Characterizing Project Evolution on a Social Coding Platform

Koji Toda*, Haruaki Tamada†, Masahide Nakamura‡¶, Kenichi Matsumoto§,

* Department of Computer Science and Engineering, Fukuoka Institute of Technology, Fukuoka, Japan.
† Faculty of Information Science and Engineering, Kyoto Sangyo University, Kyoto, Japan.
‡ Graduate School of System Informatics, Kobe University, Hyogo Japan.
§ Graduate School of Science and Technology, Nara Institute of Science and Technology, Nara, Japan.
¶ Riken, AIP, Tokyo, Japan

***Abstract*—Social coding platforms (SCPs) have realized *spontaneous software evolution*, where new source code and ideas are spontaneously proposed by altruistic developers. Although there are many projects operated by active communities performing spontaneous evolution, it is yet unclear that how such successful projects and communities have been formed and governed. In this paper, we propose a method that can investigate the history of every project in the SCP. Introducing the concept of *project as a city*, we consider every project in the SCP as a city, where a government and citizens develop a city through collaborative activities. We then identify essential attributes that characterize a state of a city. For each attribute, we develop metrics that quantify the state $S(p,t)$ of a project p at time t. An experimental evaluation investigating GitHub projects of famous code editors shows that the proposed metrics well visualize the history of the projects from essential perspectives of a city.**

***Index Terms*—social coding, software evolution, project measurement, software metrics, governance, smart city**

I. Introduction

Social coding [1] is a modern approach to software development, which places an emphasis on formal and informal collaboration. Due to the emerging *social coding platforms (SCP, for short)* such as GitHub [2] and Bitbucket [3], the social coding is now widely adopted not only in open-source projects, but in commercial software development.

In the SCP, software is developed and maintained collaboratively by a *community* of developers and users. The SCP provides a space for the people to create a project, have communication, play and build whatever they want. Moreover, every developer can hack and modify the code as he/she likes by *forking* the project. Then, the developer can *request* the project owner to merge his/her modification to the original code. If the owner finds it useful, the owner *commits the merge*, which creates a new version of the software. In GitHub, the above three activities are implemented as commands `fork`, `pull request`, and `merge`, respectively.

From the perspective of *software evolution* [4], the above process of code modification is quite unique. Traditionally, when new features and/or better solutions are requested by users or customers, a project manager designs tasks, and directs developers to work in a top-down manner. Then, the developers modify the code according to the requirement. However, in the social coding, the code modification can be *proposed* by spontaneous and altruistic developers in the community. Thus, the software evolution can be triggered in a bottom-up way. We call this new type of software evolution *spontaneous software evolution* [5].

The spontaneous software evolution by the social coding is a powerful means to achieve fast, flexible, and sustainable evolution of the software. Also, it often derives innovative ideas that a project owner has never imagined, realizing advanced features and products. For example, `microsoft/vscode` [6] is a GitHub project of a powerful code editor, where a lot of people have discussion, developers actively present codes, and many innovative features have been added. Thus, we can see that `microsoft/vscode` is a successful project in the context of spontaneous software evolution.

However, we do not know how such a successful project has been managed, or how such an active community has been formed and governed. To clarify these issues, we need to investigate the *history*, starting when the project was created, and leading to the current shape. Fortunately, since the SCP accumulates various *event logs*, we can look back how the project was for any past time. More specifically, for a given project p and timestamp t, let $S(p,t)$ denote a *state* of p at t. Observing $S(p,t)$ with varying t derives a sequence of states, which characterizes how p has been evolved. The challenge here is how to define $S(p,t)$ so that it well characterizes the spontaneous software evolution within p.

The goal of this paper is to propose a method that can cope with the following research questions:

RQ1: How and by what can we define the state $S(p,t)$?

RQ2: How has the state of a successful project changed?

