Ecosystems in GitHub and a Method for Ecosystem Identification using Reference Coupling

Kelly Blincoe, Francis Harrison and Daniela Damian
Software Engineering Global InterAction Lab
University of Victoria
Victoria, BC, Canada
Email: kblincoe@acm.org, francish@uvic.ca, danielad@uvic.ca

***Abstract*—Software projects are not developed in isolation. Recent research has shifted to studying software ecosystems, communities of projects that depend on each other and are developed together. However, identifying technical dependencies at the ecosystem level can be challenging. In this paper, we propose a new method, known as reference coupling, for detecting technical dependencies between projects. The method establishes dependencies through user-specified cross-references between projects. We use our method to identify ecosystems in GitHub-hosted projects, and we identify several characteristics of the identified ecosystems. We find that most ecosystems are centered around one project and are interconnected with other ecosystems. The predominant type of ecosystems are those that develop tools to support software development. We also found that the project owners' social behaviour aligns well with the technical dependencies within the ecosystem, but project contributors' social behaviour does not align with these dependencies. We conclude with a discussion on future research that is enabled by our reference coupling method.**

I. Introduction

Software ecosystems, defined as "a collection of software projects which are developed and which co-evolve together in the same environment" [1], have become an area of interest in recent research. Since software projects are not typically developed in isolation, studying a software project without examining its surrounding ecosystem is incomplete. Thus, analysis of software ecosystems has emerged as a novel new research area in recent years.

Projects in an ecosystem depend on one another [1]. The technical dependencies that exist between projects define the structure of the ecosystem [2]. Thus, identifying technical dependencies between software projects is a useful way to identify ecosystems. However, identifying technical dependencies between projects on a large scale has proven to be difficult [3]. Existing static dependency analysis approaches do not identify dependencies across projects. Methods for extracting dependencies from a project's source code have been proposed [2], [3], [4], but they require large amounts of memory and computation time [5]. Thus, they cannot be employed across a large set of projects. Other methods [1], [6], [7], [8], [9] avoid analyzing source code by obtaining technical dependency information from a project's configuration files, but this information is not always available or accurate. Without a way to establish dependencies between projects, researchers cannot fully understand a project's ecosystem.

To identify technical dependencies, we turn to cross-references on GitHub, a social code hosting service. GitHub encourages collaboration between users both within and across projects through its transparent interface and built-in social features. With GitHub Flavored Markdown[1], when a user cross-references another repository in a pull request, issue or commit comment a link to the other repository is automatically created. This introduces a way for developers to bring awareness across repositories. In this paper, we investigate whether these cross-references indicate a technical dependency between the two repositories by examining a set of these cross-references. We found that the cross-references are a good conceptualization of technical dependencies between projects, and we highlight several common types of technical dependencies seen in these cross-reference comments. We call our method for identifying technical dependencies reference coupling.

With an ability to identify technical dependencies between a large number of projects, ecosystems of densely connected projects can be identified. We use a popular community detection algorithm [10] to identify ecosystems across all GitHub-hosted projects. Ecosystem identification is important to help developers understand how their tasks fit into the big picture and who they need to coordinate their changes with at the ecosystem level [1], [11]. For open source ecosystems, it is also important for attracting new contributors [1] since project's within an ecosystem are more likely to attract attention. In this paper, we analyze the ecosystems identified using our methods on GitHub-hosted projects and identify a set of properties that characterize the ecosystems. Conway Law. To complement our focus on technical dependencies within ecosystems, we also investigate the social behaviour of project owners/contributors in relation to these ecosystems. Our analysis was guided by the following research questions:

RQ1: Do cross-references to other projects in issue, pull request, and commit comments indicate the existence of a technical dependency between the two projects?

RQ2: What ecosystems exist across GitHub-hosted projects and what is their structure?

RQ3: Do the project owners' and contributors' social behaviours align with the technical dependencies?

[1]https://help.github.com/articles/github-flavored-markdown/

The rest of the paper is structured as follows: Our research methods and results are presented in Section II for RQ1, Section III for RQ2, and Section IV for RQ3. In Section V, we summarize our findings and discuss open questions for future research. Section VI provides an overview of related work in software ecosystems and dependency conceptualizations. We provide a brief conclusion in Section VII.

II. Reference Coupling

RQ1: Do cross-references to other projects in issue, pull request, and commit comments indicate the existence of a technical dependency between the two projects?

A. Research Method

We studied the cross-references made to other repositories in comments on issues, pull requests and commits. We obtained data from the GHTorrent [12] project, which provides a mirror of the GitHub API data. GHTorrent obtains its data by monitoring and recording GitHub events as they occur. We used the MySQL 2014-04-02 dataset to obtain information on the projects. This dataset contains data on 2,399,526 repositories, 3,426,046 users, and their events - including commits, issues, pull requests and comments. Since the MySQL database contains only the first 256 characters of comments, we obtained all comments from GHTorrent's main MongoDB server in May 2014. The MongoDB contains the full text of all comments.

Cross-references follow the pattern User/Project#Num (e.g. rails/rails#123) or User/Project@SHA. We performed pattern matching on all comments in the GHTorrent database to identify these cross-references. We define a project as a repository and all of its forks as recommended by [13]. Since we are interested only in relationships between projects, we filtered the cross-references to ignore those where one project is a fork of the other project. We identified 89,784 comments with a cross-reference to another project.

To answer RQ1 and verify that these cross-references are a valid conceptualization of dependencies, we examined 198 random comments which cross-referenced another project. We classified a comment as a technical dependency if the comment described a work dependency, either direct or indirect, between the two projects.

We examined the comments classified as technical dependencies to identify the types of dependencies that exist through these cross-reference relationships. We used a grounded theory approach to identify types of dependencies [14]. We conducted open coding on the cross references, grouping conceptually similar comments into categories. We stopped when we achieved saturation after 49 comments. One person, familiar with software development practices, performed the qualitative coding.

B. Results

Most of the cross-reference comments (90%) were classified as technical dependencies. The remaining 10% of comments did not contain enough details for proper classification. Many of these comments simply referenced another repository using the pattern as described above with no additional text provided.

