niels, “Putting it all in the trunk: incremental software development in the freebsd open source project,” *Information Systems Journal*, vol. 11, no. 4, pp. 321–336, 2001.

[12] T. Dinh-Trong and J. Bieman, “The freebsd project: A replication case study of open source development,” *Software Engineering, IEEE Transactions on*, vol. 31, no. 6, pp. 481–494, 2005.

[13] J. Han, C. wu, and B. Lee, “Extracting development organization from open source software,” in *16th Asia-Pacific Software Engineering Conference, IEEE.*, 2009, pp. 441–448.

[14] E. S. Raymond, “The new hacker’s dictionary (3rd ed.),” in *Cambridge, MA, USA: MIT Press*, 1996.

[15] L. M. Nyman and T. Mikkonen, “To fork or not to fork: Fork motivations in sourceforge projects,” in *Source Systems: Grounding Research : IFIP Advances in Information and Communication Technology*, 2011, pp. 259–268.

[16] T. Browning, “Applying the design structure matrix to system decomposition and integration problems: a review and new directions,” in *Engineering Management, IEEE Transactions on*, vol. 48, no. 3, 2001, pp. 292–306.

[17] M. E. Sosa, S. D. Eppinger, and C. M. Rowles, “The misalignment of product architecture and organizational structure in complex product development,” in *Management Science*, vol. 50, no. 12, 2004, pp. 1674–1689.

[18] I. Herraiz, J. Gonzalez-Barahona, G. Robles, and D. German, “On the prediction of the evolution of libre software projects,” in *ICSM*, oct. 2007, pp. 405 –414.

[19] FreeBSD, “http://www.freebsd.org/,” 2013.

[20] NetBSD, “http://www.netbsd.org/about/,” 2013.

[21] OpenBSD, “http://www.openbsd.org/,” 2013.

[22] J. Wu, R. Holt, and A. Hassan, “Empirical evidence for soc dynamics in software evolution,” in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, oct. 2007, pp. 244 –254.

[23] I. Herraiz, “A statistical examination of the evolution and properties of libre software,” in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, sept. 2009, pp. 439 –442.

[24] J. C. JE, L. V. LG, and A. Wolf, “Cost-effective analysis of in-place software processes,” in *IEEE Transactions on Software Engineering*, vol. 24, no. 8, 1998, pp. 650–663.

[25] D. Atkins, T. Ball, T. Graves, and A. Mockus, “Using version control data to evaluate the impact of software tools,” in *Proceedings 21st International Conference on Software Engineering*, vol. 24, no. 8, 1999, pp. 324–333.

[26] I. Kwan, M. Cataldo, and D. Damian, “Conway’s law revisited: The evidence for a task-based perspective,” *IEEE Software*, vol. 29, no. 1, pp. 90–93, 2012.

[27] M. Goeminne and T. Mens, “A framework for analysing and visualising open source software ecosystems,” in *Proceeding IWPSE-EVOL ’10*, 2010, pp. 42–47.

[28] D. M. German, “Using software trails to reconstruct the evolution of software,” in *JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE*, vol. 16, 2004, pp. 367–384.
[29] Y. Wang, D. Guo, and H. Shi, “Measuring the evolution of open source software systems with their communities,” in *ACM SIGSOFT Software Engineering Notes*, vol. 32, no. 6, 2007.
[30] W. Zhang, Y. Yang, and Q. Wang, “Network analysis of oss evolution: An empirical study on argouml project,” in *IWPSE-EVOL11*, 2011.
[31] jsoup: Java HTML Parser, “http://jsoup.org/,” 2013.
[32] U. S. C. Analysis and Metrics, “http://www.scitools.com/,” 2013.
[33] D. Darcy, S. Daniel, and K. Stewart, “Exploring complexity in open source software: Evolutionary patterns, antecedents, and outcomes,” in *Proceedings of the 43rd Hawaii International Conference on System Sciences*, 2010, pp. 1–11.
[34] M. Simmons, P. Vercellone-Smith, and P. Laplante, “Understanding open source software through software archaeology: The case of nethack,” in *Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*, 2006, pp. 47–58.
[35] A. P.-J. A. for Microsoft Documents, “http://poi.apache.org/,” 2013.
[36] J. Herbsleb and R. Grinter, “Architectures, coordination, and distance: Conway’s law and beyond,” in *Journal IEEE Software*, vol. 16, no. 5, 1999, pp. 63–70.
[37] M. Weiss, G. Moroiu, and P. Zhao, “Evolution of open source communities,” in *IFIP International Federation for Information Processing, Volume 203, Open Source Systems*, 2006, pp. 21–32.
[38] J. Gonzalez-Barahona, L. Lopez, and G. Robles, “Community structure of modules in the apache project,” in *Workshop on Open Source Software Engineering*, 2004.
[39] C. Baldwin and K. Clark, “Design rules: The power of modularity,” in *MIT Press*, 2000.
[40] F. P. Brooks, “The mythical man-month,” in *Anniversary Edition: Addison-Wesley Publishing Company*, 1995.
[41] G. Valetto, M. Helander, K. Ehrlich, S. Chulani, M. Wegman, and C. Williams, “Using software repositories to investigate socio-technical congruence in development projects,” in *ICSE Workshops MSR*, 2007, pp. 25–25.
[42] H. Dayani-Fard, Y. Yu, J. Mylopoulos, and A. Periklis, “Improving the build architecture of legacy c/c++ software systems,” in *8th FASE*, 2005.
[43] R. Kazman and S. Carrire, “Playing detective: Reconstructing software architecture from available evidence,” in *Technical Report CMU/SEI-97-TR-010, Carnegie Mellon University*, 1997.
[44] J. Gamalielsson and B. Lundell, “Sustainability of open source software communities beyond a fork: How and why has the libreoffice project evolved?” *Journal of Systems and Software*, vol. 89, pp. 128–145, 2014.
[45] B. Ray and M. Kim, “A case study of cross-system porting in forked projects,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 53.
[46] D. German, M. D. Penta, Y.-G. G. éhéneuc, and G. Antoniol, “Code siblings: Technical and legal implications of copying code between applications,” in *MSR’09*. IEEE, 2009, pp. 81–90.
[47] M. Syeed, I. Hammouda, and T. Systa, “The evolution of open source software projects: a systematic literature review,” *Journal of Software*, vol. 8, no. 11, pp. 2815–2829, 2013.

M.M. Mahbubul Syeed received his B.Sc degree in Computer Science and Information Technology from Islamic University of Technology, Bangladesh in September, 2002 and his M.Sc degree in Information Technology from Tampere University of Technology, Finland in April, 2010. He is currently working towards his Ph.D. degree and working as a researcher in the same university. His current research interest includes study of Open Source Software ecosystem, ecosystem enabling architecture, project evolution, experimental software development, and big data mining and knowledge extraction.

Dr. Imed Hammouda joined University of Gothenburg in September 2013. Before that, he was Associate Professor of software engineering at Tampere University of Technology (TUT), Finland. At TUT, he was heading the international masters programme at the Department of Pervasive Computing. He got his Ph.D. in software engineering from TUT in 2005. Dr. Hammouda’s research interests include open source software, software architecture, software development methods and tools, and variability management. He was a founding member and leader of TUTOpen - TUT research group on open source software. He has been the principal investigator of several research projects on various open initiatives. Dr. Hammouda’s publication record includes over fifty journal and conference papers.

[V] M.M. Syeed, K. Marius Hansen, I. Hammouda, K. Manikas. Socio-Technical Congruence in the Ruby Ecosystem. In *Proceedings of the 10th International Symposium on Open Collaboration (OpenSym)*, pages 2. ACM, August, 2014.

Socio-Technical Congruence in the Ruby Ecosystem

M. M. Mahbubul Syeed
Department of Pervasive Computing
Tampere University of Technology
Tampere, Finland
mm.syeed@tut.fi

Klaus Marius Hansen
Department of Computer Science (DIKU)
University of Copenhagen
Copenhagen, Denmark
klausmh@di.ku.dk

Imed Hammouda
Department of Computer Science and Engineering
Chalmers and University of Gothenburg
Gothenburg, Sweden
imed.hammouda@cse.gu.se

Konstantinos Manikas
Department of Computer Science (DIKU)
University of Copenhagen
Copenhagen, Denmark
kmanikas@di.ku.dk

ABSTRACT

Existing studies show that open source projects may enjoy high levels of socio-technical congruence despite their open and distributed character. Such observations are yet to be confirmed in the case of larger open source ecosystems in which developers contribute to different projects within the ecosystem. In this paper, we empirically study the relationships between the developer coordination activities and the project dependency structure in the Ruby ecosystem. Our motivation is to verify whether the ecosystem context maintains the high socio-technical congruence levels observed in many smaller scale FLOSS (Free/Libre Open Source Software) projects. Our study results show that the collaboration pattern among the developers in Ruby ecosystem is not necessarily shaped by the communication needs devised in the dependencies among the ecosystem projects.

1. INTRODUCTION

A software ecosystem has been defined as "a set of businesses functioning as a unit and interacting with a shared market for software and services, together with relationships among them. These relationships are frequently underpinned by a common technological platform and operate through the exchange of information, resources, and artifacts" [15]. The technological platform is often a software system providing various levels of openness for developers.

In the case of FLOSS (Free/Libre Open Source Software) ecosystems, the software platform is typically organized into smaller inter-dependent projects attracting a developer community consisting of a large number of volunteers in addition to paid developers. Popular examples of such ecosystems include GNOME [10], Eclipse [7] and Ruby [26].

FLOSS ecosystems have been the subject of much socio-technical research (cf. e.g [14]) studying the possible relationships between their social domain represented by developer community and their technical domain associated with the software produced. It is in this context that the paper seeks to explore the mapping between the communication patterns of the developer community and the architectural dependencies among the projects within an FLOSS ecosystem.

In this work, we examine the mapping through the notion of socio-technical congruence that puts into test a well-known but insufficiently understood phenomenon known as Conway's Law [5]. We present an empirical evaluation of the law to show to what extent developers' activities within an FLOSS ecosystem can be used to approximate dependencies between the ecosystem projects. As our unit of study, we use the Ruby FLOSS ecosystem.

The study builds on top of an earlier work where we have verified Conway's Law in the context of a single FLOSS project [31]. Thus the contribution of this work is to take such empirical evaluation to an ecosystem level that may consist of thousands of interrelated projects and explore the extent to which Conway's law may hold.

The remaining of the paper is organized as follows. Section 2 introduces a number of key concepts that this study uses and the research questions explored. Section 3 presents our study design. Results are reported and discussed in Section 4, followed by a discussion of related work in Section 5. Possible limitations and threats to validity are highlighted in Section 6. Finally, Section 7 concludes the paper and shed light on future research.

2. DEFINITIONS AND RESEARCH QUESTIONS

In this section we define the set of concepts used in this study.

2.1 Conway's Law

Conway's Law, in its purest form, states that "organizations which design systems are constrained to produce systems which are copies of the communication structures of these organizations" [5]. In other words, the software product architecture reflects the organizational structure of its development team [5, 19]. In [12], Conway's Law is considered bidirectional and thus claimed to be true in reverse as well. This means the communication pattern within a developer community should reflect the architectural dependency in the developed software. Thus, Conway's Law can effectively be interpreted as the basis for studying the social and technical interdependency within a software project [24].

2.2 Socio-technical congruence

The recently defined phenomenon of 'socio-technical congruence' is an operationalization of Conway's Law. Socio-technical congruence can be defined as the match between the coordination needs established by the technical domain (i.e., the architectural dependency in the software) and the actual coordination activities carried out by project members (i.e., within the members of the developer community) [19]. This coordination need can be determined by analyzing the assignments of persons to a technical entity such as a source code module, and the technical dependencies among the technical entities [19]. Accordingly, for socio-technical congruence to be present, developers within the community should communicate if there exists a communication need indicated by technical dependencies. For example, developers working on the same module or on the interdependent modules should be coordinating.

2.3 Explicit Architecture

In general, the 'Explicit Architecture' of a software system is defined as the system structure as present in technical entities and dependencies among technical entities. In our study, the explicit architecture presents relationship among the gems in the Ruby ecosystem — a gem is a software package that contains a Ruby application or library. A relationship in this architecture represents the development and runtime dependency between two gems.

2.4 Explicit Coordination Network

The 'Explicit Coordination Network' is a social network in which two developers have a relationship if they have direct communication history, either social or technical. In the Ruby study, the communication history is deduced using the communication traces for Ruby gems hosted on the GitHub[9] software development hosting site issue tracking system.

2.5 Implicit Architecture

The 'Implicit Architecture' of a software system is defined by the elements of the Explicit Architecture and the relationships of the Explicit Coordination Network. More specifically, in the implicit architecture we identify a relationship between two elements (technical entities) if there is direct communication between any of the developers of the two elements.

In the Ruby ecosystem study, we define the Implicit Architecture of the complete Ruby ecosystem. Here, two gems are related if there are developers who have either (a) contributed to both the gems, or (b) have direct communication (e.g., one to one issue related conversation). For instance, consider that developer D_1 has contributed to gems G_1 and G_2, and developer D_2 has contributed to gem G_3. Also consider that both developers has direct communication at organizational level as shown in Fig. 1(a). Thus according to the definition, gems G_1, G_2 and G_3 are linked to each other in the Implicit Architecture (Fig. 1(b)).

2.6 Research questions

In this work, we study socio-technical congruence in the Ruby ecosystem by addressing two research questions:

1. *Does the study of socio-technical congruence has any significance at the ecosystem level?*
2. *To what extent developer's interaction within the Ruby ecosystem can approximate the actual relationship (or dependencies) among the ecosystem gems?*

In order to perform a study required by Research Question 1, data should be available related to inter-developer communication, developer contribution to the ecosystem projects, and dependencies between the individual projects. While, Research Question 2 will be given in terms of a socio-technical congruence level.

In order to address these questions, we use social network analysis techniques to analyze data collected from the RubyGems.org and GitHub repositories (see Section 3).

3. STUDY DESIGN

This section presents in detail our study design, covering discussion on the case study selection, required data sets, data acquisition, cleaning, and analysis process.

3.1 Case and Subject Selection

We use the Ruby gems ecosystem as a case in order to explore our research questions. The Ruby gems website, RubyGems.org [27], hosts packages ("gems") for the Ruby programming language. A gem contains code, documentation, and a specification. For this study, we use the specification only. Gem developers can create gems and push these to RubyGems.org while gem users (typically application developers) can install gems using RubyGems.org. Pushing and installing is typically done via the `gem` command line tool that interacts with RubyGems.org.

Two properties of RubyGems.org makes it appropriate for studying soci-technical dependencies in an ecosystem. First, RubyGems.org is very widely used: on 2014-05-02 it stated that 3,020,455,028 downloads had been made of 74,800 gems since July 2009. Secondly, RubyGems.org makes it possible to couple gems with data on the development process since most gems use GitHub[9] as a source code repository and for collaboration. In our data set (see Section 3.2), 72% of the specifications of gems referenced GitHub.

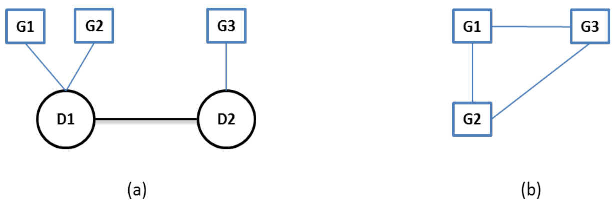


Figure 1: (a) Explicit Coordination Network with contribution to code base (b) Corresponding Implicit Architecture

3.2 Data Sets

We collect data for i) the Ruby ecosystem architecture and ii) the Ruby ecosystem coordination network.