To address RQ1, we first introduce the concept of *project as a city*. Borrowing the idea of smart city modeling [7], we consider every project in the SCP as a city, where a government and citizens develop a city through collaborative activities. We investigate a mapping that associates constructs and activities of a city with entities and actions of the SCP. We then identify essential attributes that characterize a state of a city: population, size, urban problem, problem solving ability, capability of government. Finally, we define metrics N-STAR, N-MGCL, N-RIS, R-RIS, LT-CIS, R-MGPR, LT-PR,

978-1-7281-1651-8/19/$31.00 ©2019 IEEE
SNPD 2019, July 8-11, 2019, Toyama, Japan

and R-PRLP. Each metric quantifies a corresponding attribute of a project p at timestamp t, which can be calculated from data accumulated in the SCP. Thus, the values of the metrics characterize the state $S(p,t)$, reasonably.

To address RQ2, we conduct an experiment where we observe the evolution of GitHub projects of famous code editors: `atom/atom` [8], `adobe/brackets` [9], and `microsoft/vscode` [6]. The result showed that the proposed metrics well visualized the history of the three projects from essential perspectives of a city.

II. Preliminaries

A. GitHub: Social Coding Platform

GitHub [2] is a cloud-based platform for software development. GitHub uses Git for version control of codes and documents, and also provides many features of social networking. GitHub is now one of the biggest social coding platforms, acquiring 28 million users over the world.

In GitHub, source code and documents of a software project are stored in a *repository*, which is maintained by one or several *core members*. Basically, only the core members are allowed to modify the code within the repository. A *fork* creates a copy of the repository, which allows any other developer to hack and modify the code without affecting the original. If the developer would like to share the modifications, he/she can send a *pull request*. If, after careful review, the core members agree to accept the modifications, they *merge* the modifications with the original repository. If they do not agree, the pull request is rejected and closed.

If a GitHub user is interested in a particular project, he/she can *star* the repository. The number of stars reflects the popularity of the project. If a user finds bugs, needs enhancements, or has any other requests, he/she can create an *issue* in the repository. For the issue, another user may send a *comment* to solve, or a developer may modify the code and sends a pull request. An issue is *closed* when the problem is solved.

B. Spontaneous Software Evolution

Software evolution deals with the process by which software is modified to adapt to the changing environment and/or new requirements [4]. In the conventional software project, a project manager controls the process for the evolution, while developers work as directed by the manager. The SCP like GitHub changes the process. Developers can propose modifications in the form of pull requests, while a manager reviews the pull requests and decides to merge or not.

We call this new type of evolution *spontaneous software evolution* [5]. We are currently studying how to achieve the spontaneous software evolution in projects, including commercial software development. Major challenges include:

- How should a project be governed and managed?
- How can social overheads be reduced?
- How can the motivation of communities be raised?

The goal of this paper, which investigates the history and states of projects in the SCP, is motivated from the first challenge.

III. Concept of Project as a City

A. Key Idea

To address the research question RQ1 in Section I, we here introduce a concept of *project as a city*. As seen in Section II-A, every project in a SCP is governed by core members. Developers and users, who are interested in the project, form a *community*. Everybody can see the code and event log, and has a chance to ask a request or propose code modifications. Thus, there is no static hierarchy in the community.

This observation of the SCP reminds us of a *smart city* [7]. In a smart city, various kinds of data about a city is collected and opened to citizens. Based on the information, the government and the citizens can collaboratively work to enhance performance of the city, as well as well-being of the citizens. This is quite similar to social coding in the SCP.

In [7], a *state* of a city is defined by a vector $[s_1 : v_1, s_2 : v_2, ..., s_m : v_m]$, where s_i is a state variable representing an interesting attribute, and v_i is its value. Considering a city as a state machine, a smart city service is defined as a sequence of actions that directs the city from the current (as-is) state to an acceptable (to-be) state. We take the same analogy to define the state $S(p,t)$, by associating elements in the SCP with those in a city.

B. Associating Elements

We define a mapping from basic elements of SCP to some constructs of a city. For better understanding, we take GitHub as a representative SCP in the following explanation.