We identified two main types of dependencies:

Dependency between the two projects. The most common type of dependency found was a direct technical dependency between the two projects. An example of a direct technical dependency is when an issue created in one project depends on a fix/update in another project. Another example is when a project needs to be updated based on changes made in another project.

Below we provide three examples of cross-reference comments that are indicative of direct technical dependencies between the two projects. Project names follow the pattern user/repository where user is the owner's GitHub login and repository is the name of the project repository.

Issue #449 on the sensu/sensu project describes an issue that is the result of the interaction between the sensu/sensu code and the ruby-amqp/amq-client library. The comment references a commit on the ruby-amqp/amq-client that fixes the issue.

> *"I verified that the problem is still the one referenced in ruby-amqp/amq-client#14. This fix is not merged with amq-client's '0.9.x-stable' branch. This is why I am still hitting it. The commit ruby-amqp/amq-client@60f1c59 is the fix but it resides only in the master branch."*

Issue #8 on the tsujigiri/axiom project notes that changes must be made to the code base to allow an upgrade to the latest release of the ninenines/cowboy project.

> *"Upgrade Cowboy: After Cowboy 0.6.1 Cowboy's http_req record was made opaque and can not be used directly anymore. I didn't really have the time yet to look into it, but it looks like we just need to remove all references to the record from the documentation and add directions on how to access cowboy_req:req() via the cowboy_req functions. See ninenines/cowboy#266 and ninenines/cowboy#267."*

Commit 81bbbec21c04b6392f6892f7735243387d295337 on the joyent/node project closes isaacs/node-graceful-fs issue #6, which describes a problem in the isaacs/node-graceful-fs code stemming from the use of joyent/node. GitHub allows automatic closure of issues through commit comments, even when the commit is in a different repository[2].

> *"This fixes isaacs/node-graceful-fs#6."*

Both projects depend on a third project. We also identified some cases where the comments describe a dependency on a third project that is not cross-referenced. For example, everzet/capifony's pull request #376 cross-references composer/composer's issue #1453, but the problem stems from the use of the symfony/symfony project. After identifying the source of the problem, a new issue (#411) is created on the

[2]https://github.com/blog/1439-closing-issues-across-repositories

everzet/capifony project that identifies the changes that need to be made to the way the symfony environment is set so that the composer/composer code executes correctly.

> *"As described in #376 capifony should execute composer with the right symfony environment set. Currently, with `--no-scripts` option removed in #376, composer is always executing symfony scripts with default dev environment."*

Cross-references to other repositories appearing in GitHub comments, therefore, do indicate the existence of a technical dependency. We call this conceptualization of dependencies between projects reference coupling.

III. GitHub Ecosystems

RQ2: What ecosystems exist across GitHub-hosted projects and what is their structure?

A. Research Method

We constructed a network of the technical dependency relationships established through reference coupling as described in Section II. The *Dependency Network* is defined as a directed graph $G_d =< V, E >$. The set of vertices, denoted by V, is all GitHub projects involved in at least one cross-reference. There are 18,533 projects in this set. The set of edges, denoted by E, is a set of node pairs $E(V) = \{(x, y)|x, y \in V\}$. If the project represented by node x_i cross-referenced the project represented by node y_j, there is a directed edge from x_i to y_j. The weight of each edge is the count of cross-references for the pair of projects. We filtered the edges to only consider dependencies between nodes if the pair of projects have been cross-referenced two or more times to capture only the stronger dependencies.

To identify ecosystems across projects hosted on GitHub, we used the popular Louvain community detection method [10] on the Dependency Network. The Louvain method is a greedy optimization method that aims to partition a network into communities of densely connected nodes and optimize the modularity of the network. Modularity is defined as "the number of edges falling within [communities] minus the expected number in an equivalent network with edges placed at random [15]." The Louvain method is comprised of two steps. It first optimizes modularity locally by looking for small communities. Then it aggregates the nodes in each small community and builds a new network with these aggregated nodes. It iterates on these two steps until the modularity is maximized. The Louvain method outperforms all other community detection methods in terms of both the modularity that is achieved and the computation time [10].

High modularity scores indicate that there are dense connections within the communities but sparse connections across communities, showing that an optimal solution has been found. When high modularity scores are obtained, the communities have significant real-world meaning [10]. In our network, the identified communities represent sets of projects densely connected by technical dependencies. Since dependencies that exist between projects define the structure of an ecosystem [2], these communities represent software ecosystems.

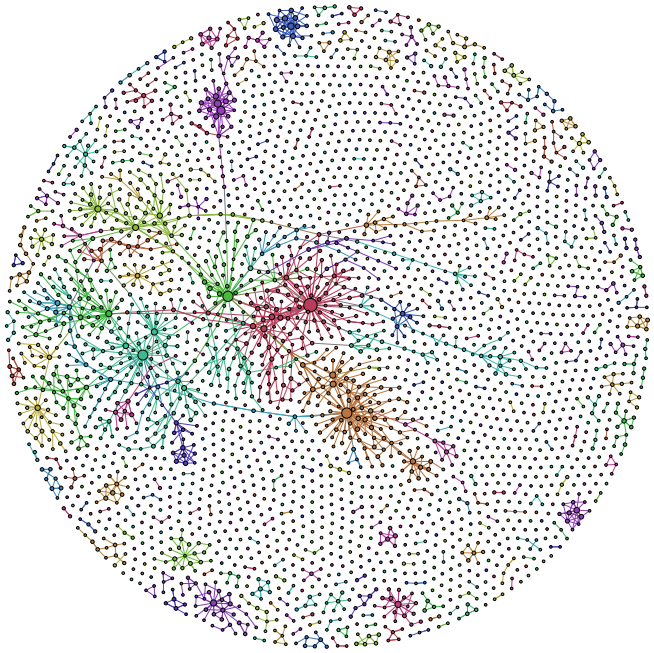

Fig. 1. All GitHub projects with cross-references. The largest connected component (or giant component) is easily identified as the well-connected subgraph appearing in the center of the graph.