To collect data for i), we use the specifications of gems from RubyGems.org. The specification contains metadata that include the gem name, dependencies to other gems, and URIs for the gem. The following listings shows an excerpt of the `aasm` gem specification in JSON format. The gem's development dependencies (that are needed to further develop the gem) include the `mime-types` gem (in a version greater than or equal to 1.25.0 and less than 2.0) and the `rake` gem (in any version). The `aasm` gem has no runtime dependencies (that are needed to run the gem). Finally, the gem's homepage is `https://github.com/aasm/aasm`.

```
{
  "name": "aasm",
  "info": "AASM is a continuation of the acts as state
       machine rails plugin, built for plain Ruby objects
       .",
  "dependencies": {
    "development": [
      {
        "name": "mime-types",
        "requirements": "~> 1.25"
      },
      {
        "name": "rake",
        "requirements": ">= 0"
      }
      ...
    ],
    "runtime": []
  },
  "homepage_uri":"https://github.com/aasm/aasm"
  ...
}
```

To collect data for ii), we use GitHub. In the example above, the GitHub project related to the gem can be identified as `aasm` (with owner `aasm`). On GitHub, collaboration is facilitated via among others issues and pull requests created by GitHub users. To, e.g., suggest fixes to `aasm`, a developer may create a branch, make modifications, and create a "pull request" for the `aasm` members to merge the modifications into the main `aasm` repository. For the `aasm` gem, e.g., there were (on 2014-04-28), 119 issues for the `aasm`. In the listing below, issue number 62 for `aasm` shows an example of two GitHub users collaborating. The issue is created via a pull request by the user `Nitrodist` and suggest a "minor fix" that is closed by the user `alto`.

```
{
  "number": 62,
  "title": "Fix migration example in README",
  "user": {
    "login": "Nitrodist",
    ...
  }
  "comments": 1,
  "body": "Minor fix! :heart: ",
  "closed_by": {
    "login": "alto",
    ...
  }
  ...
}
```

3.3 Data Collection

Data collection was done in three steps: i) retrieve a list of gems from RubyGems.org, ii) identify GitHub project for gems, iii) retrieve issues from GitHub projects.

Regarding i), data was collected from RubyGems.org using the command line API. Gems starting with the numbers 0 to 9 and ASCII characters a-z were collected using the `gem list` command. For example, the command

```
gem list -r a
```

retrieves (for the `gem` tool used in this study), a list of gems starting with the letter "a". Using this list, we used the RubyGems.org HTTP API to fetch specifications of the latest version of the gem. For example, the specification of the latest version of the `aasm` gem is available at `https://rubygems.org/api/v1/gems/aasm.json`. In this way, we retrieved 60,286 gem specifications on 2013-08-19.

Based on the list of gems, for step ii), we scanned the URIs of the gem specifications to find GitHub URIs. We assume that if a URI contains a GitHub URI, it is the URI of the gem project. We prioritized `source_code_uris` and `project_uris`. We were able to retrieve GitHub issues (possibly an empty set) from 33,960 GitHub projects in step iii).

For step iii), we retrieved all open and closed issues and comments for identified projects using the GitHub API. Because the GitHub API is rate limited we retrieved GitHub data over several days. Since we cannot link gem versions to GitHub, we assume that the data collected at GitHub represent the full history of collaboration up until the latest version of the gem. For 56% of Ruby gems, we could find GitHub data. For the remaining gems, we were either unable to identify a GitHub URL using the method described above or the gem was not developed using GitHub.

3.4 Data Refining and Structuring

In order to initiate data analysis, we refined and restructured the collected data as follows: for each Ruby gem for which we could find GitHub data, a record was created in JSON format that contains, the gem and owner names, the list of gems it depends on, and pair-wise communication between developers of the gems. Records created for the gems were collected in a single JSON file, which contains 42,803 gem records. The following listings shows the record format taking *rumember* gem as an example.

```
{
 "gem_name": "rumember",
 "github_owner": "tpope",

 "dependencies": [
      "json",
      "launchy",
      "rspec"
                 ],

 "relationships": [
    [
      "mofus",
      "tpope"
    ],
    [
      "kevincolyar",
      "tpope"
    ],
    ...
                  ]
}
```

In this listing, the Ruby gem *rumember* is owned by *tpope*. The gem has dependencies to three other gems, namely, *json*, *launchy*, *spec*, denoted by the *dependencies* structure. Pair-wise developer login names presented in *relationships* structure shows their communication. For instance, the developer *mofus* has communicated with *tpope*.

Among the 42,803 Ruby gems record listed in the JSON file, 12,520 Ruby gems records have developer communication records in GitHub. Thus, for further analysis, we restricted the data set to 12,520 Ruby gems records.

3.5 Data Analysis

This section is focused on topics related to construction of the architectures (both explicit and implicit), explicit coordination network and their use in measuring socio-technical congruence utilizing the data presented in Section 3.4.

Explicit Architecture: The Explicit Architecture shows relationships among the gems. Relationships were generated based on the development and runtime dependencies that exists between Ruby gems (presented in Section 2.3 and 3.2). At implementation level, this architecture is generated by creating edges between a gem and gems to which it has dependency by utilizing the dependency list of that gem. For instance, Figure 2(a) presents an Explicit Architecture that corresponds to the dependency record presented in Section 3.4 for the gem *rumember*. The complete architecture consists of 141,029 edges among the 12,520 gems, edge weights of which ranges between 1 and 33. A partial snapshot of this architecture is shown in Figure 2(b).

Explicit Coordination Network: Following the definition in Section 2.4, the Explicit Coordination Network was derived among the developers contributing to the gems. At implementation level, an edge is created between each pair of developer names listed in the gem records presented in Section 3.4. For instance, Figure 3(a) presents an Explicit Coordination Network that corresponds to the developer relationships present in the *relationship* structure for the record of the gem *rumember*. The complete network consists of 186,136 edges among the 55,454 developers, edge weights of which ranges between 1 and 46. A partial snapshot of this network is shown in Figure 3(b).

Implicit Architecture: The Implicit Architecture was generated following the definition in Section 2.5. For doing this, we restricted the edge weight of the Explicit Coordination Network to ≥ 2. This is done because an edge weight of 1 in this network represents only one instance of interaction between two developers, which is insignificant to consider it as a collaboration between developers. This filtering reduces the size of Explicit Coordination Network to 59,562 edges. Furthermore, we generated seven Implicit Architectures based on seven edge weight thresholds of the Explicit Coordination Network. This weight categorization of the coordination network is shown in the second column of Table 1 and the size of corresponding Implicit Architecture is presented in column 4. This catagorization was done to comprehend how congruence values change with the changes in coordination strength seen in Explicit Coordination Network.

Measuring Congruence:

Congruence was measured following the similarity measure presented in equation (1). This measure is analogous to fit / congruence measure used in organizational theory method [4], and already been applied in [31] for measuring congruence in FreeBSD project.

$$Congruence = \frac{|Ref_A \bigcap Analogous_A|}{|Ref_A|} \times 100 \qquad (1)$$

In the above equation, Ref_A is the reference architecture (either explicit or implicit), and $Analogous_A$ it the analogous architecture (either explicit or implicit) with which congruence will be measured.

This equation measures congruence between the two architectures with respect to the reference one, Ref_A. Therefore, the numerator of equation (1) identifies the commonalities between the two given architectures, then divided by the size of the reference architecture and expressed as a percentage.

To compute socio-technical congruence using the similarity measure in (1), following approach was applied: the explicit architecture was taken as the reference architecture (Ref_A) and the implicit one was taken as the analogous architecture ($Analogous_A$). The intersection operation in numerator was carried out between the explicit architecture and each of the 7 implicit architectures that were generated for the 7 weight categories in explicit coordination network. This operation identifies the number of edges (or relationships) that are identical for both the architectures. Result of this process is presented in column 5 of Table 1. Therefore, this measure illustrates verification of Conway's law, that reveals the

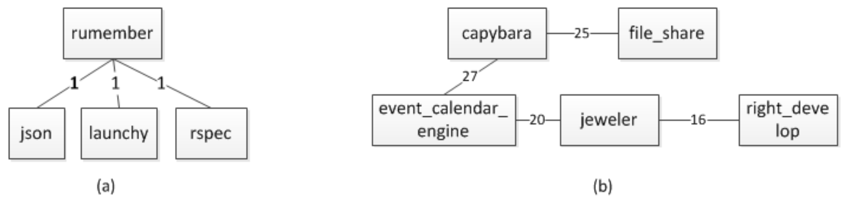


Figure 2: (a) Example Explicit Architecture for gem *rumember* (b) A partial snapshot of the Explicit Architecture

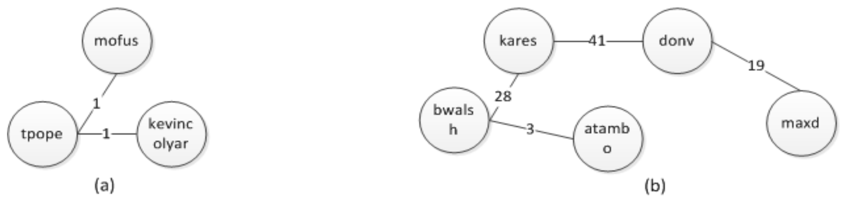


Figure 3: (a) Example Explicit Coordination Network for gem *rumember* (b) A partial snapshot of the Explicit Coordination Network

match between the dependency among the Ruby gems and the gems dependency produced due to the communication and collaboration structure of the developers.

Then to identify the extent to which the implicit architectures approximate the explicit, we calculated the similarity measure in (1). The resulted congruence measures are presented in column 6 of Table 1.

4. RESULT ANALYSIS

In this section we investigate the research questions primarily based on the data analysis presented in Section 3.

4.1 Social and Technical Dependencies in the Ruby Ecosystem

Socio-technical congruence has often been studied within a project to determine coordination quality and its consequences. Such studies make sense, because, a project often organizes itself around the products' architecture. In this setup, the main components of the product define the organization's key subtasks [13] and become the source of most relevant information pertinent to the task dependencies that define coordination need among the developers [6].

However, the term 'software ecosystem' is used in FLOSS software to refer to a collection of software projects that are developed and evolve together in the same environment [21]. Thus, initiating the study of socio-technical congruence within an ecosystem, can only be feasible if the projects within that ecosystem have dependencies and cooperation, both from technical and social perspectives. In other words, projects should have technical dependencies among them, while the developers working on those projects have communication and collaboration.

This verification in the case of the Ruby ecosystem was done by examining the explicit architecture and the explicit coordination network. The explicit architecture reveals 141029 number of edges among the 12520 gems. Each edge in this architecture shows either development or runtime dependency between two gems. Similarly, the explicit coordination network generates 186136 number of relationship edges among 55454 developers. Each edge in this network shows communication between two developers as shown by the GitHub issue tracking system. This quantity of relationships among the projects, both from social and technical domains, sets the favorable ground for undertaking the socio-technical congruence measure in Ruby ecosystem.

4.2 Socio-Technical Congruence in the Ruby Ecosystem

Result obtained from the socio-technical congruence measure carried out for Ruby gems ecosystem is reported in Table 1. Observed results positioned this study to offer the following insights:

The congruence measure, as a whole, is significantly low for all the 7 edge weight categories of explicit coordination network, as shown in the trend chart in Figure 4. This chart plots the congruence measure against the edge weight limit of the coordination network. According to this chart, for edge weight ≥ 2, the congruence measure is 76.2%, which drops sharply with the increasing edge weight limit. For instance, congruence value drops to 46.3% for edge weight ≥ 8, which goes down as low as 36.1% for weight ≥ 19.

As explained in Section 2.4, the edge weight in the explicit coordination network depicts the strength of collaboration among the developers. Thus, communication and collaboration get strongly tied with the increased edge weight in this network. The stated congruence measure, in this connection, shows that congruence measure decreases with the increased developer collaboration. This observation led us to infer that

The collaboration pattern among the developers in Ruby ecosystem is not necessarily shaped by the communication needs devised in the dependencies among the gems.

The following explanations can be offered in support to this

Table 1: Socio-Technical Congruence in Ruby Ecosystem

Constraints		Edge count of the Architectures		Congruence	
No of Gems Selected	Weight limit for Explicit Coordination Network	Explicit Architecture (Ref_A)	Implicit Architecture ($Analogous_A$)	Intersection count ($Ref_A \bigcap Analogous_A$)	Congruence $= \frac{Ref_A \bigcap Analogous_A}{\|Ref_A\|} \times 100)$
12520	≥ 2	28361	10472491	21604	76.2
	≥ 8	28361	3283494	13121	46.3
	≥ 9	28361	2765429	12269	43.3
	≥ 10	28361	2448552	11737	41.4
	≥ 12	28361	2283553	11488	40.6
	≥ 14	28361	1948130	10613	37.5
	≥ 19	28361	1843492	10211	36.1

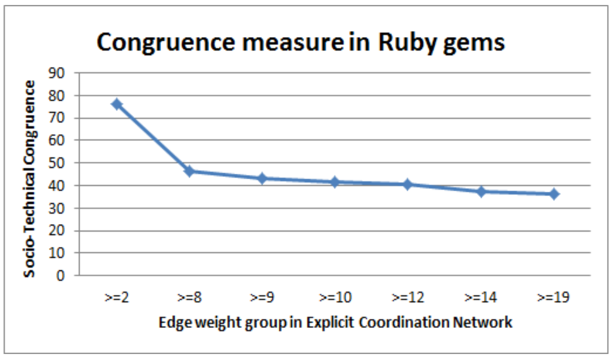


Figure 4: Socio-Technical Congruence in Ruby Ecosystem

inference. Interdependency among projects in an ecosystem often exist at a higher abstraction level. For instance, in the case of the Ruby ecosystem, the gems' dependencies are due to development and runtime dependencies. A development dependency defines a gem that is necessary at development time for further development, whereas a runtime dependency represents a gem that is necessary at runtime. Therefore, such dependencies can not define the concrete task dependencies at the development level that could necessarily devise the coordination needs among developers responsible for those tasks. It is thus possible that the developers who have extensive collaboration (as seen in the Explicit Coordination Network) belong to the same gem or related gems that have dependency at development level. For instance, the Explicit Coordination Network that have edge weight ≥ 19 contains 362 developers. Around 79% of these developers (285 developers out of 362) works for the same gem *jdbc-jtds*. Therefore, it is obvious that these 285 developers should have extensive communication and collaboration, as their development tasks are bound to have technical dependencies. Thus strong socio-technical congruence, as proposed by [20], might imply better coordination among the developers, which can give ground to better support the management of changes and maintain quality.

However, Socio-Technical congruence within an FLOSS project have already been measured in [31]. In this paper, Socio-Technical congruence has been measured using equation (1) during the entire lifespan of FreeBSD project. Reported result identified that congruence is significantly higher in the FreeBSD project which has a stable evolution history for the last seven stable releases of the project. This implies that the collaboration pattern of the FreeBSD developers are due to the communication need established by the dependencies within the software components that are contributed by them. We argue that similar congruence measure may be seen for a gem (e.g., *jdbc-jtds*) in the Ruby ecosystem.

In summary, this discussion lead us to conclude that strong socio-technical congruence exists among developers within a project, which, however, decreases significantly at ecosystem level.

5. RELATED WORK

In this section we highlighted the prior works that fall within the scope of this research.

5.1 Socio-Technical Congruence

In literature it has been stated that high degree of congruence between the social and technical domain is a natural consequence and a desired property for collaborative development activities [2],e.g., software engineering. Such claim has also been accredited in other studies: for instance, studies that identified that effective Socio-Technical alignment in a project offers faster completion of modification requests [4] with higher build success [20] and product quality [1].