- A project is associated with a *city*.
- The code of the project is associated with *function* of the city. Thus, merging new code lines corresponds to developing new function of the city.
- The core members are associated with the *government*, since they have authority to change the city.
- Users and developers who are interested in the project correspond to *citizens*. For a practical purpose, we regard a user who gave a star to the project as a citizen.
- An issue is associated with a *problem in (or request for)* the city, which is raised by a citizen. Creating (or closing) an issue corresponds to posing (or solving, respectively) the problem.
- A pull request is associated with a *solution proposal* for a problem. Creating a pull request based on an issue corresponds to a situation that a citizen sends a proposal of solving a problem to the government. Merging a pull request corresponds to a situation that the government accepts the proposal and applies the solution to the city.

C. Essential Attributes Characterizing Evolution

Once we consider every project in the SCP as a city, it would be easier to consider what are the essential attributes to define the state. When we discuss the history and the evolution of a city, the following attributes are important, although they do not cover everything of the city:

1) **Population:** It is an indicator characterizing how much the city attracts citizens.

2) **Size:** It is an indicator characterizing how much the city is developed to be smarter and more convenient.
3) **Problems:** It is an indicator that reflects how much citizens have difficulty in living in the city.
4) **Ability of Problem Solving:** It reflects the performance of the city resolving the problem.
5) **Capability of Government:** It reflects the performance of government to process proposals from citizens and solve the problems.

For a given project p in SCP and timestamp t, we consider a metric $m_i(p,t)$ that quantitatively measures i-th essential attribute. Then, we define the state as a vector:

$$S(p,t) = [m_1(p,t), m_2(p,t), ..., m_n(p,t)]$$

where concrete definition of m_i's is given in the next section.

IV. Developing Metrics for Characterizing Evolution of Project as a City

A. Population

The population is the most basic and important factor of the city. This is because increasing population promotes activities of the city, opinions from new citizens, and workloads to solve the problems. The population surely influences other attributes of the city, such as the size of the city, the quantity and the divergence of problems, performance of problem solving, and the capability of the government.

When a software development project is associated with a city, the population should be the number of all *participants* of the projects, including core members, non-core developers, and interested users. However, it is difficult to count the exact number of participants since not all the participants directly give feedback or code contribution.

We adopt the *number of stars* to characterize the population of the project. This is because every user can add a star, even if he/she does not give direct contribution but is at least interested in the project. Therefore, the number of stars is a reasonable metric to quantify the population of the project.

We define the metric N-STAR(p,t) as the number of stars that a project p has earned by the time t.

Note, however, that the number of stars in GitHub is a bit different from the real population of a city, strictly speaking. In GitHub, once a user attaches a star, he/she would not detach the star even if the project becomes out of interests. In that sense, the population of a project should be evaluated by observing the growth rate of N-STAR(p,t) as well as its absolute value.

B. Size

The size of a city is also a key factor as important as the population, representing how much the city is developed. When a software project is associated with a city, the degree of city development can be associated with the amount of source code that has been added or modified within the repository.

The amount of source code can be simply measured by LoC (Lines of Code). In GitHub, when a pull request is merged, modifications of the source code are recorded as a `diff` format, from which we can calculate the number of added/deleted lines of code. By accumulating the lines of code for all merged pull requests, we can represent the size of how much the project has been developed.

We here define the metric *the number of merged code lines*. Let p be a project, t be timestamp, r be a pull request. Let $\mathrm{lmod}(r)$ be the lines of code that are modified within r. Let $\mathrm{cpr}(p,t)$ be a set of pull requests in p closed before t, and $\mathrm{mpr}(p,t) \subseteq \mathrm{cpr}(p,t)$ be a set of pull requests merged before t. Then, the metric *the number of merged code lines*, N-MGCL(p,t), is defined by

$$\text{N-MGCL}(p,t) = \sum_{r \in \mathrm{mpr}(p,t)} \mathrm{lmod}(r) \tag{1}$$

N-MGCL(p,t) represents the lines of code that have been merged within p before t, characterizing the size of p at t.

C. Problem

The problem within a city is an obvious obstacle for future evolution of the city. Various problems may arise as the population and the size of the city grow. It is not surprising to see that the number of problems is increasing when no action is taken. Preferably, any problem should be resolved as soon as it occurs. If the problems remain unsolved, the citizens are disappointed to the city and the government, then they would leave. Therefore, government need to recognize remaining issues and make an action plan to solve them.

