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 indicated by 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" [16]. The technological platform is often a software system providing various levels of openness for developers to add applications.

[*]OpenSym '14, August 27 - 29 2014, Berlin, Germany
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3016-9/14/08...$15.00.
http://dx.doi.org/10.1145/2641580.2641586

In the case of FLOSS (Free/Libre Open Source Software) ecosystems, the software ecosystem 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 [11], Eclipse [8], and Ruby [27].

FLOSS ecosystems have been the subject of much socio-technical research (cf., e.g., [15]) studying the possible relationships between their social domain represented by the developer community and their technical domain associated with the software produced. It is in this context that this 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 which 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 [32]. 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 parts of the paper is organized as follows. Section 2 offers a discussion on the related works to form the ground for this study. Section 3 introduces a number of key concepts related to the research questions explored. Section 4 presents our study design. Results are reported and discussed in Section 5. Future extension to this work is presented in brief in Section 6, followed by a discussion on the possible limitations and threats to validity in Section 7.

Finally, Section 8 concludes the paper.

2. RELATED WORK

In this section we highlight prior work that falls within the scope of this research.

2.1 Studies on Socio-Technical Congruence

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 claims have 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 [21] and product quality [1].

On the other hand, lack of socio-technical congruence is often correlated with lower productivity with increased number of code changes [3][9] and negative performance level [31] within the organization. It is, thus, advised to measure congruence to evaluate the actual coordination quality within the organization [3].

2.2 Studies of OSS Ecosystems

Social interaction in open source software ecosystems, i.e. the interaction and dependencies among 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 [18] analyze the Ruby Git repository, graph the developer and gem interaction and define three developer roles according to their analysis.

Dabbish et al. [6] analyzed "social coding" in GitHub per se. Their results point to that visible feedback on GitHub support collaboration and learning.

Scacchi [29, 30] 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 [19] analyze the developer contribution in sourceforge.net to reveal the tendency of developers to contribute to single projects in the repository. Jergensen et al. [17] study GNOME developer participation and showed that the onion model hypothesis of new contributors in FOSSD projects does not apply to this project. Ververs et al. [33] 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 [23, 24, 12]. Software evolution in ecosystems is also addressed by Yu et al. [34]. They analyse the evolution of software from the biological viewpoints: evolution in term of symbiosis and in terms of Darwinism. Robbes et al. [26], on the other hand, study the evolution of OSS software in terms of the ripple effect, i.e. the effects to software when APIs change.

3. DEFINITIONS AND RESEARCH QUESTIONS

In this section we define the set of concepts used in this study.

3.1 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's architecture reflects the organizational structure of its development organization [5, 20]. In [13], Conway's Law is considered bidirectional and thus claimed to be true in reverse as well. This means the organization pattern within a developer community should reflect the architectural dependencies in the developed software. Thus, Conway's Law can effectively be interpreted as a basis for studying the social and technical interdependency within a software project [25].

3.2 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 dependencies in the software) and the actual coordination activities carried out by project members (i.e., within the members of the developer community) [20]. 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 can be analyzed via the technical entities [20]. 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.

3.3 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 those 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.

3.4 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 case of Ruby ecosystem, the communication history is deduced using the communication traces in the issue tracking system of the GitHub[10] software development hosting site for gems hosted there.

3.5 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 have direct communication 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)).

3.6 Research questions

In this work, we study socio-technical congruence in the Ruby ecosystem by addressing two research questions:

1. *Can socio-technical congruence be studied at the ecosystem level so as to yield insight into the social and technical organization of software ecosystems?*
2. *To which extent can developer's interaction within the Ruby ecosystem approximate the actual relationship (or dependencies) among the ecosystem gems?*

In order to perform a study as 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. Investigation of Research Question 2 will be done by determining 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 4).

4. 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.

4.1 Case and Subject Selection

We use the Ruby gems ecosystem as a unit of analysis in order to explore our research questions. The Ruby gems website, RubyGems.org [28], 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 socio-technical dependencies in a software 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[10] as a source code repository and for collaboration. In our data set (see Section 4.2), 72% of the specifications of gems referenced GitHub.