On the other hand, lack of Socio-Technical congruence is often subjected to lower productivity with increased number of code changes [3][8] and negative performance level [30] within the organization. It is, thus, advised to measure congruence to evaluate the actual coordination quality within the organization [3].

5.2 Study of OSS Ecosystems

Social interaction in open source software ecosystems, i.e. the interaction and dependencies between developers, and the effect it has to the software produced but also to the ecosystem as a whole, is a perspective of software ecosystems that has been the focus in a number of studies. In this context, Kabbedijk and Jansen [17] analyze the Ruby Git repository, graph the developer and gem interaction and define three developer roles according to their analysis. Scacchi [28, 29] analyses different perspectives of free and open source software development (FOSSD) underlining the concept of multi-project (FOSSD) software ecosystem, a set of different FOSSD projects under the same repository. He states that software evolution in this kind of ecosystems depends on a number of parameters, people (developers) and their interaction being one of them. Raj and Srinivasa [18] analyze the developer contribution in sourceforge.net to reveal the tendency of developers to contribute to single projects in the repository. Jergensen et al. [16] study GNOME developer participation and showed that the onion model hypothesis of new contributors in FOSS projects does not apply to this project. While Ververs et al. [32] study how development activity changes before and after events (e.g. commits) in the Debian ecosystem and argue that frequent events in the ecosystems ensure developer commitment.

Moreover, the literature reports on a number of tools for analyzing open source ecosystems where the analysis mainly focuses on user interaction and software evolution [22, 23, 11]. Software evolution in ecosystems is also addressed by Yu et al. [33, 34] where they analyse the evolution of software from the biological viewpoints: evolution in term of symbiosis and in terms of darwinism. While, Robbes et al. [25] study the evolution of OSS software in terms of the Ripple effect, i.e. the effects to software when APIs change.

6. THREATS TO VALIDITY

The following aspects have been identified which could lead to threats to validity of this study.

External validity (how results can be generalized): This paper disseminates an empirical study of the Conway's Law in an ecosystem level. Our empirical data are collected from the Ruby ecosystem, where we apply the theory. The fact that we only study the Ruby ecosystem and that this is the only study of this of the Conway's Law in an ecosystem level, at least to our knowledge, allow for questioning the extent to which our results can be generalised. Our study intends to trigger additional studies of this kind, in different ecosystems, in order to reach closer to a conclusion that can be generalised.

Internal validity (confounding factors can influence the findings): When examining the empirical data of our study, we note that we could only retrieve the GitHub issues for 56% of the total Ruby gems, although 72% of the gem specification referenced GitHub, successfully extracted the issues of a total of $33,960$ GitHub projects. This is either because our method was unable to identify a valid GitHub URL or the gems were not developed using GitHub. The high number of unidentified issues can pose threats to the validity of the study.

Construct validity (relationship between theory and observation): Among the identified GitHub gems we limited our study to 12,520 gems, as only for those gems we were able to identify developer communication data. The reasoning here is that implicit architecture that was generated based on explicit coordination network, would only contain relationships among these 12520 projects. Thus considering the whole population of gems would lead to biased observation. This might pose threat to construct validity to this study.

7. CONCLUSIONS

In this paper, we studied the socio-technical congruence, and the significance of Conway's Law, in the context of the Ruby FLOSS ecosystem. Our study shows that the congruence measure in Ruby is relatively low, which indicates that the collaboration pattern among the developers in the Ruby ecosystem is not necessarily shaped by the communication needs devised in the dependencies among its ecosystem projects. In contrast, the individual ecosystem projects themselves still enjoy higher levels of congruence.

Our findings can be explained by the fact that developers often communicate with peers involved in the same projects, which offer a narrow enough context for collaboration. In the case of Ruby, the ecosystem level turns out be too broad for such communication activities. From ecosystem heath perspective, a low congruence level could mean that developers might be unaware of important project dependencies or might have missed opportunities to collaborate with other relevant ecosystem developers.

As future, work we plan to investigate the inverse Conway's Law in the Ruby ecosystem examining whether the projects dependency structures can themselves be used to approximate the coordination network between developers.

8. ACKNOWLEDGEMENTS

This work has been partially funded by the TUT Graduate school, Finland.

9. REFERENCES

[1] F. P. Brooks. The mythical man-month. *Anniversary Edition: Addison-Wesley Publishing Company*, 1995.

[2] T. Browning. Applying the design structure matrix to system decomposition and integration problems: a review and new directions. *IEEE Transactions on Engineering Management*, 48(3):292–306, 2011.

[3] M. Cataldo, P. Wagstrom, J. Herbsleb, and K. Carley. Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In *Proceedings of the 2006 conference on Computer Supported Cooperative Work*, pages 353–363. ACM, 2006.

[4] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley. Identification of coordination requirements: Implications for the design of collaboration and awareness tools. *CSCW'06*, 2006.

[5] M. E. Conway. How do committees invent? *Datamation*, 14(4):28–31, 1968.

[6] R. Daft and K. Weick. Towards a model of organizations as interpretation systems. In *Academy of Management Review*, volume 9, pages 284–295, 1984.

[7] Eclipse – the eclipse foundation open source community website. `http://www.eclipse.org`. Accessed 2014-05-02.

[8] K. Ehrlich, M. Helander, G. Valetto, S. Davies, and C. Williams. An analysis of congruence gaps and their effect on distributed software development. *Proc. Socio-Technical Congruence Workshop at ICSE Conf.*, 2008.

[9] GitHub. Build better software, together. `https://www.github.com`. Accessed 2014-05-02.

[10] GNOME. `http://www.gnome.org`. Accessed 2014-05-02.

[11] M. Goeminne and T. Mens. A framework for analysing and visualising open source software ecosystems. In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and*

International Workshop on Principles of Software Evolution (IWPSE), IWPSE-EVOL '10, pages 42–47, New York, NY, USA, 2010. ACM.

[12] J. Han, C. Wu, and B. Lee. Extracting development organization from open source software. In *16th Asia-Pacific Software Engineering Conference, IEEE.*, pages 441–448, 2009.

[13] E. V. Hippel. Task partitioning: an innovation process variable. In *Research Policy*, volume 19, pages 407–418, 1990.

[14] S. Jansen, S. Brinkkemper, and M. A. Cusumano. *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry.* Edward Elgar Publishing, 2013.

[15] S. Jansen, A. Finkelstein, and S. Brinkkemper. A sense of community: A research agenda for software ecosystems. In *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 187–190. IEEE, 2009.

[16] C. Jergensen, A. Sarma, and P. Wagstrom. The onion patch: migration in open source ecosystems. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 70–80, New York, NY, USA, 2011. ACM.

[17] J. Kabbedijk and S. Jansen. Steering insight: An exploration of the ruby software ecosystem. In B. Regnell, I. Weerd, O. Troyer, W. Aalst, J. Mylopoulos, M. Rosemann, M. J. Shaw, and C. Szyperski, editors, *Software Business*, volume 80 of *Lecture Notes in Business Information Processing*, pages 44–55. Springer Berlin Heidelberg, 2011. 10.1007/978-3-642-21544-5_5.

[18] R. P. M. Krishna and K. G. Srinivasa. Analysis of projects and volunteer participation in large scale free and open source software ecosystem. *SIGSOFT Softw. Eng. Notes*, 36:1–5, March 2011.

[19] I. Kwan, A. Schroter, and D. Damian. Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. In *IEEE Trans. Software Eng.*, volume 37, pages 307–324, 2011.

[20] I. Kwan, A. Schröter, and D. Damian. Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. *IEEE Transactions on Software Engineering*, 37(3):307–324, 2011.

[21] M. Lungu, M. Lanza, T. Gîrba, and R. Robbes. The small project observatory: visualizing software ecosystems. In *Science of Computer Programming*, volume 75, pages 264–275, 2010.

[22] M. Lungu, M. Lanza, T. Gîrba, and R. Robbes. The small project observatory: Visualizing software ecosystems. *Science of Computer Programming*, 75(4):264 – 275, 2010. Experimental Software and Toolkits (EST 3): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT 2008).

[23] J. Pérez, R. Deshayes, M. Goeminne, and T. Mens. Seconda: Software ecosystem analysis dashboard. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 527 –530, march 2012.

[24] E. S. Raymond. The new hacker's dictionary (3rd ed.). In *Cambridge, MA, USA: MIT Press*, 1996.

[25] R. Robbes and M. Lungu. A study of ripple effects in software ecosystems (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 904–907, New York, NY, USA, 2011. ACM.

[26] Ruby programming language. `https://www.ruby-lang.org`. Accessed 2014-05-02.

[27] RubyGems.org. Your community gem host. `https://www.rubygems.org`. Accessed 2014-05-02.

[28] W. Scacchi. Free/open source software development: recent research results and emerging opportunities. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers*, ESEC-FSE companion '07, pages 459–468, New York, NY, USA, 2007. ACM.

[29] W. Scacchi. The future of research in free/open source software development. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 315–320, New York, NY, USA, 2010. ACM.

[30] M. Sosa, S. Eppinger, and C. Rowles. The misalignment of product architecture and organizational structure in complex product development. *Management Science*, 12(50):1674–1689, 2004.

[31] M. Syeed and I. Hammouda. Socio-technical congruence in oss projects: Exploring conway's law in freebsd oss evolution. In *Proceedings of 9th International Conference of Open Source Systems (OSS), Springer*, pages 109–126, 2013.

[32] E. Ververs, R. van Bommel, and S. Jansen. Influences on developer participation in the debian software ecosystem. In *Proceedings of the International Conference on Management of Emergent Digital EcoSystems*, MEDES '11, pages 89–93, New York, NY, USA, 2011. ACM.

[33] L. Yu, S. Ramaswamy, and J. Bush. Software evolvability: An ecosystem point of view. In *Software Evolvability, 2007 Third International IEEE Workshop on*, pages 75 –80, oct. 2007.

[34] L. Yu, S. Ramaswamy, and J. Bush. Symbiosis and software evolvability. *IT Professional*, 10(4):56 –62, july-aug. 2008.

[VI] M.M. Syeed, and I. Hammouda. Who Contributes to What? Exploring Hidden Relationships Between FLOSS Projects. In *Proceedings of the 10th IFIP WG 2.13 International Conference of Open Source Systems (OSS'2014)*, pages 21–30. Springer, May, 2014.

Who Contributes to What? Exploring Hidden Relationships between FLOSS Projects

M.M. Mahbubul Syeed[1] and Imed Hammouda[2]

[1] Department of Pervasive Computing,
Tampere University of Technology, Finland
mm.syeed@tut.fi

[2] Department of Computer Science and Engineering,
Chalmers and University of Gothenburg, Sweden
imed.hammouda@cse.gu.se

Abstract. In this paper we address the challenge of tracking resembling open source projects by exploiting the information of which developers contribute to which projects. To do this, we have performed a social network study to analyze data collected from the Ohloh repository. Our findings suggest that the more shared contributors two projects have, the more likely they resemble with respect to properties such as project application domain, programming language used and project size.

1 Introduction

With the exponential increase of Free/Libre Open Source (FLOSS) projects [6], searching for resembling open source components has become a real challenge for adopters [7]. By **resemblance**, we mean similarity factors between projects such as features offered, technology used, license scheme adopted, or simply being of comparable quality and size levels.

In this paper we exploit the information of 'which developers contribute to which FLOSS projects' to identify resembling projects. Our assumption is that if a developer contributes to several projects, simultaneously or at different times, then there might be implicit relationships between such projects. For example, one FLOSS project may use another as part of its solution [17] or two projects could be forks of a common base [16].

The idea of collecting and studying data of who contributes to which FLOSS projects is not new. The question has been the focus of many studies due to its relevance from many perspectives. For instance, the question has been significant for companies who want to identify who influences and controls the evolution of a specific project of interest[2], or to explore the social structure of FLOSS development [1], or simply to study what motivates people to join open source communities [3]. In this work, we address the following research questions:

1. How do FLOSS project development communities overlap?
2. To what extent can developer sharing in FLOSS projects approximate resemblance between the projects themselves?

For answering these questions, we used social network analysis techniques to analyze data collected from the Ohloh repository [4].

2 Study Design

This section presents in detail our study design, covering discussion on the data sources, required data sets, data acquisition, cleaning, and analysis process along with validation and verification of the analysis process.

2.1 Data Source

For this study we selected Ohloh data repository [4], which is a free, public directory of open source software projects and the respective contributors.

Ohloh collects and maintains development information of over 400 thousand FLOSS projects, and provide analysis of both the codes history and ongoing updates, and attributing those to specific contributors. It can also generate reports on the composition and activity of project code bases. These data can be accessed and downloaded through a set of open API which handles URL requests and responses [4]. The response data is expressed as an XML file, an example of which is shown in Fig. 1.

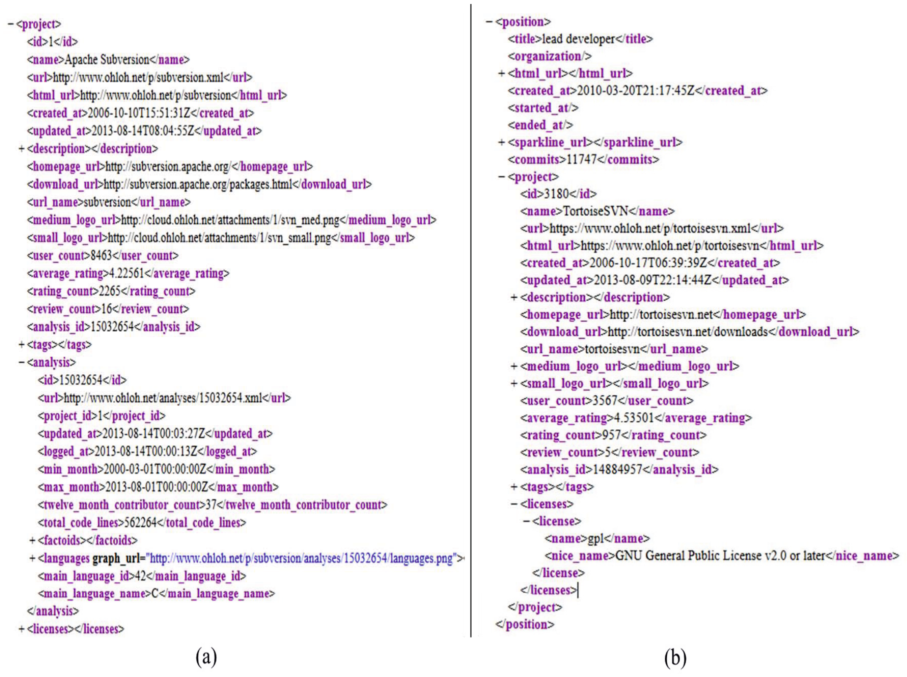
Our selection of Ohloh data repository is predominantly influenced by the following factors: (a) Ohloh data can be publicly reviewed, which in turn makes it one of the largest, most accurate, and up-to-date FLOSS software directories available; (b) the use of Ohloh repository makes FLOSS data available in a cleaned, unified and standard platform independent format. This makes the process of data analysis and visualization independent of technology, and data repository.

2.2 Data Collection

The following information has been collected from Ohloh repository in relation to this study:

Developer Account Information: An Account represents an Ohloh member, who is ideally a contributor to one or more FLOSS projects. Ohloh records a number of properties (or attributes) for an account.Among thousands of registered members, we collected account information of top 530 contributors according to assigned kudo rank of 10 and 9. Kudo rank is a way of appreciation to the FLOSS contributors through assigning a number between 1 and 10 for every Ohloh account [5].