To identify properties of the identified ecosystems, we analyzed visualizations of the Dependency Network. We used the Gephi [16] graphing tool to create visualizations. We inspected visualizations of the network to visually identify patterns. We triangulated the patterns visible in the network with statistics of the network. We also examined the type of projects prominent in each ecosystem.

B. Results

Of the 18,533 projects in the Dependency Network, 10,484 (57%) are a part of the largest connected component (commonly referred to as the giant component [17]), which is the largest subgraph in which every node is connected to every other node by some path. The connected components isolated from the giant component are primarily comprised of *same owner* communities in which all nodes in the connected component are projects owned by the same GitHub user or organization. For example, the second largest connected component is comprised of 65 nodes, of which, all but two are owned by GitHub user deathcap[3]. Figure 1 shows the full Dependency Network, though for visibility we only display nodes with degree of 3 or greater. As visible on the graph, most of the nodes isolated from the giant component are connected to only a small number of nodes. In fact, 75% of nodes not in the giant component are connected to only one other node.

Since we are most interested in studying the popular GitHub ecosystems, we focus our analysis on the interconnected part

[3]https://github.com/deathcap

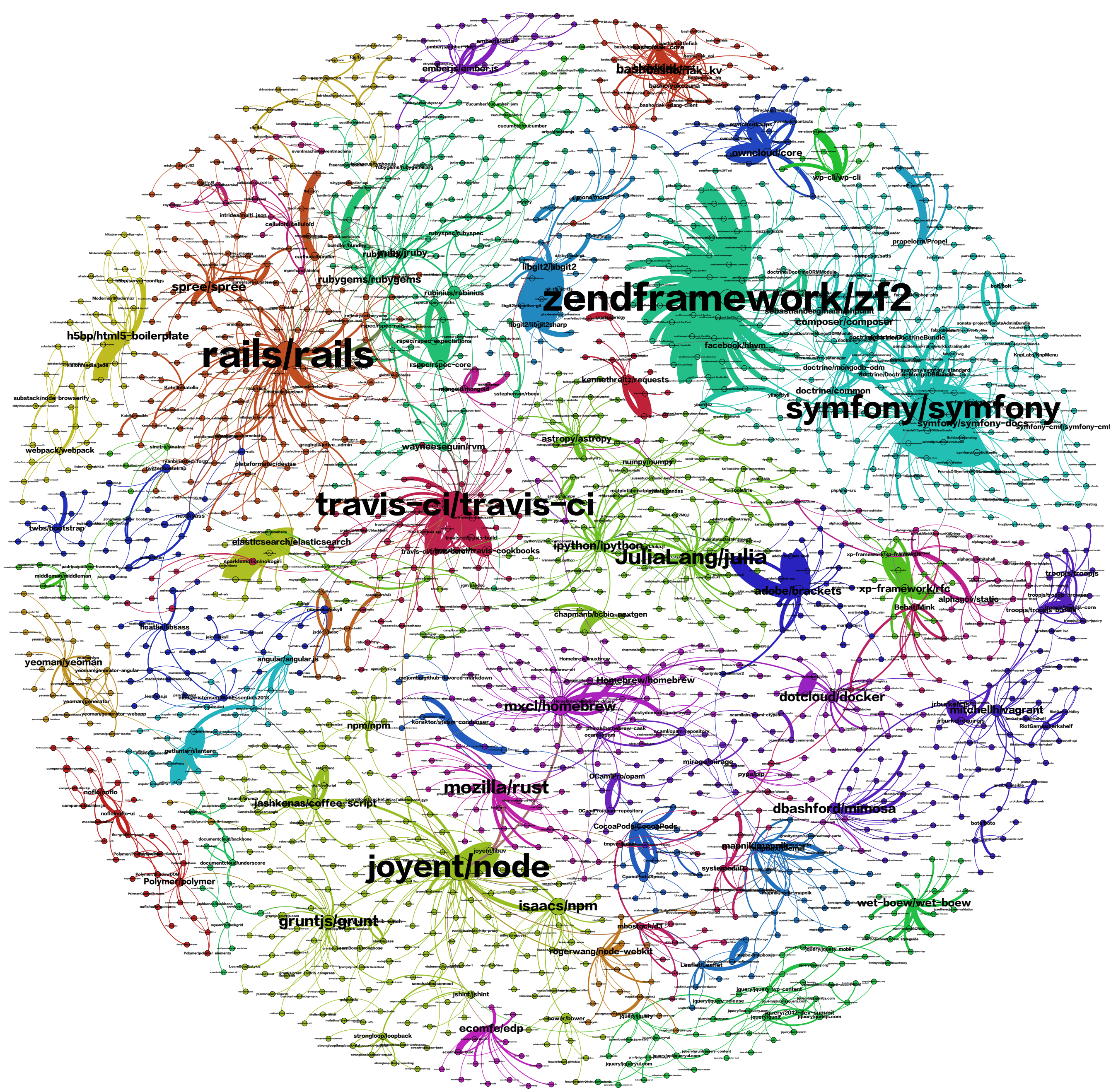

Fig. 2. Ecosystems in the largest connected component of GitHub-hosted projects. Project names follow the pattern user/repository where user is the owner's GitHub login and repository is the name of the project repository.

of the network or the giant component. Figure 2 shows the giant component. The color of the nodes represent communities as detected by the Louvain method. We obtained a modularity score of 0.913 (out of a possible range of 0 to 1). This high modularity score indicates that the detected communities are much more tightly connected by technical dependencies than would appear in a random graph.

There were 43 ecosystems identified in this network. Nodes are sized according to their authority to display the nodes that are more prominent in each ecosystem. When a node has a high number of cross-reference relationships pointing to it, it has a high authority value [18]. Table I shows the most well-connected project node (highest Authority value) in each of the ecosystems.

C. Properties of GitHub Ecosystems

Ecosystems revolve around one central project. As depicted in Figure 2, each ecosystem appears to revolve around one main project. In Table I, the most well-connected project node in each ecosystem is listed along with a description

TABLE I
ECOSYSTEMS IN GITHUB. DETAILS OF THE MOST WELL-CONNECTED NODE IN EACH ECOSYSTEM.