4.2 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: ",
```

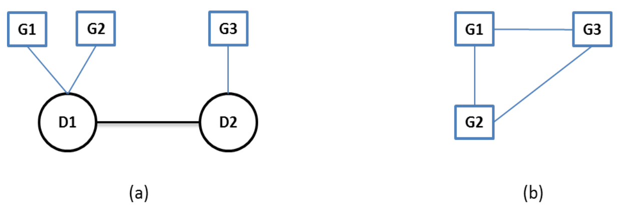


Figure 1: (a) Explicit Coordination Network with contribution to code base (b) Corresponding Implicit Architecture

```
  "closed_by": {
    "login": "alto",
    ...
  }
  ...
}
```

4.3 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.

4.4 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 the `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 gem records listed in the JSON file, 12,520 Ruby gem records had developer communication records in GitHub. Thus, for further analysis, we restricted the data set to 12,520 Ruby gem records.

4.5 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 4.4.

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 3.3 and 4.2). 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 4.4 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 3.4, 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 4.4. 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 3.5. 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 categorization 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 [32] 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.

As an illustration, consider the explicit architecture presented in Figure 4 and the implicit architecture shown in Figure 1(b). A congruence measure between the two using equation (1), would be as follows,

Congruence
$= \frac{|[E_{G1-G3}, E_{G1-G4}] \bigcap [E_{G1-G2}, E_{G1-G3}, E_{G2-G3}]|}{\left|[E_{G1-G3}, E_{G1-G4}]\right|} \times 100$
$= \frac{1}{2} \times 100$
$= 50\%$

Here, the explicit architecture is taken as the Ref_A and the implicit architecture is considered as the $Analogous_A$.

Additionally, it should be notated that the congruence would be 0% if developers that communicated in one project G1 never communicated with people in G3 and G4 and 100% if people from G1 also communicated in with people communicating in G3 and G2.

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 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.

5. RESULT ANALYSIS

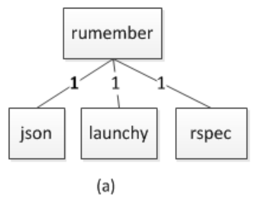
In this section we investigate the research questions primarily based on the data analysis presented in Section 4.

5.1 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 product's architecture. In this setup, the main components of the product define the organization's key subtasks [14] and become the source of most relevant information pertinent to the task dependencies that define coordination needs among the developers [7].

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 [22]. 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



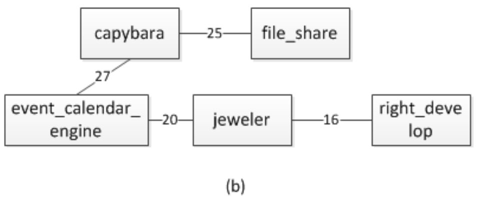
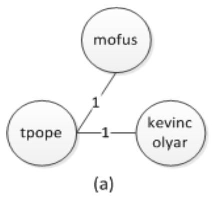


Figure 2: (a) Example Explicit Architecture for gem *rumember* (b) A partial snapshot of the Explicit Architecture



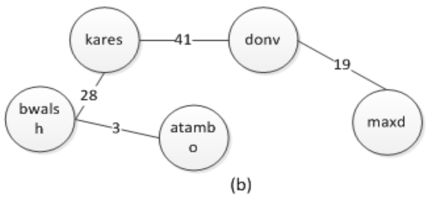


Figure 3: (a) Example Explicit Coordination Network for gem *rumember* (b) A partial snapshot of the Explicit Coordination Network

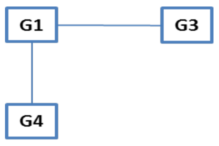


Figure 4: (a) Example Explicit Architecture

by examining the explicit architecture and the explicit coordination network. The explicit architecture reveals 141,029 edges among the 12,520 gems. Each edge in this architecture shows either development or runtime dependency between two gems. Similarly, the explicit coordination network generates 186,136 number of relationship edges among 55,454 developers. Each edge in this network shows communication between two developers as shown in the GitHub issue tracking system. This quantity of relationships among the projects, both from social and technical domains, sets the ground for socio-technical congruence analysis in the Ruby ecosystem.

5.2 Socio-Technical Congruence in the Ruby Ecosystem

Results obtained from the socio-technical congruence analysis carried out for Ruby gems ecosystem are reported in Table 1. Observed results positioned this study to offer the following insights:

The congruence measure, as a whole, is low for all the seven edge weight categories of the explicit coordination network, as shown in the trend chart in Figure 5. 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 3.4, the edge weight in the explicit coordination network measures the strength of collaboration

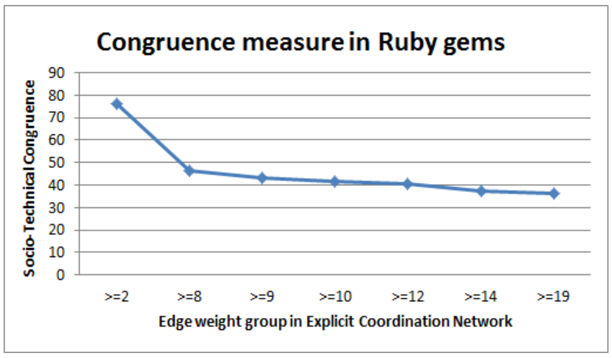


Figure 5: Socio-Technical Congruence in Ruby Ecosystem

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 indicated by dependencies among the gems.

The following explanations can be offered in support to this inference. Interdependency among projects in an ecosystem often exist at a higher abstraction level and may be decoupled in general. For instance, in the case of the Ruby ecosys-