Project Data: A Project represents a collection of source code, documentation, and web site data presented under a set of attributes, a list of which can



```
- <project>
    <id>1</id>
    <name>Apache Subversion</name>
    <url>http://www.ohloh.net/p/subversion.xml</url>
    <html_url>http://www.ohloh.net/p/subversion</html_url>
    <created_at>2006-10-10T15:51:31Z</created_at>
    <updated_at>2013-08-14T08:04:55Z</updated_at>
  + <description></description>
    <homepage_url>http://subversion.apache.org/</homepage_url>
    <download_url>http://subversion.apache.org/packages.html</download_url>
    <url_name>subversion</url_name>
    <medium_logo_url>http://cloud.ohloh.net/attachments/1/svn_med.png</medium_logo_url>
    <small_logo_url>http://cloud.ohloh.net/attachments/1/svn_small.png</small_logo_url>
    <user_count>8463</user_count>
    <average_rating>4.22561</average_rating>
    <rating_count>2265</rating_count>
    <review_count>16</review_count>
    <analysis_id>15032654</analysis_id>
  + <tags></tags>
  - <analysis>
      <id>15032654</id>
      <url>http://www.ohloh.net/analyses/15032654.xml</url>
      <project_id>1</project_id>
      <updated_at>2013-08-14T00:03:27Z</updated_at>
      <logged_at>2013-08-14T00:00:13Z</logged_at>
      <min_month>2000-03-01T00:00:00Z</min_month>
      <max_month>2013-08-01T00:00:00Z</max_month>
      <twelve_month_contributor_count>37</twelve_month_contributor_count>
      <total_code_lines>562264</total_code_lines>
    + <factoids></factoids>
    + <languages graph_url="http://www.ohloh.net/p/subversion/analyses/15032654/languages.png">
      <main_language_id>42</main_language_id>
      <main_language_name>C</main_language_name>
    </analysis>
  + <licenses></licenses>
```

(a)

```
- <position>
    <title>lead developer</title>
    <organization/>
  + <html_url></html_url>
    <created_at>2010-03-20T21:17:45Z</created_at>
    <started_at/>
    <ended_at/>
  + <sparkline_url></sparkline_url>
    <commits>11747</commits>
  - <project>
      <id>3180</id>
      <name>TortoiseSVN</name>
      <url>https://www.ohloh.net/p/tortoisesvn.xml</url>
      <html_url>https://www.ohloh.net/p/tortoisesvn</html_url>
      <created_at>2006-10-17T06:39:39Z</created_at>
      <updated_at>2013-08-09T22:14:44Z</updated_at>
    + <description></description>
      <homepage_url>http://tortoisesvn.net</homepage_url>
      <download_url>http://tortoisesvn.net/downloads</download_url>
      <url_name>tortoisesvn</url_name>
    + <medium_logo_url></medium_logo_url>
    + <small_logo_url></small_logo_url>
      <user_count>3567</user_count>
      <average_rating>4.53501</average_rating>
      <rating_count>957</rating_count>
      <review_count>5</review_count>
      <analysis_id>14884957</analysis_id>
    + <tags></tags>
    - <licenses>
      - <license>
          <name>gpl</name>
          <nice_name>GNU General Public License v2.0 or later</nice_name>
        </license>
      </licenses>
    </project>
</position>
```

(b)

Fig. 1. (a) Project Information (b) Developer Position Information

be found in Fig. 1(a). We collected information of 4261 projects to which a total of 530 developers have contributed.

Position Information: A position is associated with each Ohloh account, which represents the contributions that the account holder has made to the project(s) within Ohloh. Information maintained in a position repository can be found in Fig. 1(b). We collected the position information for each of the 530 contributors.

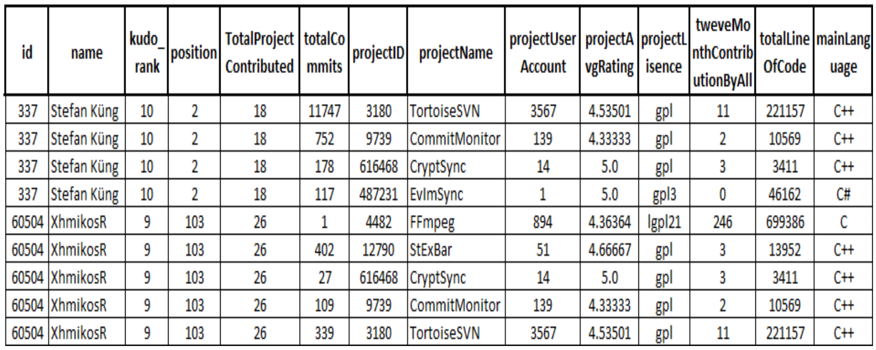
For downloading these repository data, we have implemented Java programs, one for each repository. Each Java program implements the API corresponding to a repository by combining the repository URL's and the unique API key to query the database. The result data set is in XML format.

2.3 Data Processing

From the collected repository data (i.e., the XML files as presented in Section 2.2), we parsed only the information that has significance to this study. The parsed information is recorded under a defined set of attributes/tags in XLSX files, one for each XML repository file.

Collected information is then merged to built a complete database required for data analysis. A partial snapshot of this database can be visualized in Fig. 2. Each row presents detailed information about (a) a contributor, (b) a project in which he/she contributed, and (c) the record of contribution to that project.

To automate data parsing and merging, parsers and data processors were written in Java. These programs use Jsoup HTML parser [9] and Apache POI [8] for parsing the XML files and to create database in XLSX format, respectively.



id	name	kudo_rank	position	TotalProjectContributed	totalCommits	projectID	projectName	projectUserAccount	projectAvgRating	projectLisence	tweveMonthContributionByAll	totalLineOfCode	mainLanguage
337	Stefan Küng	10	2	18	11747	3180	TortoiseSVN	3567	4.53501	gpl	11	221157	C++
337	Stefan Küng	10	2	18	752	9739	CommitMonitor	139	4.33333	gpl	2	10569	C++
337	Stefan Küng	10	2	18	178	616468	CryptSync	14	5.0	gpl	3	3411	C++
337	Stefan Küng	10	2	18	117	487231	EvlmSync	1	5.0	gpl3	0	46162	C#
60504	XhmikosR	9	103	26	1	4482	FFmpeg	894	4.36364	lgpl21	246	699386	C
60504	XhmikosR	9	103	26	402	12790	StExBar	51	4.66667	gpl	3	13952	C++
60504	XhmikosR	9	103	26	27	616468	CryptSync	14	5.0	gpl	3	3411	C++
60504	XhmikosR	9	103	26	109	9739	CommitMonitor	139	4.33333	gpl	2	10569	C++
60504	XhmikosR	9	103	26	339	3180	TortoiseSVN	3567	4.53501	gpl	11	221157	C++

Fig. 2. Partial Snapshot of the database

2.4 Data Analysis

Data analysis targeting to answer the research questions composed of two steps:

First, we created an **Implicit Network** in which two projects have a relationship if both are contributed to by the same contributor. An edge weight in this network represents the number of such common contributors between two projects. As an illustration, consider contributors *Stefan Küng* and *XhmikosR* in Fig. 2. The former contributor has contributed to 4 FLOSS projects as listed under the *projectName* column, while the latter has contributed to 5 projects. Both developers contributed to projects *TortoiseSVN*, *CryptSync*, and *CommitMonitor*. In total, Fig. 2 lists 6 distinct projects. An implicit network among these 6 projects is shown in Fig. 3. Projects *TortoiseSVN*, *CryptSync*, and *CommitMonitor* are linked with edge of weight 2 (i.e. 2 shared developers). All other edges have weight 1 as those project pairs have only one shared contributor. For example, *FFmpeg* and *CommitMonitor* have only developer *XhmikosR* in common.

The complete network is composed of 194424 edges between 4261 projects, with edge weight varies between 1 and 41. Complete edge list of this network can be found in [10].

Second, we measured the extent to which this implicit network comply with the factors often used to classify projects. For this study we selected five factors that are often cited by popular forges (e.g., SourceForge [11]) for categorizing FLOSS projects. These factors include programming language, project size, license, project rating [4] and project domain. Project size was further categorized

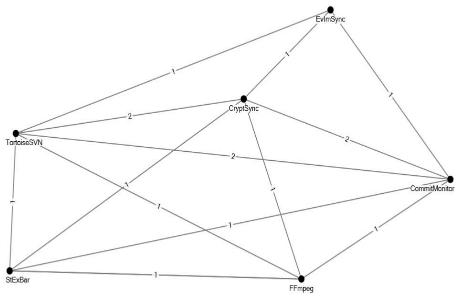


Fig. 3. An illustration of the implicit network

into very large (500K SLOC), large (50K-500K SLOC), medium (5K-50K SLOC) and small (<5K SLOC), according to current literature [19]. Similarly, project rating was classified into top ($\leq$ 4), high ($\leq$ 3 and <4), medium ($\leq$ 2 and <3) and low (<2) on a scale of 5.

For each factor, we identified from the implicit network the number of edges in which both projects have the same value. Due to the large size of the implicit network, we limited this investigation to top ranked 262 edges (the edges that have weight greater than 10) and least ranked edges (random selection of 500 edges from the edges that have weight of 1).

The result of this analysis is reported in Table 2. As an illustration of this approach, consider the project factor *Language* in Table 2. Among the top 262 edges, projects in 228 edges (87.03%) use same programming language, whereas among the bottom 495 edges, projects in 233 edges (47%) have same languages.

2.5 Program Verification

Two pass evaluations were conducted to verify the correctness of the implemented programs. First, the programs were tested with limited number of data samples taken from the collected data. Notified bugs (e.g., errors in parsed data for an HTML tag) were fixed accordingly. Second, a manual checking on a random sample of the actual collected data was done. The correctness of collected data in the second pass was reported to be over 98%.

3 Result Analysis

In this section we investigate the research questions primarily based on the implicit network (described in Section 2.4) revealing projects relationships based on common contributor(s), and by evaluating its compliance with the project factors (presented in Table 2) often used to classify FLOSS projects.

1. How do FLOSS project development communities overlap?

In this study we examined the implicit relationships among 2641 FLOSS projects that are contributed to by 530 contributors. Based on common contributor(s) as a relationship criterion, the implicit network reveals 194424 edges (implicit relations) between 4261 projects. Edge weight lays between 1 and 41, which simply reflects the total number of common contributors between projects. A partial snapshot of this network is shown in Figure 4, in which, for instance, projects *Debian* and *x.Org* have a relationship edge with weight 17. This is because 17 contributors contributed to both projects. This result, all together, portrays the collaborative nature of contribution by FLOSS community members.

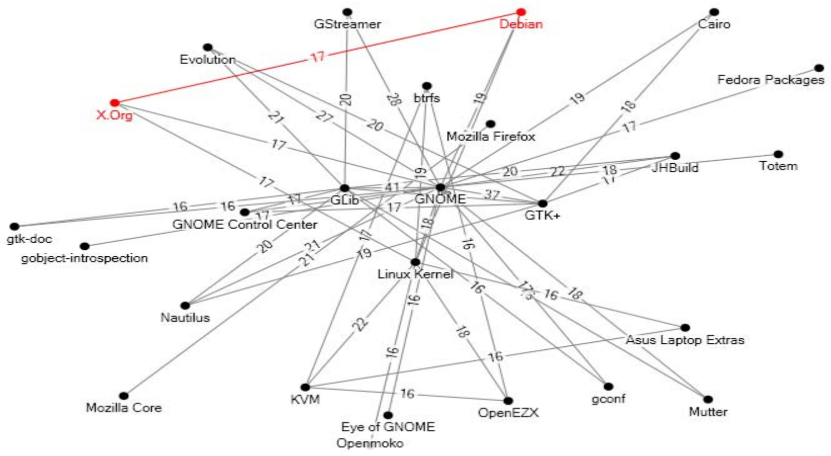


Fig. 4. Partial snapshot of the Implicit Network

To dig further and reason about such large deviation of edge weight, we counted the edges within a certain weight range, result of which is shown in Table 1. As presented in the table, the majority of the edges has low edge weight count (192559 edges out of 194424 have weight bellow 5). Investigating the cause, we observed that projects in these relationships are either medium or small size projects. Even in case where both projects have similar project sizes (3rd row of Table 2), 30% of the projects are medium or small sized. Hence, it is reasonable that communities of such projects should be small. Contrary to this, projects that are within the high edge weight count (e.g., edge weight over 10), are among the very large or large project groups, thus justifying the large overlapping community of contributors.

The above observation is analogous to the *richer gets rich* phenomenon in FLOSS projects collaboration [12], which states that communities that already had a high population would effectively attract more contributors.

Additionally, this structure of sharing contributors among multiple projects is supported by the small-world phenomenon [13]. In a small-world structure several projects are connected with each other through one or more links, e.g., common contributors. In this setup, with increasing number of common contributors, the communities of related projects (as realized by the implicit network) become strongly interconnected. We argue that this in turn may affect project success: The productivity of the contributors is boosted by providing them a dense communication channel to acquire more quantity and variety of information and knowledge resources [14].

Table 1. Edge count under edge weight category

Edge Weight Category	**Edge Count**
Less than 5	192559
Between 5 and 10	1603
Between 10 and 20	252
Greater than 20	10

Furthermore, contributors often participated in projects that belong to the same domain or are sub-projects to a larger one, and that utilizes same programming language(s) as development medium. High percentage of commonalities in implicit network under these two categories (as reported in row 6 and 2 of Table 2) vindicated the claim. This complies with the fact that contributors develop relationships on the common ground of interest [15].

Based on the above observation and discussion it can be affirmed that FLOSS communities often prefer to participate in related projects with participation count varies with the size of the projects.

2. To what extent can developer sharing in FLOSS projects approximate resemblance between the projects themselves?

Within the scope of this investigation, we rationalized the projects relationship in implicit network against the actual factors that relate FLOSS projects. In doing so, we measured the extent to which the implicit network comply with the factors often used to classify projects. This approach is explained in detail in Section 2.4, and the result of which is presented in Table 2.

Among the five project factors, implicit network could effectively approximate three of them, namely, project domain, programming language and project size. Column six in Table 2 shows high percentage of compliance of the implicit relationships to these project factors.

Contributors are most often attracted towards projects that fall within the same project domain or are the sub-projects of a larger one. As can be seen in row six of Table 2, among the top listed 262 edges, 257 (98%) has conformance to same project domain. Similar observation holds (with 70% of conformance) for bottom 500 edges as well. This implies that similar project domain most effectively creates favorable ground for attracting contributors to participate in them.

Table 2. Compliance of the edges in Implicit Network to that of project factors

Project Factor	**Edge Selection Category**	**Selected edges within the category**	**No of Edges in which both nodes have attribute value**	**Edges having same attribute value for the nodes**	**(%) count**	**Additional Info**
Language	Top [Edge weight >10]	262	262	228	87.03%	
	Bottom [Edge weight = 1]	500	495	233	47%	
Project Size	Top [Edge weight >10]	262	262	168	64.13%	Very large: 139 Large: 29
	Bottom [Edge weight = 1]	500	500	150	30%	Very large: 25 Large: 70 medium: 35 Small: 20
License	Top [Edge weight >10]	262	214	84	39.26%	
	Bottom [Edge weight = 1]	500	290	96	33.1%	
Project Rating	Top [Edge weight >10]	262	182	124	68.14%	Top: 107 High: 17
	Bottom [Edge weight = 1]	500	145	130	89.66%	Top: 120, High: 10
Project Domain	Top [Edge weight >10]	262	262	257	98%	
	Bottom [Edge weight = 1]	500	500	350	70%	

Language similarity is found to be one of the major selection factors for contributors participation. According to the data in the second row of Table 2, 87.03% of the top ranked 262 edges have language similarity in contrast to only 47% similarity for the bottom 500 edges. This observation approves that language similarity among projects offers strong support to attract large number of contributors.