Project	Description	Stars	Ecosystem Size	Degree (in,out)
joyent/node	Framework	39,373	10.08%	69 (53,16)
symfony/symfony	Framework	10,985	8.46%	93 (53,40)
rails/rails	Framework	29,744	7.92%	93 (65,28)
JuliaLang/julia	Programming Language	5,531	6.74%	51 (35,16)
rubygems/rubygems	Package Manager	1,304	6.04%	22 (14,8)
mxcl/homebrew	Package Manager	13,723	3.94%	48 (21,27)
zendframework/zf2	Framework	5,841	3.88%	72 (65,7)
travis-ci/travis-ci	Development Tool (Continuous Integration Platform)	3,693	3.50%	70 (54,16)
wet-boew/wet-boew	Framework	688	3.34%	19 (15,4)
twbs/bootstrap	Framework	41,828	3.29%	9 (9,0)
dbashford/mimosa	Development Tool (Browser development)	472	2.43%	25 (20,5)
h5bp/html5-boilerplate	Framework	31,926	2.37%	19 (15,4)
mitchellh/vagrant	Framework	9,274	2.10%	23 (15,8)
libgit2/libgit2	Library	5,161	2.05%	20 (11,9)
Behat/Mink	Development Tool (Testing)	673	1.99%	13 (9,4)
OCamlPro/opam	Package Manager	118	1.89%	9 (8,1)
basho/riak	Database	2,520	1.83%	27 (18,9)
Polymer/polymer	Library	8,787	1.83%	16 (11,5)
mapnik/mapnik	Development Tool (Toolkit for developing mapping applications)	1,003	1.78%	20 (12,8)
mozilla/rust	Programming language	5,604	1.78%	36 (29,7)
alphagov/static	Other (GOV.UK static files/resources)	67	1.73%	13 (10,3)
adobe/brackets	Development Tool (code editor)	23,921	1.46%	26 (16,10)
CocoaPods/CocoaPods	Development Tool (dependency manager)	5,711	1.46%	14 (9,5)
yeoman/yeoman	Development Tool (web development tools)	7,246	1.46%	18 (13,5)
angular/angular.js	Framework	42,950	1.40%	12 (8,4)
dotcloud/docker	Development Tool (application container engine)	14,270	1.35%	24 (19,5)
emberjs/ember.js	Framework	14,185	1.29%	20 (12,8)
owncloud/core	Other (personal cloud storage tool)	3,222	1.19%	26 (13,13)
typhoeus/typhoeus	Library	2,465	1.19%	6 (4,2)
facebook/hhvm	Other (Virtual machine)	11,506	1.08%	15 (10,5)
celluloid/celluloid	Framework	2,855	0.86%	9 (6,3)
xp-framework/rfc	Framework	0	0.86%	16 (14,2)
rogerwang/node-webkit	Framework	19,737	0.86%	16 (11,5)
ecomfe/edp	Development Tool (front-end development platform)	264	0.86%	18 (15,3)
kennethreitz/requests	Library	13,812	0.81%	13 (10,3)
documentcloud/underscore	Library	7,135	0.81%	6 (4,2)
middleman/middleman	Development Tool (website generator)	4,179	0.75%	8 (5,3)
elasticsearch/elasticsearch	Other (search and analytics tool)	10,700	0.70%	11 (11,0)
chapmanb/bcbio-nextgen	Other (RNA-seq analysis tool)	173	0.59%	10 (9,1)
wp-cli/wp-cli	Development Tool (command line interface for WordPress)	1,968	0.59%	13 (9,4)
cucumber/cucumber	Development Tool (Testing)	5,142	0.49%	7 (4,3)
jsdoc3/jsdoc	Development Tool (API documentation generator)	2,909	0.49%	6 (3,3)
propelorm/Propel	Development Tool (Object-Relational Mapping)	893	0.49%	7 (7,0)

of the project, the number of stars the project has, the size of the associated ecosystem, and the node's degree. Each of these projects has a higher in-degree than out-degree with the exception of the mxcl/homebrew project. On the other hand, low-degree project nodes are four times as likely to be dependent on another project than they are to have a project depend on them. This shows that ecosystems are being formed around a central project with the other projects in the ecosystem mostly depending on that central project. This results in a star pattern. The twbs/bootstrap ego network (Figure 3) clearly depicts this pattern within the graph.

Predominant type of ecosystems is software development support. Interestingly, nearly all of the ecosystems are centered around projects whose purpose is to support software development, such as frameworks, libraries and programming languages. In fact, of the 43 ecosystems, there are only 5 whose purpose is not to support software development. There are 13 frameworks, 5 libraries, 3 package managers, 2 programming languages, 1 database, and 14 other tools that support software development like a testing tool, a continuous integration platform, and an API documentation generator.

Ecosystems are interconnected. The graph in Figure 1 shows two types of communities that occur in GitHub-hosted projects, those that are part of the largest connected component and those that are isolated from the largest connected component. The majority of project nodes, 10,484 or 57%, are involved in the largest connected component, indicating that many ecosystems are connected to each other across the projects in our Dependency Network. The next biggest connected component in the graph is only 65 nodes indicating that the ecosystems that are isolated are small and have not attracted public attention.

Figure 2 displays the interconnected part of the network, and the connections between the ecosystems are apparent. As an

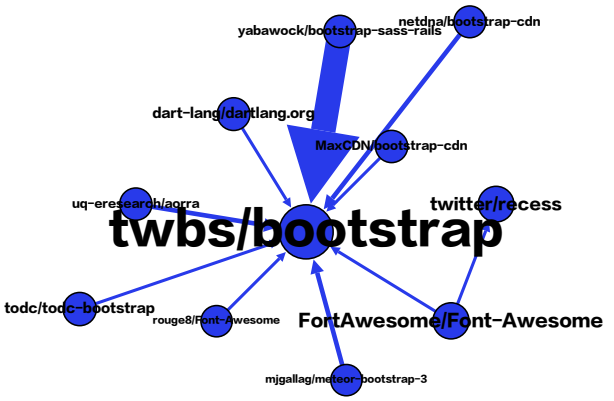

Fig. 3. twbs/bootstrap Ego Network. Portraying a sample star pattern in the network.