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 ($\frac{Ref_A \bigcap Analogous_A}{\lvert Ref_A \rvert} \times 100$)
12,520	≥ 2	28,361	10,472,491	21,604	76.2
	≥ 8	28,361	3,283,494	13,121	46.3
	≥ 9	28,361	2,765,429	12,269	43.3
	≥ 10	28,361	2,448,552	11,737	41.4
	≥ 12	28,361	2,283,553	11,488	40.6
	≥ 14	28,361	1,948,130	10,613	37.5
	≥ 19	28,361	1,843,492	10,211	36.1

tem, 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 [21], might imply better coordination among the developers, which can give ground to better support the management of change and maintenance of quality.

However, Socio-Technical congruence within an FLOSS project have already been measured in [32]. In this paper, Socio-Technical congruence has been measured using equation (1) during the entire lifespan of the FreeBSD project. The reported results identify that congruence is higher in the FreeBSD project which has a stable evolution history for the last seven stable releases of the project. This indicates 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.

6. FUTURE WORK

Further study in relation to socio-technical dependency at ecosystem level could fork in several directions. One way to extend the work is to examine socio-technical congruence at different abstraction levels of an ecosystem. For instance, initiating the study for individual project, project clusters that are tightly connected, and the whole ecosystem for a given period of evolution. This would result deeper insight and comprehensive understanding on the topic.

Additionally, a statistical analysis need to be performed in devising the significance of the congruence measure to that of the success factors (e.g., quality, sustainability) at different abstraction level of the ecosystem.

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): This paper reoports on an empirical study of the Conway's Law on an ecosystem level. Our empirical data is 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 the Conway's Law at an ecosystem level, at least to our knowledge, makes it unclear to which our results can be generalised. Our study intends to trigger additional studies of this kind, in different ecosystems, in order to reach conclusions that can be generalised.

Internal validity (confounding factors that 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 specifications referenced GitHub, and we successfully extracted the issues of a total of $33,960$ GitHub gems. 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 12,520 projects. Thus considering the whole population of gems our current approach may lead to biased observation. This might pose threats to construct validity of this study.

8. 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 indicated by the dependencies among its ecosystem projects. In contrast, the individual ecosystem projects themselves have 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 an ecosystem health 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.

9. ACKNOWLEDGEMENTS

This work has been partially funded by the TUT Graduate school, Finland.

10. 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] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1277–1286. ACM, 2012.

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

[8] Eclipse – the eclipse foundation open source community website. `http://www.eclipse.org`. Accessed 2014-05-02.

[9] 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.

[10] GitHub. Build better software, together. `https://www.github.com`. Accessed 2014-05-02.

[11] GNOME. `http://www.gnome.org`. Accessed 2014-05-02.

[12] 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.

[13] 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.

[14] E. V. Hippel. Task partitioning: an innovation process variable. In *Research Policy*, volume 19, pages 407–418, 1990.

[15] S. Jansen, S. Brinkkemper, and M. A. Cusumano. *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry.* Edward Elgar Publishing, 2013.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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).

[24] 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.

[25] E. S. Raymond. The new hacker's dictionary (3rd ed.). In *Cambridge, MA, USA: MIT Press*, 1996.

[26] 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.

[27] Ruby programming language. `https://www.ruby-lang.org`. Accessed 2014-05-02.

[28] RubyGems.org. Your community gem host. `https://www.rubygems.org`. Accessed 2014-05-02.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] L. Yu, S. Ramaswamy, and J. Bush. Symbiosis and software evolvability. *IT Professional*, 10(4):56 –62, july-aug. 2008.