The factor project size also imitate analogous results to that of language factor (row three in Table 2). Projects that are very large or large in size (64.13%) are able to manage larger collaborative contributor community than medium or smaller sized projects (30%).

Additionally, results on project rating show that contributors are more attracted towards highly rated projects than low rated ones (row 5 of Table 2).

However, contributors participation to the projects is not constrained by the licenses of the respective projects. Our investigation reported that very low percentage of projects in implicit network have same license terms (only 39.26% of the top ranked edges and 33.1% of the bottom edges as presented in row 4 of Table 2).

Projects that are linked in the implicit network with high edge weight count most likely belong to the same domain, use the same programming languages, or have similar project size.

The following aspects have been identified which could lead to threats to validity of this study.

External validity (how results can be generalized): This study includes 4261 FLOSS projects that are contributed by 530 contributors. Though, these projects cover a wide spectrum of FLOSS territory according to project size, domain, used languages and licenses, we cannot claim completeness of this justification.

Internal validity (confounding factors can influence the findings): The data used in this study is limited to the one provided by Ohloh and may raise trust concerns.

Construct validity (relationship between theory and observation): Data analysis programs written for this study produce data accuracy of over 98%, which was measured with random sample of collected data. This may affect the construct validity.

4 Conclusions

This paper studied to what extent resembling FLOSS components can be tracked based on community activities. Based on our findings, we claim that the proposed approach could approximate to a satisfactory degree resemblance between projects with respect to project domain, programming language and project size. Contrary to these factors, license terms of the projects came out as the least influential factor among all. This finding points out the fact that individual contributors may not be concerned about the licensing issues while selecting projects for contribution. These claims however, need further study. In this regard, a questionnaire to the open source community could be planned and carried out.

Acknowledgement. This work is funded by the Nokia Foundation Grant, 2013 and the TiSE Graduate school, Finland.

References

1. Crowston, K., Howison, J.: The social structure of Free and Open Source Software development. First Monday 10(2) (2005)
2. Aaltonen, T., Jokinen, J.: Influence in the Linux Kernel Community. In: Feller, J., Scacchi, B.F.W., Sillitti, A. (eds.) Open Source Development, Adoption and Innovation. IFIP, vol. 234, pp. 203–208. Springer, Boston (2007)
3. Bonaccorsi, A., Rossi, C.: Altruistic individuals, selfish firms? the structure of motivation in open source software. First Monday (1-5) (2004)
4. Ohloh, `http://www.ohloh.net/` (last accessed November 2013)
5. Ohloh kudo rank, `http://meta.ohloh.net/kudos/` (last accessed November 2013)
6. Deshpande, A., Riehle, D.: The Total Growth of Open Source. In: Russo, B., Damiani, E., Hissam, S., Lundell, B., Succi, G. (eds.) Open Source Development, Communities and Quality. IFIP, vol. 275, pp. 197–209. Springer, Boston (2008)

7. Rudzki, J., Kiviluoma, K., Poikonen, T., Hammouda, I.: Evaluating Quality of Open Source Components for Reuse-Intensive Commercial Solutions. In: Proceedings of EUROMICRO-SEAA 2009, pp. 11–19 (2009)
8. Apache POI-Java API for Microsoft Documents, `http://poi.apache.org/` (last accessed September 2013)
9. jsoup: Java HTML Parser, `http://jsoup.org/`
10. Research Data, `http://datasourceresearch.weebly.com/`
11. Source Forge, `http://sourceforge.net` (last accessed November 2013)
12. Weiss, M., Moroiu, G., Zhao, P.: Evolution of Open Source Communities. In: Damiani, E., Fitzgerald, B., Scacchi, W., Scotto, M., Succi, G. (eds.) Open Source Systems. IFIP, vol. 203, pp. 21–32. Springer, Boston (2006)
13. Vir Singh, P.: The Small-World Effect: The Influence of Macro-Level Properties of Developer Collaboration Networks on Open-Source Project Success. ACM TOSEM 20(2), Article 6 (2010)
14. Watta, D.: Networks, dynamics, and the small world phenomenon. Amer. J. Sociology 105, 493–527 (1999)
15. Madey, G., Freeh, V., Tynan, R.: The open source software development phenomenon: An analysis based on social network theory. In: Americas Conf. on Information Systems, pp. 1806–1813 (2002)
16. Robles, G., González-Barahona, J.M.: A Comprehensive Study of Software Forks: Dates, Reasons and Outcomes. In: Hammouda, I., Lundell, B., Mikkonen, T., Scacchi, W. (eds.) OSS 2012. IFIP AICT, vol. 378, pp. 1–14. Springer, Heidelberg (2012)
17. Orsila, H., Geldenhuys, J., Ruokonen, A., Hammouda, I.: Update Propagation Practices in Highly Reusable Open Source Components. In: Russo, B., Damiani, E., Hissam, S., Lundell, B., Succi, G. (eds.) Open Source Systems. IFIP, vol. 275, pp. 159–170. Springer, Boston (2008)
18. Gonzalez-Barahona, J.M., Robles, G., Dueñas, S.: Collecting Data About FLOSS Development: The FLOSSMetrics Experience. In: Proceedings of FLOSS 2010, Cape Town, South Africa, pp. 29–34 (2010)
19. Scacchi, W.: Understanding Open Source Software Evolution: Applying, Breaking, and Rethinking the Laws of Software Evolution. John Wiley and Sons (2003)

[VII] M.M. Syeed, I. Hammouda, and C. Berko. Exploring Socio–Technical Dependencies in Open Source Software Projects — Towards an Automated Data-driven Approach. In *Proceedings of 17th the Academic MindTrek Conference*, pages 273–280. ACM, September, 2013.

Exploring Socio-Technical Dependencies in Open Source Software Projects

Towards an Automated Data-driven Approach

M.M. Mahbubul Syeed
Tampere University of Technology
mm.syeed@tut.fi

Imed Hammouda
Tampere University of Technology
imed.hammouda@tut.fi

Csaba Berko
Tampere University of Technology
csberko@yahoo.com

ABSTRACT

Comprehension of Open Source Software (OSS) projects is traditionally driven by the plethora of data produced and maintained by these projects. The data, in one hand, encapsulates the tacit knowledge on the evolution of the software itself. And, on the other hand, provides the history of communication and collaboration of the community. Acquisition and analysis of such data has been mostly manual or semi-automated and error-prone, mainly due to unstructured and substandard data representation. This increases the validity threat of the reported results and makes it incomparable across the studies. With the advancement of data management tools and technologies, many third party data providers are putting serious effort to provide OSS project's data in a standard and platform independent format. In this paper, we propose a framework to fully automate the analysis and visualization of OSS evolution data through the use of existing data services. As a proof of concept we implemented a tool named POMAZ. We demonstrate the applicability of the tool in the context of two related open source projects FFmpeg and GStreamer.

Keywords

Open Source Software, Data Analysis, Socio-Technical Congruence

1. INTRODUCTION

For the past couple of decades, the research community has been studying the open source movement from a range of perspectives including economy [12], law [39], technology [33], sociology [30], and information systems [20]. These dimensions are not independent of each other, and decisions regarding any one may be influenced by the decisions taken for the other dimensions. For example dual-licensing [26] is often regarded as both a legality and business model. Another example is the evolution pattern of Open Source Software (OSS) projects, which is often regarded as a socio-technical issue [18]: The way a project community operates may affect the way the project source code is maintained, and vice versa. There are plenty of similar hidden patterns and dependencies that represent both opportunities for action and for reflection. Indeed, mining and exploring such implicit and hidden patterns is an important step supporting the comprehension and the evaluation of OSS projects.

For examining the various patterns and dependencies in OSS projects, researchers often resort to public project data. Projects are typically associated with rich repositories containing various sources of information such as source code, bug reports, mailing lists, and revision history logs. However, working with such data is both an opportunity and challenge. On the one hand, data offer a nice tool for a rich and more objective decision making process. On the other hand working with heterogeneous and big data might be a challenging task related to acquisition, exploration, and analysis activities [19]. For example, acquisition of such data has been mostly manual or semi-automated and error-prone, mainly due to unstructured and substandard data representation.

In this paper, we argue that such challenges can be partly circumvented through the use of third party data collection services of OSS projects. These data are openly available to analyze and can be used with simple API's. This advancement opens the door to fully automate the data analysis and visualization process, and overcoming the deficiencies of earlier works. Researchers would then focus on implementing techniques and tools using ready available and structured data. Within the wide spectrum of open source dimensions, the main focus of this paper is to examine the socio-technical perspective as an important explanatory factor for many interesting comprehension questions. We also take this perspective merely from a pragmatic point of view: most OSS project data deal with community and source code related information.

The contribution of the paper is threefold: First, we review the social and technical dimensions of open source software. Second, we introduce a tool concept for automated data-driven exploration approach of OSS projects. Third, we present a generic tool named POMAZ implementing the proposed approach. The tool uses existing and new methods for measuring evolving project attributes and for querying statistical information from project repository. Using the tool, it is possible to monitor the evolution of OSS projects from different viewpoints and set forecasts up regarding their future development. We discuss the applicability of the tool in the case of two competitive projects FFmpeg [7] and GStreamer [11].

The rest of the paper is structured as follows. In Section 2, we review the socio-technical perspectives of open source. In Section 3, we set the stage for our proposed approach and tool concept. A concrete tool environment implementing the approach is then presented in Section 4. In Section 5, we demonstrate the feasibility of the approach by discussing a number of scenarios applied to two OSS projects. Finally, in Section 6, we draw some final conclusions and point out directions for future work.

2. BACKGROUND

In this section, we present a holistic view on the research of OSS Projects through identification and definition of the targeted perspectives of OSS projects that were explored in current literature. Subsequently our motivation behind the work is portrayed.

Research towards OSS projects can be classified along several distinct facets, namely, Social [40][30], Technical [31][4], Socio-Technical [36][3], Legal [39], and Economical [12]. In this paper, however we focus on the first three facets, i.e., Social, Technical and Socio-Technical. Because these facets have been found central when considering studies pertaining evolution of OSS projects [37].

In what follows we discuss the three facets and current research within. Representative inquiries within these facets are also presented, which are investigated in this study.

2.1 Social

OSS development is a perceptible example of the collaborative community-based model [35]. The communities are at the core of the social dimension of OSS projects which comprises of the community of developers and users [40].

Based on the level of participation, the role of the community members can be classified within the range of project leaders (maintainers) and core members (contributors) to active and passive users [42][43][45]. Project leaders are the ones who usually initiate the project and make the major development decisions. Core members often make significant contributions to a project over time. Depending on task allocation, core members can further be subdivided into creators (leaders), communicators (managers), and collaborators [8]. Active users are the part of the community who often report bugs, but do not fix them. Finally, passive users are all remaining users who just use the system.

Community members contributing to an OSS project are not strictly bound to any organizational rules, regulation and structure. Rather they voluntarily join and contribute to the project. These people belong to discrete geographical locations having significant difference in background, timezone, language and cultural distances [36]. Often simple communication media like email, wiki, chat [44] are used to establish communication, collaboration among the community members. To develop, share and contribute to the software code, revision tracking systems (such as CVS, SVN), change logs, bug tracking systems (like bugzilla) [14], and project hosting cites like, sourceforge, github [46] are used.

Research directing towards social dimension mostly explored the social interaction data stored in the communication archives such as email, chat, wikis [10]. The plethora of data stored in these archives are effectively utilized to build the significant body of knowledge that comprises: (a) the understanding of motivations, participation, and performance of the community members [30], (b) formation of the community structure, and practices, (c) the communication and collaboration patterns [40], and (d) the success factors [10].

Within this facet, the following issue, for example, would be worth exploring: how does open source community change during project evolution? Can we perceive any community evolutionary patterns?

2.2 Technical

The technical domain of an OSS project often constitutes the software code-base, and activity traces concerning its development and maintenance, such as, change log/revision history log, commit records, and bug reports. In maintaining the code-base and associated data over the releases of the software, several data management tools are used for instance CVS/SVN to maintain the code-base, and bugzilla, jira bug reporting systems for fault/feature request management.

Research within the technical domain have contributed heavily in understanding the complexity of the software development through exploration of the sources listed above. Focus of such research can be classified along two dimensions, namely, intra-project analysis and inter-project analysis.

Inter-project studies can be classified further into macro and micro level studies. For instance, macro level studies explores how the different versions of the code-base evolve, e.g., in case of forked projects [32]. Whereas, micro level studies include (but not limited to) the followings, (a) understanding the software growth in size [31] (e.g., lines of code, commits, comments, commit total), complexity (e.g., cyclomatic complexity, halstaed complexity) [33], and modularity; (b) study the quality aspects of the software [4], and (c) predict the evolution [41], maintainability (e.g., refactoring) [24], defect density [16] of the code base.

Inter-project studies are mostly exploring relationships and dependencies among related projects, for instance studying update propagation patterns among different OSS projects [27].

A example inquiry within this domain would be to investigate how the technical documentation within the code base evolve with the growth of the software.

2.3 Socio-Technical

The Socio-technical perspective of an OSS project bonds the social and the technical dimensions, e.g., the people, artifacts, computational environments, and enables to develop a method, or taxonomy, to analyze, and understand OSS development [23]. A high degree of correspondence between the two dimensions, termed as socio-technical dependency in literature, is a natural consequence and desired property for a collaborative development environment, such as OSS projects [2].

Understanding socio-technical dependency through the analysis of congruence between the social and technical dimension is vital for supporting collaboration and coordination in OSS projects [36][3]. For instance, developers who are modifying interdependent code modules but not communicating may introduce potential future integration problems.

Though the topic is relatively new for the research community, yet there exists encouraging results in current literature. To be precise, higher level of socio-technical congruence increases the efficiency of software development process [3]. Also the success of software build process and the quality is dependent on the congruence level [18][22]. Measurement of congruence using data mined from OSS repositories reported high and consistent congruence between the two domains during the evolution of such projects [36]. And it is suggested that for OSS projects, socio-technical congruence can be measured at different points in time in order to optimally guide software development activity [6].

A classical scenario to investigate within this facet would be to study the effect of the community on the software, for example studying the correlation between the growth of the community and the software in terms of size.

3. TOWARDS AN AUTOMATED DATA DRIVEN ANALYSIS

Studies falling within the three facets (discussed in Section 2) are predominantly influenced and driven by high-confidence data on the structure of OSS [41]. These data present a sheer volume of information comprising both the development (i.e., technical) and communication (i.e., social) history of the project [1]. This alleviates the study of Socio-Technical dependencies through mining and examining the tacit knowledge buried in these artifacts.

This section provides a holistic view on the benefactor of such artifacts, associated challenges in exploiting the sources, and a methodological classification that are effectively used for such exploration.

3.1 Data Repositories in OSS Projects

Traditionally, each OSS project maintains its data through different data management tools and made them publicly available through public interfaces, e.g., Internet. Additionally, many of these projects are nowadays moving to third party hosting facilities which provide efficient data management and communication support.

Accumulation of the tacit knowledge from these data sources requires several activities, such as, data acquisition, cleaning, analysis and interpretation (e.g., visualization). A key concern of this process is that most of the activities requires manual effort, specially data acquisition and cleaning. Because, data offered by the above sources are heterogeneous in nature, as different projects use different data management tools, have different data representation, and that the data might be scattered in different archives. Thus it is a challenging task to ensure the quality of the collected data, even following standard data collection process.

3.2 OSS Project Analysis Approaches

Different analysis approaches and methods have been using the public data differently. We argue that such approaches can be categorized into three generations, as presented in [9]. These generations are distinct from each other according to the degree of automation they offer in exploring and analyzing the data sources.