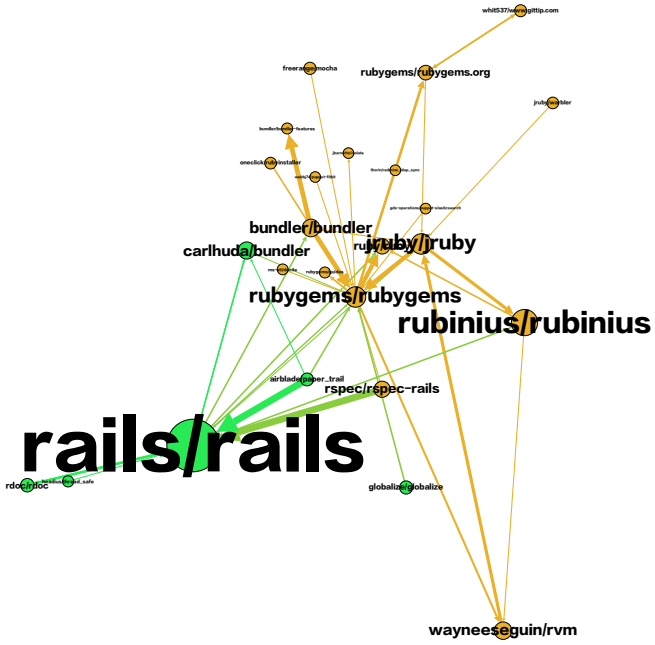

Fig. 4. rubygems/rubygems Ego Network. Portraying connections between ecosystems.

example, Figure 4 shows the rubygems/rubygems ego network, clearly depicting its connection to the rails/rails project. This is not surprising, since the rubygems project is a package management framework for the Ruby programming language and rails/rails is a web application framework written in Ruby. There is a direct connection between the rubygems/rubygems and rails/rails nodes. In addition, there are projects, like carlhuda/bundler and airblade/paper_trail, which connect the two projects.

IV. Social Behaviour of Project Owners/Contributors

RQ3: Do the project owners' and contributors' social behaviours align with the technical dependencies?

A. Research Method

To complement our investigation of technical dependencies and connectedness of projects in Github, we also sought to understand the social behaviour of project owners/contributors in relation to these ecosystems. We studied two of GitHub's social relationships, following users and starring projects. On GitHub, users can *follow* other users to receive notification on their activity and *star* a repository to bookmark it or indicate interest in the project. To understand how the social behaviour of project owners/contributors relates to the identified ecosystems, we examine the alignment between social and technical connections between the projects.

To answer our research question, we construct project-to-project networks based on the following and starring activity of the project owners and contributors. We ran correlations between the edge weights in the Dependency Network with the edge weights in each social network to determine if there is a relationship between the technical dependencies and the social connections. Pearson correlations were used since the data was normally distributed.

Project Owners. We constructed two networks using the following and starring relationships by considering the actions of the project owners. The *Owner Stars Network*, $G_{os} =< V, E >$, and the *Owner Follows Network*, $G_{of} =< V, E >$, are both undirected graphs whose set of vertices is all GitHub projects involved in at least one cross-reference. For the Owner Follows Network, there is an edge from nodes x_i to y_j if the owner of project x_i follows the owner of project y_j. There is an edge from x_i to y_j in the Owner Stars Network if an owner of any project in our dataset has starred both project x_i and project y_j.

To compare the social connections with the technical dependencies, we compare the edge weights of these two networks with the edge weights of the Dependency Network described earlier. The edge weights of these three networks represent the following:

- Dependency Network G_d: Number of technical dependencies, measured through reference coupling, between the two project nodes.
- Owner Follows Network G_{of}: 0 if neither project owner follows the other, 1 if one project owner follows the other project owner, and 2 if both project owners follow each other.
- Owner Stars Network G_{os}: Number of project owners who have starred both projects.

Project Contributors. We constructed two additional networks using these following and starring relationships by considering the actions of the project contributors (users who have made commits on the project or are members of the project). The *Contributor Stars Network*, $G_{cs} =< V, E >$, and the *Contributor Follows Network*, $G_{cf} =< V, E >$, are also undirected graphs whose set of vertices is all GitHub projects involved in at least one cross-reference. The Contributor Follows Network has an edge from nodes x_i to y_j if a contributor

TABLE II
PROJECT OWNERS: CORRELATIONS BETWEEN TECHNICAL DEPENDENCIES AND SOCIAL BEHAVIOUR.

	Pearson Correlation	p-value
Technical Dependencies and Following	0.91	<0.001
Technical Dependencies and Stars	0.79	<0.001

TABLE III
PROJECT CONTRIBUTORS: CORRELATIONS BETWEEN TECHNICAL DEPENDENCIES AND SOCIAL BEHAVIOUR.

	Pearson Correlation	p-value
Technical Dependencies and Following	0.0002	0.98
Technical Dependencies and Stars	0.001	0.88

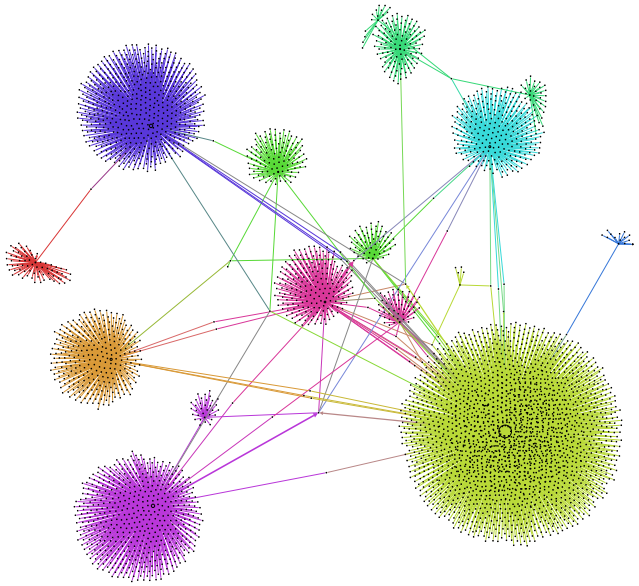

Fig. 5. The Owner Follows Network, G_{of}.