For the first generation approaches, the task of data gathering and evaluation are mostly manual. For instance, first generation quality models for OSS (e.g., OpenBRR, QSOS) falls within this category [9]. The second generation approaches offer semi-automated process. A representative example would be the second generation quality models (e.g., QualOSS) [9].

Following the trend towards the automation of OSS data analysis approaches, it can be argued that eventually the third generation approaches would endeavor fully automated methods and techniques [9]. To elevate this process of automation, we present in this paper, a methodological framework to articulate the analysis of the collaborative design of the OSS projects. We further argue that our framework would portray unified and comparable analysis of OSS projects by establishing links between the heterogeneous components of a project's hybrid data sources.

3.3 The Framework

In this section we present a generic three-layer framework to fully automate the data driven analysis of the OSS projects. Taking the advantage of highly generic and standard representation of OSS data, provided by the third party data providers, this framework will seamlessly integrate the three layers to serve the purpose of data analysis and visualization. The proposed framework is presented in Figure 1.

The key considerations for deriving this framework includes, data independence, re-usability of the data, and increase of the validity measure and comparability of results produced by the research community. In this regard, data independence ensures the fact that methods derived for OSS data analysis should be independent of OSS data, and can be applied across OSS projects. Re-usability should enforce logging of data to local repositories to minimize costly request over Internet for subsequent access. Additionally, a standard representation and access mechanism of data across OSS projects would produce cohesive research results, with greater validity, confidence, and comparability.

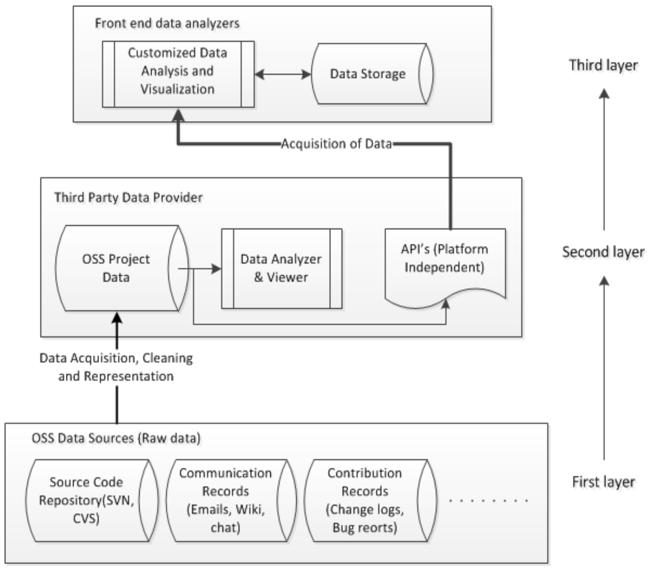

Front end data analyzers
Customized Data Analysis and Visualization
Data Storage
Third layer
Acquisition of Data
Third Party Data Provider
OSS Project Data
Data Analyzer & Viewer
API's (Platform Independent)
Second layer
Data Acquisition, Cleaning and Representation
OSS Data Sources (Raw data)
Source Code Repository(SVN, CVS)
Communication Records (Emails, Wiki, chat)
Contribution Records (Change logs, Bug reorts)
. . . .
First layer

Figure 1: Framework for third generation data analyzers

As shown in Figure 1, the first layer of this framework consists of the raw data sources offered by the hosting sites. For instance, most OSS projects provides public interfaces for each of the data sources to be accessed and downloaded.

None-the-less, hosting cites, such as sourceforge and github provide similar facilities for the projects hosted by them.

The second layer constitutes the third party data providers, a representative example would be OHLOH [25]. We propose three essential components for this layer. First, the data storage. The purpose of this storage is to perform data acquisition, essentially from all available sources provided by the first layer. Additionally, it will perform adequate data cleaning, and provide categorical representation of data with unified and standard format. Second, the data analyzer and viewer module, which would provide high level data analysis and visualization of selected projects. For instance, visualizing project activity for a selected period. Third, a platform independent interface, which would be built to provide access to the data. For instance, a set of API's can be build to query the database, which will return the resultant data set in a platform independent format, such as XML. Thus methods and tools implemented on top of these interfaces would essentially produce unified results that are coherent and comparable to each other.

The third layer in the framework is the front end data analyzers. This layer consists of a set of customized tools and methods, and the local data storage unit. The methods will support the need of different parties, e.g., research works targeting to understand different perspectives of OSS projects. These tools make use of the unified data provided by the second layer through standard set of API's. This standardization of data removes the burden of semi-automated data gathering and preprocessing activities, and facilitate researchers to concentrate more on the soundness of the underlying data analysis methodologies. Also the data storage unit could cache the data collected through the API's and as well as store the processed one for further use.

Within this framework, the second layer implements the data independence for the tools and methods built in third layer, whereas, the data storage in the third layer offers reusability of data.

4. PROOF OF CONCEPT

In exploring the effectiveness of the framework, a conceptual architecture encompassing the proposals of the framework is derived. Corresponding tool support is built and evaluated through multiple case studies.

4.1 The Architecture

The proposed architectural design presented in Figure 2. The data provider in this reference architecture is the OHLOH repository, thus corresponds to the second layer of the framework. An elaborated discussion on OHLOH is presented in Section 4.2. The other modules presented within this architecture (modules within the rectangle box in Figure 2) are the implementation of the third layer in the framework. The Connector module in this layer is responsible for the low-level connection and data collection. This module also handles problems concerning missing data, false data or unavailability of the data provider repository.

Data holder module implements the reusability of the data. This module stores already collected data in local disk, thus minimizing expensive server communication for repeated use of the same data.

The analyzer accepts the user query, identifies and collect the required data from the Data holder, and carries out necessary computing. The resulting information is represented

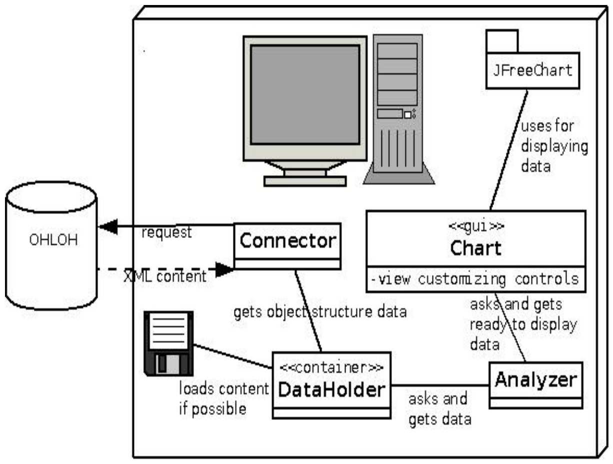


Figure 2: The architecture

in a format in which the chart can show it.

The Chart module handles all GUI operations (such as events) in the application, it communicates with the user what kind of charts do they need, and asks the analyzer for the information. As soon as it arrived, Chart displays the information using the open source JFreeChart library [13].

4.2 OHLOH: the Third Party Data Provider

The OHLOH [25] repository was selected as the third party data provider for OSS project data. This constitute the second layer of our proposed framework. OHLOH collects development information about OSS projects and makes them available on their homepage. OHLOH maintains data collected from 400 thousand projects. These data can be accessed through a set of open API which handles URL requests and responses. The response data is expressed as an XML file, an example of which is shown in Figure 3.

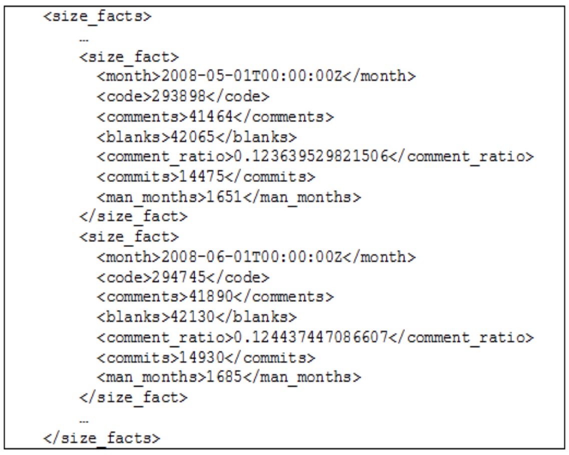

```
<size_facts>
    …
    <size_fact>
      <month>2008-05-01T00:00:00Z</month>
      <code>293898</code>
      <comments>41464</comments>
      <blanks>42065</blanks>
      <comment_ratio>0.123639529821506</comment_ratio>
      <commits>14475</commits>
      <man_months>1651</man_months>
    </size_fact>
    <size_fact>
      <month>2008-06-01T00:00:00Z</month>
      <code>294745</code>
      <comments>41890</comments>
      <blanks>42130</blanks>
      <comment_ratio>0.124437447086607</comment_ratio>
      <commits>14930</commits>
      <man_months>1685</man_months>
    </size_fact>
    …
</size_facts>
```

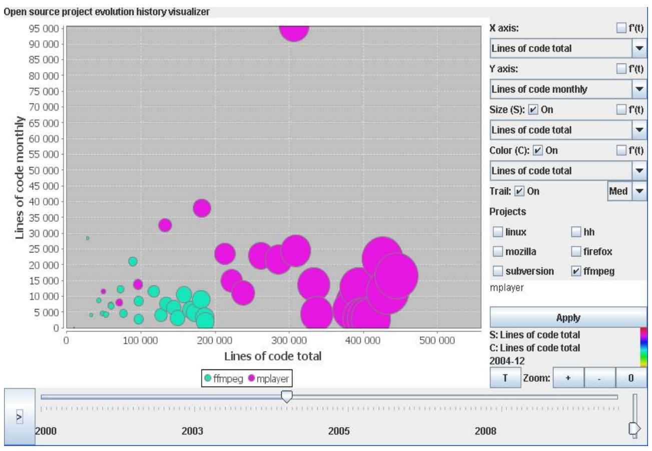
Figure 3: An example of XML content

The use of OHLOH repository makes OSS data available in a clean, unified and standard format. This makes the process of data analysis and visualization independent of technology and data repository, which in turn took off the burden of data acquisition, collection, and cleaning from heterogeneous sources of OSS projects.

For the purpose of this study the following metrics, representative of the three dimensions are selected: Lines of code, lines of comments, commits (monthly, yearly and total), and contributors (active contributors and total contributions). To aid fine grained research with the data, having the derivate of the different data sequences is also possible; it makes determining inflexion points easier for example.

4.3 Pomaz: the Tool Demonstration

POMAZ is a prototype tool implemented on top of the architecture presented in Figure 2. Figure 4 depicts the user interface of the tool applied to two OSS projects FFmpeg and MPlayer. Data of each project is shown using bubbles of certain color information. The tool is capable to represent four evolution attributes in addition to timeline information shown in the lower bar of the figure. The timeline bar has a manual time adjustment slider and an animation speed adjuster slider to control time information. The four project attributes are in turn shown by the X axis, the Y axis, the size of the bubble, and color of the bubble, as shown in the right top part of the figure. Using POMAZ, it is also possible to visualize monthly changes in the attributes by taking the derivative of the data. This can be done by ticking the checkboxes shown in front of the four dimensions. It is possible to visualize as many projects as desired. The last visited projects are maintained in the tool GUI. Zooming functionality on visualized data is also available.



X axis: Lines of code total ☐ f'(t)
Y axis: Lines of code monthly ☐ f'(t)
Size (S): ✓ On ☐ f'(t) Lines of code total
Color (C): ✓ On ☐ f'(t) Lines of code total
Trail: ✓ On Med
Projects: ☐ linux ☐ hh ☐ mozilla ☐ firefox ☐ subversion ✓ ffmpeg mplayer
S: Lines of code total
C: Lines of code total
2004-12

Figure 4: The user interface of POMAZ

5. CASE STUDY

As stated earlier, FFmpeg [7] and GStreamer [11] have been selected as case study OSS projects. The two projects can be considered as competing, though the two communities do cooperate with each others. FFmpeg is more widely used and popular. GStreamer provides compatibility with FFmpeg through an own developed plug-in.

5.1 Conceptualization of OSS Dynamics

In this section we illustrate the applicability of the tool by analyzing the example scenarios presented in Section 2 within the contest of the selected case study projects.

Q1. How does the community affect the software?

Investigation of this question is two fold. First, the growth pattern of the community (in size) to the that of the growth of software (in terms of both total lines of code and total commits) is measured. Second, the community activity is studied with respect to volume of contribution and contribution quality. The acquired knowledge of this investigation is as follows,

Software growth vs community growth. Using POMAZ, we plotted the total Lines of code (in X-axis), total Lines of commit (in Y-axis), active contributors (in Bubble size) for the two case study projects. The analysis result is shown in Figure. 5.

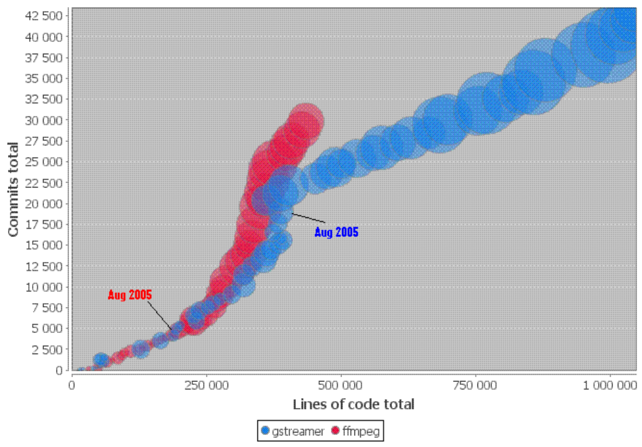


Figure 5: Growth pattern of the community in relation to the software.

According to the results (in Figure 5), in FFmpeg, a sharp increase in number of active contributors was noticed along with higher number of commits. But relatively less lines of code were contributed. The scenario is however different in the case of GStreamer. The increase in number of active contributors is associated with an increase in both commits and lines of code.

Community activity. In POMAZ we plotted the derivative of the commits data (in Y-axis), the derivative of the lines of code data (in X-axis) and active contributors (in Bubble size), the result of which is illustrated in Figure. 6.

It is observed that in most cases (in Figure. 6), when GStreamer has high contributor activity (bubble is big), it either has more commits or makes a significant change in code size (either in plus or minus). This observation contradicts with FFmpeg. In this case, the big bubbles are mostly centered around the the vertical baseline, which indicates that the high contributor activity does not necessarily result neither in change of line count, nor in commits count. It is also noticed that especially in many cases a positive growth in commits is associated with a negative growth in lines of code. This indicates that a significant amount of restructuring and refactoring might be carried out during this period of the projects.

Additionally, both projects maintain a linear growth in number of line of codes and number of comments. This can be noticed from Figure. 7 in which we plotted total Lines of

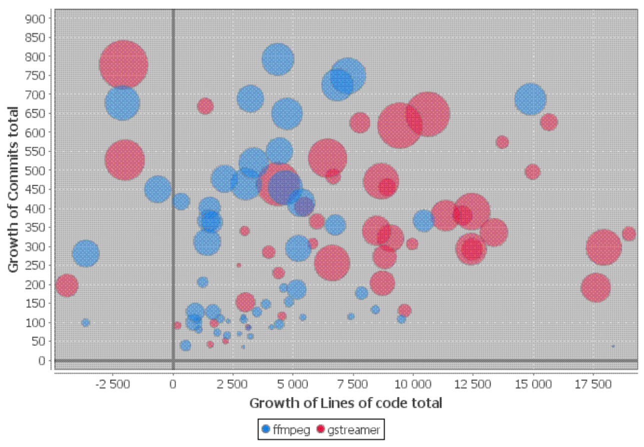


Figure 6: Community activities

code (in X-axis), total Lines of comments (in Y-axis), active contributors (Bubble size).