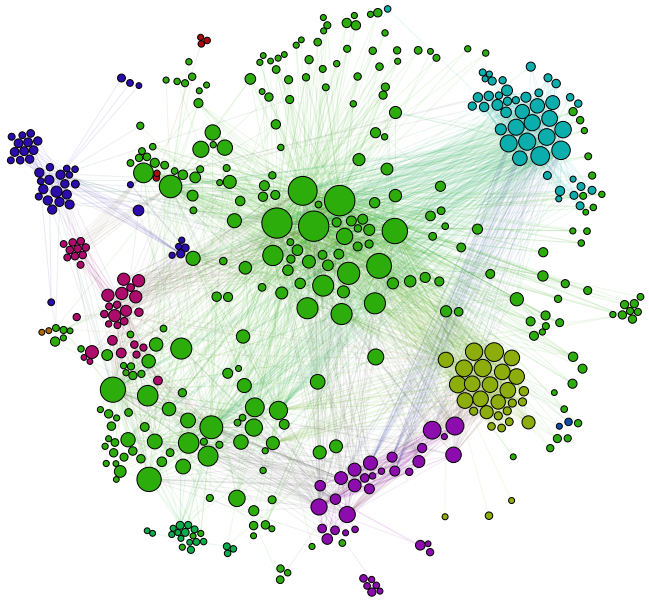

Fig. 6. The Contributor Follows Network, G_{cf}.

of project x_i follows a contributor of project y_j. The weight of each edge is the count of contributors with following relationships for the pair of projects. The Contributor Stars Network has an edge from x_i to y_j if a contributor to any project in our dataset has starred both project x_i and project y_j. The weight of each edge is the count of project contributors who have starred both projects.

B. Results

Project Owners. Table II shows strong, positive correlations between the technical dependencies and the social behaviour of the owners. Along with these strong correlations, Figure 5 shows a pronounced star pattern in the Owner Follows Network. This indicates that the project owners in an ecosystem tend to follow the owner of the central repository.

Project Contributors. As shown in Table III, the social behaviour of project contributors does not align with the technical dependencies. This indicates that, while the project owners seem to follow the right people and are aware of the right projects based on the technical dependencies that exist in the ecosystem, the social behaviour of project contributors is not aligned with project dependencies.

Figure 6 shows the Contributor Follows Network. As shown, the structure is quite different than the Dependency Network. Communities do not have one central project and the network is much more densely connected.

V. DISCUSSION

We developed a way to detect technical dependencies between projects by considering the cross-references made in comments on GitHub. We found that cross-references are commonly included in GitHub comments. By analyzing the content of these cross-references, we showed that they are a valid conceptualization of technical dependencies. We used this conceptualization of dependencies, called reference coupling, to identify ecosystems. To do this, we used a popular community detection algorithm [10] on the dependency network, which identifies clusters of nodes densely connected by technical dependencies. These detected communities represent software ecosystems.

Through analysis of the resulting ecosystems found in GitHub-hosted projects, we showed that the ecosystems are centered mostly around projects that support software development through developing frameworks and toolkits. The predominant structure of the ecosystems is a star where one central project is the hub of the ecosystem.

Our method detects technical dependencies that may not manifest themselves in source code by identifying issues, pull requests or commits that rely on another project, and, therefore, it can identify dependencies not identified by other methods. Our method is analogous to the logical coupling method that detects dependencies within a project proposed by Gall et al. [19] except at the ecosystem level. Where

logical coupling detects dependencies when artifacts have been worked on together, our method detects dependencies when issues, pull requests or commits have been worked in conjunction with another project (as evidenced through user-specified cross-references). Thus, the dependencies established through our method are those that are *logical*.

Limiting the detected dependencies to those that are logical is important when using those dependencies to identify ecosystems. Methods that detect technical dependencies between projects through analysis of code or configuration files may not be best suited for identifying software ecosystems. For example, when one project uses another project, it does not necessarily mean the two software projects are evolving together in the same environment, especially when the dependency is to an established, off-the-shelf software package. Thus, identifying all relationships that manifest in the source code or configuration files may result in dependencies that are not important for the identification of ecosystems.

Our method also allows for the identification of dependencies across all projects hosted on GitHub. Other methods that detect dependencies are limited to analyzing a given project or set of projects. Analyzing the dependencies of a popular project through its source code or configuration files to identify its ecosystem would not identify projects that rely on that project. We saw that most ecosystems across the GitHub-hosted projects are centered around one main project and many projects depend on that project without a reciprocal relationship. These relationships would be missed if only the dependencies of the main project were studied to identify its ecosystem. Since the ecosystems are not always well-defined, it would be impossible to know which other projects to consider for analysis. Thus, our method is better suited to identifying ecosystems since it is not limited in the number of projects it can analyze.

GitHub's built-in social features and transparency may foster these comments about project interconnections. Future work should investigate if similar comments are made on other software development environments that would allow this method to be extended beyond GitHub. Future work should also compare the dependencies established via reference coupling to those that manifest themselves in the source code to provide greater insight into the types of dependencies that manifest using this method.

A. A Research Agenda

The ability to easily identify technical dependencies between a large set of projects opens the door for many interesting avenues of research.

Socio-technical analysis. Studies that have attempted to study how communication aligns with dependencies across projects have been limited to studying well-defined ecosystems where dependency information is publicized in some way. For example, dependencies can be made available through a project's configuration files, dependency manager files or through publicly available dependency specifications [1], [6], [7], [8], [9], [20]. Our method allows the identification of technical dependencies more broadly across projects and opens the door to continuing the study of socio-technical alignment across a larger set of projects and their stakeholders.