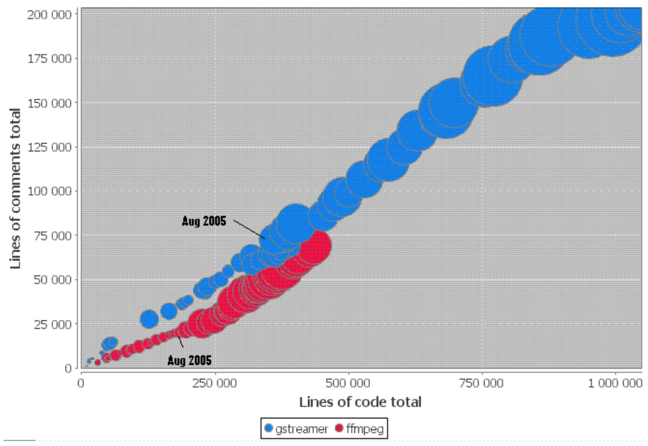


Figure 7: Community activity in relation to the software.

The liner growth in both code and comment might be an indication of quality software development process. Case-wise analysis showed that in FFmpeg, when the comment ratio jumped, the contributor activity jumped as well. After that, comment ratio and contributor activity did not change a lot. GStreamer also followed the similar trend with a slight exception in later phase, where the comment ratio dropped a bit.

Q2. How does community changes with the project evolution?

To conceive the community change over the evolution of the project, we plotted the total line of codes (in X-axis), the growth of active contributors (in Y-axis). The bubble size in the chart denotes the most recent data. The chart is shown in Figure. 8. From the figure it is hard to derive any pattern of change in community size. Yet, the change in community size follows the traditional notion of freedom in the OSS project. That is, people in OSS projects are free to join or leave at any given time. This can be observed from the figure that for both the projects people are jointing or leaving in adhoc basis, without following any particular pattern, starting from the early stage of the projects. This asynchronous change in community size can effectively be justified with the degree of freedom the community members enjoy in participating to an OSS project.

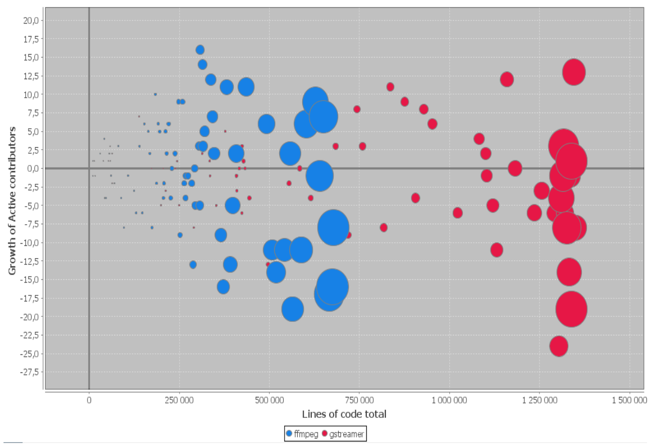


Figure 8: Community Growth pattern

Q3. How does documentation follow the growth of software?

In observing how the documentation (e.g., commenting) of the source code evolves with the code contribution, we plotted the derivative of the line of codes (in X-axis) and the derivative of the line of comments (in Y-axis). The result reveals interesting observations (Figure. 9). First, in both projects, the growth of comment follows the growth of code. In case of GStreamer, this growth pattern is more linear than FFmpeg. For FFmpeg, in many occasion, there are more comments committed than the code. Second, in FFmpeg, comment grows with the negative growth of code base. Third, for GStreamer, there are occasions when both code and comment have declined. Accumulation of these results shows a healthy coding practice within both the projects, in which comments on the code only decreases along with the decrease of the code size.

6. DISCUSSION

In this paper we presented a methodological framework to perform fully automated analysis and visualization of OSS projects. We also offered an architectural design built on top of the framework, and correspondingly provide tool support to demonstrate the applicability.

The software community has proposed a wide range of analysis tools for open source projects. Examples of such tools are listed in Table. 1. As observed, the example tools address different dimensions of open source, such as, technical, social and licensing. Furthermore, the tools use raw data extracted from the publicly available repositories. This is in contrast to our approach which uses data from third party data providers. Both approaches to data access come with own advantages and limitations.

In this regard, our proposed framework would help achieving unified research results taking advantage of already avail-

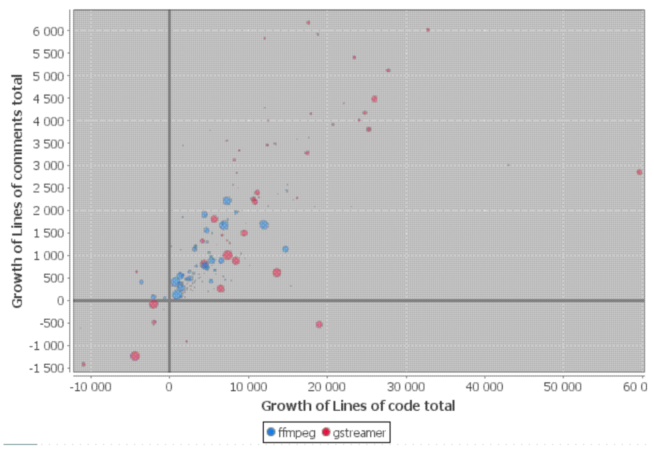


Figure 9: Documentation and code size

Table 1: A two dimensional classification of OSS analysis tools

-	Software	Communication	Configuration
Code analysis	[17] [34]		[47] [34]
SNA	[21] [34]	[15][5] [34]	[29]
Licensing	[28][38]		

able data collected and presented in a standard and platform independent format. Additionally, the time consuming and erroneous process of data acquisition process are eliminated, stipulating higher validity on the collected data. This approach is also in line with recent approaches of big data analysis, such as social network services.

On the contrary, the limiting factor of this approach includes the trust issue on the data provided by the third party data provider, and at the same time studies are limited to the data provided by the data provider. For example, geographical developer distribution and some other information are impossible due to lack of data in current OHLOH's system.

The proposed approach in this paper represents only a small step towards a full fledged solution framework for automated analysis of open source projects. We argue that, similar to the rise of data services such as OHLOH, future analysis tools would be developed as plug-ins loaded online in third party service containers. This is analogous to social network sites like Facebook, which acts as an open container for custom applications in addition to providing API for hosted data.

7. ACKNOWLEDGMENTS

This work was partially funded by TiSE and Ulla Tuomisen säätiö, Finland.

8. REFERENCES

[1] A. Al-Ajlan. The evolution of open source software using eclipse metrics. In *NISS*, pages 211–218, 2009.

[2] T. Browning. Applying the design structure matrix to system decomposition and integration problems: a review and new directions. *IEEE Transactions on Engineering Management*, 48(3):292–306, 2011.

[3] M. Cataldo, P. Wagstrom, and J. D. Herbsleb. Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In *ACM CSCW*, pages 353–362, 2006.

[4] D. Challet and Y. L. Du. Microscopic model of software bug dynamics: Closed source versus open source. *IJRQSE*, 12(6), 2005.

[5] K. Crowston and J. Howison. The social structure of free and open source software development. In *First Monday*, 2005.

[6] K. Ehrlich, G. Valetto, and M. Helander. Using software repositories to investigate socio-technical congruence in development projects. In *ICGSE*, 2007.

[7] FFmpeg. http://ffmpeg.org/. *accessed March 2013.*

[8] P. Gloor. *Swarm Creatitity.* Oxford Uni. Press, 2006.

[9] R. Glott, A.-K. Groven, K. Haaland, and A. Tannenberg. Quality models for free/libre open source software-towards the silver bullet? In *EUROMICRO*, pages 439–446, 2010.

[10] M. Goeminne and T. Mens. A framework for analyzing and visualizing open source software ecosystems. In *IWPSE-EVOL*, pages 42–47, 2010.

[11] GStreamer. http://gstreamer.freedesktop.org/. *Last accessed March 2013.*

[12] F. Hecker. Setting up shop: The business of open-source software. *IEEE Software*, 16(1):45–51, 2009.

[13] JFreeChart. http://www.jfree.org/jfreechart/. *accessed March 2013.*

[14] M. B. K. Fogel. Open source development with cvs: Learn how to work with open source software. In *The Coriolis Group*, 1999.

[15] Y. Kamei, S. Matsumoto, H. Maeshima, Y. Onishi, M. Ohira, and K. Matsumoto. Analysis of coordination between developers and users in the apache community. In *OSS*, pages 81–92, 2008.

[16] C. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *TSE*, 25(4):493–509, 1999.

[17] P. Knab, M. Pinzger, and A. Bernstein. Predicting defect densities in source code files with decision tree learners. In *MSR*, pages 119–125, 2006.

[18] I. Kwan, A. Schrter, and D. Damian. Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. *TSE*, 37(3):307Ű324, 2011.

[19] M. Lehman and J. Ramil. Rules and tools for software evolution planning and management. In *Annals of Software Engineering*, volume 11, pages 15–44, 2001.

[20] J. Ljungberg. Open source movements as a model for organising. *European Journal of Information Systems*, 9(4):208–216, Dec. 2000.

[21] J. Martinez-Romo, G. Robles, M. Ortuño-Perez, and J. M. Gonzalez-Barahona. Using social network analysis techniques to study collaboration between a floss community and a company. In *OSS*, pages 171–186, 2008.

[22] N. Nagappan, B. Murphy, and V. R. Basili. The influence of organizational structure on software quality: an empirical case study. In *ICSE*, page 521Ű530, 2008.

[23] K. Nakakoji, K. Yamada, and E. Giaccardi. Understanding the nature of collaboration in open-source software development. In *APSEC*, pages 827–834, 2005.

[24] E. Nasseri and S. Counsell. System evolution at the attribute level: An empirical study of three java oss and their refactorings. In *ITI*, pages 653–658, 2009.

[25] OHLOH. https://www.ohloh.net/. *Last accessed March 2013.*

[26] M. Olson. ual licensing. *In Open Sources 2.0, OŠReilly*, 2005.

[27] H. Orsila, J. Geldenhuys, A. Ruokonen, and I. Hammouda. Update propagation practices in highly reusable open source components. In *OSS*, pages 159–170, 2008.

[28] M. D. Penta and D. German. Who are source code contributors and how do they change? In *WCRE*, pages 11–20, 2009.

[29] G. Porruvecchio, S. Uras, and R. Quaresima. Social network analysis of communication in open source projects. In *Int. Conf. on Agile Processes in Software Engineering and Extreme Programming*, pages 220–221, June 2008.

[30] J. Roberts, I.-H. Hann, and S. Slaughter. Understanding the motivations, participation, and performance of open source software developers: A longitudinal study of the apache projects. *Management Science*, 52:984–999, 2006.

[31] G. Robles, J. Amor, J. Gonzalez-Barahona, and I. Herraiz. Evolution and growth in large libre software projects. In *Principles of Software Evolution*, pages 165 – 174, 2005.

[32] G. Robles and J. Gonzalez-Barahona. A comprehensive study of software forks: Dates, reasons and outcomes. In *OSS*, pages 1–14, 2012.

[33] R. Sangwan, P.Vercellone-Smith, and C. Neill. Use of a multidimensional approach to study the evolution of software complexity. *Journal of Innovations in Systems and Software Engineering*, 6(4):299–310, 2010.

[34] A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb. Tesseract: Interactive visual exploration of socio-technical relationships in software development. In *ICSE*, pages 23–33, May 2009.

[35] S. Shah. Motivation, governance, and the viability of hybrid forms in open source software development. *Journal of Management Science*, 52(7):1000–1014, July 2006.

[36] M. M. Syeed and I. Hammouda. Socio-technical congruence in oss projects: Exploring conways law in freebsd oss evolution. In *9th OSS*, pages 109–126. Springer, June 2013.

[37] M. M. Syeed, I. Hammouda, and T. Systa. The evolution of open source software projects: a systematic literature review. *Journal of Software*, May 2013.

[38] T. Tuunanen, J. Koskinen, and T. Karkkainen. Asla: reverse engineering approach for software license information retrieval. In *CSMR*, pages 291–294, Mar. 2006.

[39] M. Välimäki. *The Rise of Open Source Licensing. A Challenge to the Use of Intellectual Property in the Software Industry.* Turre Publishing, 2005.

[40] M. Weiss, G. Moroiu, and P. Zhao. Evolution of open source communities. In *IFIP*, pages 21–32, 2006.

[41] H. Wen, R. DŠSouza, Z. Saul, and V. Filkov. Evolution of apache open source software. *Modeling and Simulation in Science, Engineering and Technology, Springer*, pages 199–215, 2009.

[42] R. Wendel, J. Bruijn, and M. Eeten. Protecting the virtual commons, information technology & law series. In *T.M.C. Asser Press*, pages 44–50, 2003.

[43] J. Xu and G. Madey. Evolution of open source communities. In *NAACOSOS*, 2004.

[44] Y. Yamauchi, M. Yokozawa, T. Shinohara, and T. Ishida. Collaboration with lean media: How open-source software succeeds. In *CSCW*, pages 329–338, 2000.

[45] Y. Ye and K. Nakakoji. The co-evolution of systems and communities in free and open source software development. pages 59–82.

[46] M. Zanetti. The co-evolution of socio-technical structures in sustainable software development: Lessons from the open source software communities. In *ICSE*, pages 1587 – 1590, 2012.

[47] Y. ZHOU and J. DAVIS. Open source software reliability model: an empirical approach. In *WOSSE*, volume 30, pages 1–6, July 2005.

[VIII] M.M. Syeed. Binoculars: Comprehending Open Source Projects through graphs. In *Proceedings of 8th IFIP WG 2.13 International Conference of Open Source Systems (OSS'2012)*, pages 350–355. Springer, September, 2012.

Binoculars: Comprehending Open Source Projects through Graphs

M.M. Mahbubul Syeed

Tampere University of Technology, Finland
mm.syeed@tut.fi

Abstract. Comprehending Open Source Software (OSS) projects requires dealing with huge historical information stored in heterogeneous repositories, such as source code versioning systems, bug tracking system, mailing lists, and revision history logs. In this paper, we present Binoculars, a prototype tool which aims to provide a platform for graph based visualization and exploration of OSS projects. We describe the issues need to be addressed for the design and implementation of a graph based tool and distill lessons learned for future guideline.

1 Introduction

Open Source Software (OSS) has gained interest in both commercial and academic world over the past decade due to its high quality. Successful OSS projects produce a rich set of software repositories, coming with a large number of versions reflecting their development and evolution history. These repositories consist of the source code, change logs, bug reports and mailing lists.

To know the facts related to such OSS project development, composition, and the possible risks associated with its use, one has to explore the huge information stored in the repositories. But often such repository contains heterogeneous information with different data representation, which also varies significantly from project to project. Thus a tool support for uniform data representation and customizable visualization mechanism is required to ease the comprehension of OSS projects.

In this paper, we present the tool Binoculars as the first step towards a graph based platform to comprehend and visualize OSS projects. Video demonstration of the tool Binoculars can be seen from [11].

2 Tool Support for Comprehending OSS Projects: A Review

This section presents a review on tool supports that offer different visualization approaches for comprehending OSS projects.

The tool, CodeSaw [10] provides a time series representation of social interaction data in juxtaposed displays. This tool explores links between one's contributions to that of social interactions. In this context, the tool Tesseract [10] explores the multi-perspective relationships in a project for a user-selected time period (i.e., the evoluiton), and represents them via four juxtaposed displays.

In [10], FASTDash was proposed as an interactive conflict management tool which provides a spatial representation of the shared code base by highlighting team members current activity. The tool CollabVS [10] addresses this issue at editing time, and provides a visual representation of conflicting code and a communication mechanism. The tool Palantir [10] performs similar task by graphically displaying the shared workspace to the developers with the information of what others are doing, and calculating the severity of such activities. Also the tool Augur [10] provides a line oriented view of the source code with colors for each pixel line indicating the location of the modification work and how recently it was conducted. This visualization allows to see how much activity has taken place recently and where that activity has been located.