In this study, we found that when dependencies exist between a pair of projects, the project owners tend to be following the owner of the other project. Conway was the first to describe the possibility of an alignment between social connections and technical dependencies in software engineering projects, commonly referred to as Conway's law [21]. The transparent nature of GitHub could encourage technical connections between projects by providing an awareness of activity across projects. An interesting future research question is understanding how and when these technical dependencies and social connections came to exist. Did the social connections exist first and result in a technical dependency or did the technical dependency exist first and result in a social connection? If the social connections existed first, what was the driver behind the creation of the technical dependency? Perhaps, the awareness of the other project, enabled through GitHub's notifications, was enough to spur a technical dependency indicating that GitHub's transparency is changing the landscape of OSS projects. These research questions could be investigated in future research.

While the project owners' social behaviours (following users and starring projects) aligned with the technical dependencies in our study, we did not witness such an alignment for all project contributors. The follower network of project contributors showed that there were no clear central projects and communities were densely connected. This is in contrast to the technical dependency network. These results align with recent research that found that the reasons behind following others extends beyond project coordination needs [22]. Future work should investigate the usefulness of following others for coordination purposes.

It is also worth studying in more detail the coordination needs of developers on OSS projects. Perhaps the mere existence of a technical dependency does not imply a coordination need, especially given the transparent environment of GitHub. Our previous work [23] begun this investigation, but coordination needs at the ecosystem level are also worthy of investigation.

Ecosystem emergence and evolution. The most prominent nodes in Figure 2 are not always the most popular projects on GitHub when considering the number of stars each project has. In fact, the two projects with the most stars, angular/angular.js and twbs/bootstrap, have significantly smaller ecosystem size and lower degree than other projects. Future work can investigate how and why ecosystems emerge and why some projects become popular without growing a large ecosystem. Such a study could include a temporal analysis of the composition of the ecosystem and density of connections together with a temporal analysis of project history information such as number of contributors, forks, stars, etc. It would also be worth triangulating results with other information on important

project events now commonly available through blogs and wikis. Such a study on the evolution of ecosystems can be a first step in understanding when and why projects accumulate an ecosystem.

Ecosystem size and strength of connections and project success. On many open source projects, volunteers are crucial to project success as they rely on volunteers to submit new features and fix bugs. As a project accumulates more projects in its ecosystem, it is also likely to increase its contributions as developers on dependent projects will be more likely to fix bugs that they encounter through their dependency. Future research could investigate this relationship to identify if the size of a project's ecosystem is a good predictor of various project health and success metrics like the number of contributions it receives or the number of forks it has.

Automatic detection of ecosystems. Another avenue for future research is creating tools to support developers at the ecosystem level. It is important for developers to know who they need to coordinate with across the ecosystem and to understand how their tasks fit into the big picture. A tool could be developed that automatically identifies technical dependencies across projects and provides a visualization of the ecosystem. Such a tool could increase awareness of coordination needs that extend outside project boundaries and help developers gain a better view of the ecosystem surrounding their project.

B. Threats to Validity

One threat stems from our selection of the GHTorrent dataset, which may not be a full copy of all GitHub data [12]. Nevertheless, it is a best-effort approach that has been widely accepted in the research community as evidenced by its inclusion as the dataset for the MSR 2013 Mining Challenge [24] and the many recent papers that utilize its data in their analysis.

Another threat is that our analysis was limited to only projects hosted on GitHub. We do not generalize our results to other coding hosting environments. Future work can investigate whether similar cross-reference comments that indicate technical dependencies are found in other coding hosting environments.

Our manual exploration of cross reference comments illustrates a variety of types of technical dependencies, but these results also can not be generalized. While we achieved saturation in our results, our results could be impacted by selection bias. To mitigate this, we ensured an equal number of comments for each source (commit, issue, pull request) were included in our sample. Further, for each repository, the manual analysis was performed by only one person introducing a possible risk of unreliable results. However, the types of dependencies identified seem reasonable for any software project. Future work can continue this investigation by examining the content of cross-reference comments across a wide range of projects and code hosting environments.

VI. Related Work

Much work has been done studying the technical, business and social dimensions of software ecosystems [25]. Studies of OSS software ecosystems have mostly focused on the analysis of a well-defined OSS ecosystem like Eclipse [4], [26], Ruby on Rails [20], [27] or Apache [6], [8]. Instead, we introduce a way to identify unknown ecosystems by using community detection methods.

Several studies [28], [29] have used community detection algorithms to detect communities across GitHub projects, but they have focused on relationships between developers rather than technical dependencies between projects. Thung et al. [30] constructed project-to-project networks for GitHub-hosted projects, but edges between projects in their network represent a single developer contributing to both projects. This method can not be used to detect dependencies between projects since developers can often work on multiple independent projects and, thus, sharing developers is not an indication of a technical dependency. We use technical dependencies for community detection since the structure of an ecosystem is defined by its technical dependencies [2].

Analysis of a project's source code is a common technique to identify technical dependencies within a project (intra-project). However, these techniques do not scale up to identify dependencies between projects (inter-project). Lungu et al. [2] describe several methods for extracting inter-project dependencies by considering external method and class calls in a project's source code. However, when investigating a large number of projects, obtaining the source code for every project is not always feasible. Collecting source code data across an entire versioning system would require multiple TBs of data and more than a year in processing time [5]. Ossher et al. [3] introduced a technique that analyzes import statements in Java source code to resolve inter-project dependencies. Businge and Serebrenik [4] employ a similar technique in their study of the Eclipse ecosystem. However, this technique still requires obtaining a large amount of source code and, therefore, requires a large amount of memory. These techniques, therefore, are limited in the number of projects that can be studied.

Previous studies have proposed ways to identify technical dependencies without relying on analysis of source code. One method is to identify technical dependencies by examining declared dependencies from a project's configuration files or its dependency management tool like Maven [1], [6], [7], [8], [9]. However, not all projects declare dependencies in configuration files or employ a dependency manager, and, even for those that do, the data can be missing. Bavota et al. [8] found that this information was missing in 37% of releases in a study of the Apache project. Syeed et al. [20] extracted metadata on inter-project dependencies from the published specifications at rubygems.org in their study of the Ruby on Rails ecosystem. However, the specified dependencies may be out of date and the approach is specific to only projects that publish dependency specifications.