In [10], the tool Ariadne utilizes call-graph approach to visualize social dependency of the developers due to code sharing. Similarly, the tool Expertise Browser [10] determines developers expertise from historical contributions.

Though the tools discussed above provide useful insight of OSS projects through different visualization approaches, yet none effectively explores graph based visualization of OSS projects. We thus add another dimension towards the comprehension of OSS projects by providing a graph based data representation and visualization. The principal argument here is that graph structures are most suitable for analyzing data that exhibits inherent relationships. In this context, the repository data produced by OSS projects exhibit strong relationships among them due to common work space sharing and exchange of information. For example, community members often share many technical competencies, values, and beliefs over online discussion forums. Similarly, code artifacts have interrelationships due to architectural dependency as well as due to contributions from multiple community members. Thus, OSS projects can be effectively comprehended through graph based representation and visualization.

3 Graph Based Visualization

In this section we concentrate on the available methods and techniques exploited in literature for graph based data representation and visualization. We also put a discussion on pros and cons of such techniques.

Graph based data representation and visualization can be effectively utilized when there exists inherent relations among data elements [3]. In such visualization, one can generate any number of links (i.e., edges) between two data points (i.e., nodes), and can easily traverse a given path through the data. This visual experience can be enhanced further by using layout algorithms, navigation and interaction methods, and incremental exploration mechanisms [3].

A significant amount of libraries, frameworks and toolkits are developed to support such visualization. To mention a few, GraphEd [4],the Tom Sawyer Software Graph Editor Toolkit [5], Graphlet [6], JUNG [1] provide APIs with different layout algorithms, customization, generic graphics and interprocess communication to create task-specific tools. Libraries and frameworks like GTL, LINK, GFC, GDT, and GVF provide support for both general and specific purpose graph visualization [3]. Within open source domain, Graphviz [10] and Zest [10] provides comprehensive set of APIs to support such visualization. Although there is no widely used standard for graph description formats, GML [7] and GraphXML [8] are available.

Despite of such benefits and supports for graph visualization, there are inherent shortcomings to such techniques. This includes, (a) difficulties in visualizing and comprehending large graphs. For example, a graph with thousands of nodes would cause performance bottleneck of the platform used and decrease the viewability (or usability) of such visualization significantly. In general, comprehension and detail analysis of data in graph structures is easiest when the size of the displayed graph is small [3]; (b) efficiency of a graph layout algorithm may be scale upto several hundred nodes, not beyond that; (c) time complexity for visualization, interaction and update of a graph is relatively high and increases with increase in graph size.

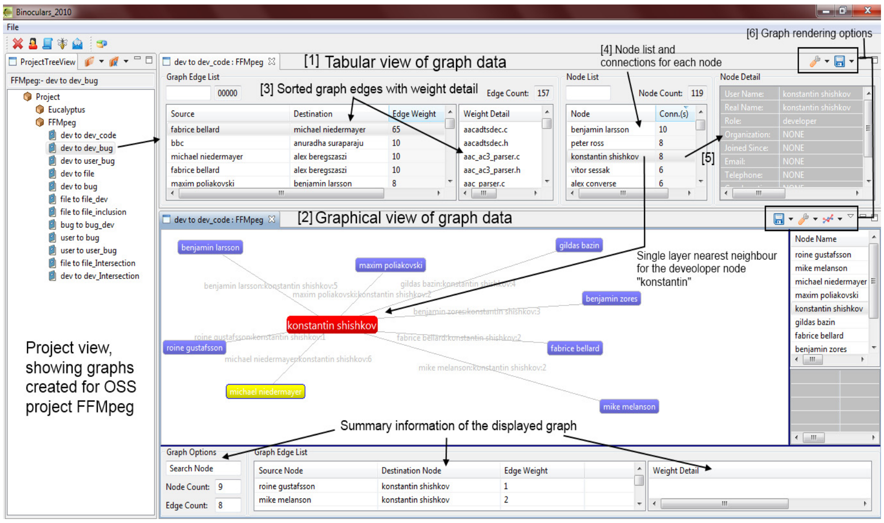
So far no single toolkit or framework mentioned above has proved to be sufficient to cope with these problems. Thus design decision for implementing an efficient graph visualization tool should ruminate the followings, (a) provide appropriate level of data abstraction. This keeps the graph structure small enough for effective comprehension and increase the efficiency of layout algorithms. To explore the graph, incremental exploration mechanism should be implemented, (b) time complexity of an algorithm should be measured accurately.

4 Binoculars: A Graph Based Platform

This section describes the requirements to design and implement a graph based visualization tool and presents Binoculars as a representative example. These requirements are derived considering the characteristics of OSS projects and the shortcomings of graph visualization techniques. The usability features of Binoculars are also presented. Fig. 1 shows the main interface of Binoculars.

First requirement is to provide an architectural model supporting well defined extension points for extending functionalities. As OSS analysis tools of this kind operate on project data, thus a good starting point is to model a generalized and standard data representation. This forms the system kernel and provides interfaces to build functionalities over it. The conceptual architecture of Binoculars is shown in Fig.2. In Binoculars, we defined a data repository structure to store both project and graph data (Fig.2), and use XML data format for representation (Fig.3(a)). XML is chosen over others due to (a) its inherent power of extensibility with new tags, (b) standard formating, and (c) graph generation and manipulation seems flexible with XML.

Having modeled such a repository, the next step is to decide what data to represent and how. For current implementation of Binoculars, we explored CVS or SVN checkouts, bug reporting system and mailing list. To represent data we adopt the following approach- first identified each entity within an OSS project which plays a role (either active or passive). For example, a community member


Binoculars_2010
File
ProjectTreeView
FFMpeg:- dev to dev_bug
Project
Eucalyptus
FFMpeg
dev to dev_code
dev to dev_bug
dev to user_bug
dev to file
dev to bug
file to file_dev
file to file_inclusion
bug to bug_dev
user to bug
user to user_bug
file to file_Intersection
dev to dev_Intersection
Project view, showing graphs created for OSS project FFMpeg
dev to dev_code : FFMpeg
[1] Tabular view of graph data
[2] Graphical view of graph data
[3] Sorted graph edges with weight detail
[4] Node list and connections for each node
[5]
[6] Graph rendering options
Single layer nearest neighbour for the deveoloper node "konstantin"
Summary information of the displayed graph
benjamin larsson
maxim poliakovski
gildas bazin
benjamin zores
konstantin shishkov
roine gustafsson
fabrice bellard
michael niedermayer
mike melanson
benjamin larsson:konstantin shishkov:5
maxim poliakovski:konstantin shishkov:2
gildas bazin:konstantin shishkov:4
benjamin zores:konstantin shishkov:3
roine gustafsson:konstantin shishkov:1
fabrice bellard:konstantin shishkov:2
michael niedermayer:konstantin shishkov:6
mike melanson:konstantin shishkov:2

Graph Edge List

00000 Edge Count: 157

Source	Destination	Edge Weight
fabrice bellard	michael niedermayer	65
bbc	anuradha suraparaju	10
michael niedermayer	alex beregszaszi	10
fabrice bellard	alex beregszaszi	10
maxim poliakovski	benjamin larsson	8

Weight Detail
aacadtsdec.c
aacadtsdec.h
aac_ac3_parser.c
aac_ac3_parser.h
aac_parser.c

Node List

Node Count: 119

Node	Conn.(s)
benjamin larsson	10
peter ross	8
konstantin shishkov	8
vitor sessak	6
alex converse	6

Node Detail

User Name:	konstantin shishkov
Real Name:	konstantin shishkov
Role:	developer
Organization:	NONE
Joined Since:	NONE
Email:	NONE
Telephone:	NONE

Node Name
roine gustafsson
mike melanson
michael niedermayer
maxim poliakovski
konstantin shishkov
gildas bazin
fabrice bellard
benjamin zores

Graph Options

Search Node

Node Count: 9

Edge Count: 8

Graph Edge List

Source Node	Destination Node	Edge Weight
roine gustafsson	konstantin shishkov	1
mike melanson	konstantin shishkov	2

Weight Detail

Fig. 1. User interface of Binoculars

(e.g., developer, user as active entities), and a code file, a single thread of mail and bug report (as passive entities). Then we identified unique set of attributes to describe each entity and provide values with the data mined from the sources presented above. In XML each such attribute is presented as a tag. Fig.3(a) shows an example of a code file representation.

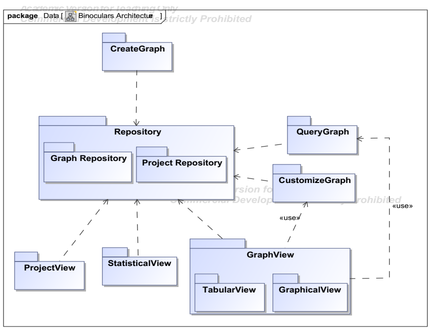
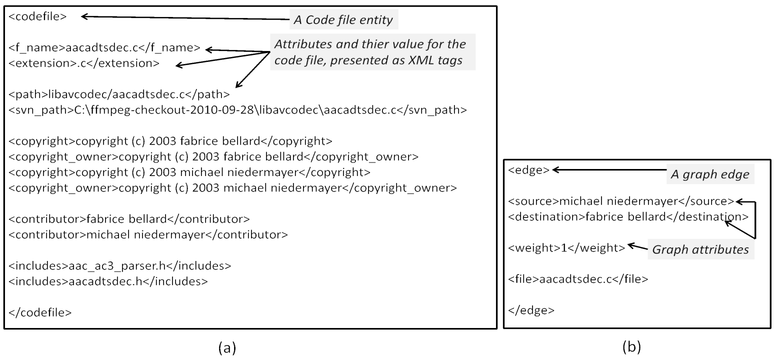


Fig. 2. Conceptual architecture of Binoculars

Third, a set of methods should be derived to transform repository data into graphs. These methods and the graph data should be reusable in a sense that one or more graph data can be reused by a method to generate new graphs. In Binoculars, graph data are stored using XML representation (example, Fig. 3(b)). As the methods operates on XML tags, thus one interface works for all repository data. As shown in Fig 2, CreateGraph module implements these graph generation methods, which are discussed in [9].

Fourth, provide a GUI support to visualize, render and manipulate graph data. This GUI design for graph visualization is often constrained by the limiting factors of the available visualization techniques discussed in section 3. To cope with these issues we took the following measures. We provided a two-way visualization of a graph, e.g., tabular and graphical (Fig.1 items 1,2). Tabular view provides complete graph information consisting of (a) Graph with nodes and (weighted) edges; (b) Node list with degree count for each node; (c) description of each node; (d) Summary data on graph; and (e) Options to render a graph (Fig.1, item 3,4,5,6, respectively). Thus user can get complete graph data with detail information in real time for large graphs with thousands of nodes. Then, depending on the option selected for rendering a graph, a modified (or abstracted) version of the graph (in tabular view) can be viewed in graphical form. As shown in Fig.1 item 2, a single level nearest neighbor graph showing the developers to whom developer "Konstantin" has direct communication in FFMpeg project [10]. Hence the graphical view (Fig.1 item 2) always shows a tailored version of the complete graph provided in tabular view (Fig.1 item 1), thus minimizing the performance bottleneck of layout algorithms.



```
<codefile>

<f_name>aacadtsdec.c</f_name>
<extension>.c</extension>

<path>libavcodec/aacadtsdec.c</path>
<svn_path>C:\ffmpeg-checkout-2010-09-28\libavcodec\aacadtsdec.c</svn_path>

<copyright>copyright (c) 2003 fabrice bellard</copyright>
<copyright_owner>copyright (c) 2003 fabrice bellard</copyright_owner>
<copyright>copyright (c) 2003 michael niedermayer</copyright>
<copyright_owner>copyright (c) 2003 michael niedermayer</copyright_owner>

<contributor>fabrice bellard</contributor>
<contributor>michael niedermayer</contributor>

<includes>aac_ac3_parser.h</includes>
<includes>aacadtsdec.h</includes>

</codefile>
```

```
<edge>

<source>michael niedermayer</source>
<destination>fabrice bellard</destination>

<weight>1</weight>

<file>aacadtsdec.c</file>

</edge>
```

A Code file entity
Attributes and thier value for the code file, presented as XML tags
(a)
A graph edge
Graph attributes
(b)

Fig. 3. (a) XML representation of a code file repository in FFMpeg project. (b) XML presentation of a developers relationship graph generated from (a).

Other options for rendering a graph includes (Fig.1, item 6), customization based on (a) given range of edge weights, (b) selected set of nodes or edges from the original graph, (c) a given attribute value (e.g., gio-location= "america").

None-the-less, searching, sorting, zooming, and saving graph data in XML format can also be performed. As in Fig. 2, rendering mechanisms are implemented in QueryGraph and CustomizeGraph module, and the visualization are handled by ProjectView, GraphView and StatisticalView modules.

Fifth, selection of platform and packages for implementation should be steered by it's easy extension and distribution. Our choice in this issue is to release

Binoculars as an OSS. Thus we utilized well established and maintained OSS platforms and packages, e.g., Eclipse, Eclipse RCP, ZEST, DOM, and JFreeChart. Reference to these platforms can be found here [10].

5 Discussion and Future Work

In this paper we put a discussion on the requirements to model and implement a graph based platform for comprehending OSS projects, and present the tool Binoculars as a first step towards establishing such a platform. Our starting point is the design of a repository to capture the essence of OSS projects and then built tool functionalities over it to operate on repository data. We also discuss the inadequacy of graph visualization techniques and distill possible solution.

Future extension of this tool includes, (a) visualization on the evolution of socio-technical aspects of OSS projects, (b) Incremental exploration mechanism on the displayed graph, and (c) a formal language query support.

References

1. Souza, C.R.B., Quirk, S., Trainer, E., Redmiles, D.F.: Supporting collaborative software development through the visualization of socio-technical dependencies. In: ACM SIGGROUP Conference on Supporting Group Work, pp. 147–156 (2007)
2. Mockus, A., Herbsleb, J.: Expertise browser: A quantitative approach to identifying expertise. In: ICSE, pp. 503–512 (2002)
3. Herman, I., Melancon, G., Marshall, M.S.: Graph visualization and navigation in information visualization: A survey. In: TVCG, IEEE, vol. 6(1), pp. 24–43 (2000)
4. Di Battista, G., Eades, P., Tamassia, R., Tollis, I.G.: Graph Drawing: Algorithms for the Visualization of Graphs. Prentice Hall (1999)
5. Becker, R.A., Eick, S.G., Wilks, A.R.: Visualizing Network Data. In: TVCG, IEEE, vol. 1(1), pp. 16–28 (1995)
6. Argawal, P.K., Aronov, B., Pach, J., Pollack, R., Sharir, M.: Quasi–Planar Graphs Have a Linear Number of Edges, pp. 1–7. Springer, GD (1995)
7. Himsolt, M.: GML — Graph Modelling Language. University of Passau (1997)
8. Herman, I., Marshall, M.S.: GraphXML. Reports of the Centre for Mathematics and Computer Sciences (1999)
9. Syeed, M.M., Aaltonen, T., Hammouda, I., Systä, T.: Tool Assisted Analysis of Open Source Projects: A Multi-Faceted Challenge. IJOSSP 3(2), 43–78 (2011)
10. References (2012), http://rajit-cit.wix.com/syeed#!refrences
11. Binoculars Demo (2012), http://www.youtube.com/watch?v=cMoYq6JOpQE

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland

ISBN 978-952-15-3525-3
ISSN 1459-2045