Our approach, which does not rely on analyzing source code, takes advantage of the cross-references that can be made in comments on GitHub. These cross-references are user-specified links between a pair of projects. They are made in comments on pull requests, issues, and commits as developers

coordinate and manage their work dependencies.

VII. Conclusion

In this paper, we proposed a new method for detecting technical dependencies between projects, called reference coupling, which utilizes user-specified cross-references between projects. We used this reference coupling method to identify ecosystems in GitHub-hosted projects by using an existing community detection algorithm to identify densely connected clusters of projects. Through an analysis of the identified ecosystems, we find that most ecosystems are centered around a single project. While small, unpopular ecosystems remain isolated, most ecosystems are interconnected. The isolated ecosystems tend to contain projects owned by the same GitHub user or organization. The popular ecosystems are mostly centered around tools that support software development.

Our reference coupling method opens the door for future research in software ecosystems including studying the socio-technical relationships, evolution, health and success of ecosystems.

Acknowledgment

This work was partly funded by NSERC Canada.

References

[1] M. F. Lungu, "Reverse engineering software ecosystems," Ph.D. dissertation, University of Lugano, 2009.

[2] M. Lungu, R. Robbes, and M. Lanza, "Recovering inter-project dependencies in software ecosystems," in *Proceedings of the International Conference on Automated Software Engineering*. ACM, 2010, pp. 309–312.

[3] J. Ossher, S. Bajracharya, and C. Lopes, "Automated dependency resolution for open source software," in *Proceedings of 7th Working Conference on Mining Software Repositories*. IEEE, 2010, pp. 130–140.

[4] J. Businge, A. Serebrenik, and M. van den Brand, "Survival of eclipse third-party plug-ins," in *Proceedings of 28th International Conference on Software Maintenance*. IEEE, 2012, pp. 368–377.

[5] A. Mockus, "Amassing and indexing a large sample of version control systems: Towards the census of public source code history," in *Proceedings of 6th Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 11–20.

[6] F. W. Santana and C. M. L. Werner, "Towards the analysis of software projects dependencies: An exploratory visual study of software ecosystems." in *Proceedings of International Workshop on Software Ecosystems*. Citeseer, 2013, pp. 7–18.

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

[8] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the apache community upgrades dependencies: an evolutionary study," *Empirical Software Engineering*, pp. 1–43, 2014.

[9] D. M. German, J. M. Gonzalez-Barahona, and G. Robles, "A model to understand the building and running inter-dependencies of software," in *Proceedings of 14th Working Conference on Reverse Engineering*. IEEE, 2007, pp. 140–149.

[10] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, 2008.

[11] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A project memory for software development," *Transactions on Software Engineering*, vol. 31, no. 6, pp. 446–465, 2005.

[12] G. Gousios and D. Spinellis, "Ghtorrent: Github's data from a firehose," in *Proceedings of the 9th Working Conference on Mining Software Repositories*. IEEE, 2012, pp. 12–21.

[13] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 92–101.

[14] J. Corbin and A. Strauss, *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage, 2008.

[15] M. E. Newman, "Modularity and community structure in networks," *Proceedings of the National Academy of Sciences*, vol. 103, no. 23, pp. 8577–8582, 2006.

[16] M. Bastian, S. Heymann, M. Jacomy *et al.*, "Gephi: an open source software for exploring and manipulating networks." *Proceedings of International AAAI Conference on Web and Social Media*, vol. 8, pp. 361–362, 2009.

[17] M. Molloy and B. Reed, "Critical subgraphs of a random graph," *The Electronic Journal of Combinatorics*, vol. 6, no. R35, p. 2, 1999.

[18] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *Journal of the ACM*, vol. 46, no. 5, p. 604632, 1999.

[19] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proceedings of International Conference on Software Maintenance*. IEEE, 1998, pp. 190–198.

[20] M. Syeed, K. M. Hansen, I. Hammouda, and K. Manikas, "Socio-technical congruence in the ruby ecosystem," in *Proceedings of The International Symposium on Open Collaboration*. ACM, 2014, p. 2.

[21] M. E. Conway, "How do committees invent," *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[22] K. Blincoe and D. Damian, "Implicit coordination: A case study of the rails oss project," in *Proceedings of International Conference on Open Source Systems*, 2015, to appear.

[23] K. Blincoe, G. Valetto, and D. Damian, "Do all task dependencies require coordination? the role of task properties in identifying critical coordination needs in software projects," in *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 2013, pp. 213–223.

[24] G. Gousios, "The ghtorent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 233–236.

[25] C. Werner and S. Jansen, "A systematic mapping study on software ecosystems from a three-dimensional perspective," *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*, pp. 59–81, 2013.

[26] D. Dhungana, I. Groher, E. Schludermann, and S. Biffl, "Software ecosystems vs. natural ecosystems: learning from the ingenious mind of nature," in *Proceedings of the 4th European Conference on Software Architecture: Companion Volume*. ACM, 2010, pp. 96–102.

[27] J. Kabbedijk and S. Jansen, "Steering insight: An exploration of the ruby software ecosystem," in *Software Business*. Springer, 2011, pp. 44–55.

[28] S. Syed and S. Jansen, "On clusters in open source ecosystems." in *Proceedings of International Workshop on Software Ecosystems*. Citeseer, 2013, pp. 19–32.

[29] Y. Yu, G. Yin, H. Wang, and T. Wang, "Exploring the patterns of social behavior in github," in *Proceedings of the 1st International Workshop on Crowd-based Software Development Methods and Technologies*. ACM, 2014, pp. 31–36.

[30] F. Thung, T. F. Bissyandé, D. Lo, and L. Jiang, "Network structure of social coding in github," in *Proceedings of 17th European Conference on Software Maintenance and Reengineering*. IEEE, 2013, pp. 323–326.