Within the concept of the project as a city, a problem is associated with an issue (a bug report, a request of a new feature, etc.). The issue is closed when the problem is resolved. Therefore, we measure *remaining* issues to characterize the problems that the city currently has. For a project p and timestamp t, let N-AIS(p,t) be the number of all issues that have been created before t within p. Let N-CIS(p,t) be the number of closed issues that have been created before t within p. Then, we define the *number of remaining issues* N-RIS(p,t), and *the ratio of remaining issues* R-RIS(p,t):

$$\text{N-RIS}(p,t) = \text{N-AIS}(p,t) - \text{N-CIS}(p,t) \tag{2}$$

$$\text{R-RIS}(p,t) = \frac{\text{N-RIS}(p,t)}{\text{N-AIS}(p,t)} \tag{3}$$

D. Ability of Problem Solving

The ability of problem solving is an important factor for the future development of the city. If the government or the citizens cannot solve problems quickly, more problems will be accumulated, which declines citizen's satisfaction. From this reason, the city should have an ability to solve the problems as quickly as possible.

To quantify the ability of the problem solving, we adopt the time taken to solve an issue. If it takes very long time to close an issue, or an issue remains unsolved, we can see that the ability is low. Thus, we characterize the ability by the *lifetime* of the issue. For a given issue x, let $\mathrm{since}(x)$ be the time when x is created, and let $\mathrm{until}(x)$ be the time when x is closed.

Then, the lifetime of issue x, denoted by $lt(x)$, is defined by a time difference:

$$\mathrm{lt}(x) = \mathrm{until}(x) - \mathrm{since}(x)$$

Now let $\mathrm{CIS}(p,t)$ be a set of issues within p that have been closed before t. Then, we define the *lifetime of closed issues*, LT-CIS(p,t) as follows:

$$\text{LT-CIS}(p,t) = \frac{\sum_{x \in \mathrm{CIS}(p,t)} \mathrm{lt}(x)}{\text{N-CIS}(p,t)} \tag{4}$$

The greater value of LT-CIS(x) indicates that the problems tend to be left unresolved for a long time. For convenience, we represent the time difference $\mathrm{lt}(x)$ and LT-CIS(p,t) by a unit of day.

E. Capability of Government

An important role of the government is to review every proposal from citizens (i.e., pull request) and to decide if the proposal should be accepted or rejected. To evaluate this process, we consider the following three attributes: acceptability, agility, and soundness.

1) Acceptability: The acceptability is the degree of how much the government accepts proposals from citizens. Therefore, the acceptability is related to openness or the degree of democracy of the city. If most of the proposals from the citizens are not accepted, the citizens are disappointed to the government, and they would leave the the city.

Since the proposal is associated with a pull request, we measure the ratio of merged pull requests among all pull requests to characterize the acceptability of government. If the ratio is too low, developers as citizens may not be motivated to propose new pull requests.

We define the *ratio of merged pull requests*, R-MGPR(p,t) as follows:

$$\text{R-MGPR}(p,t) = \frac{|\mathrm{mpr}(p,t)|}{|\mathrm{cpr}(p,t)|} \tag{5}$$

where $\mathrm{mpr}(p,t)$ and $\mathrm{cpr}(p,t)$ are those defined in Section IV-B.

2) Agility: The agility represents the degree of how quickly the government determines the acceptance (or rejection) of the proposal. Therefore, the agility is related to the speed of evolution of the city. If the government spends long time to decide the acceptance, the evolution of the city becomes slow down or is stopped.

To quantify the agility of the government, we adopt the lifetime of pull requests. For a pull request r, let $\mathrm{since}(r)$ be the time when r is created, and let $\mathrm{until}(r)$ be the time when x is closed. Then, the *lifetime of pull request* r, denoted by $\mathrm{lt}(r)$, is defined by a time difference:

$$\mathrm{lt}(r) = \mathrm{until}(r) - \mathrm{since}(r)$$

Then, for a project p and timestamp t, we define the *lifetime of pull requests*, LT-PR(p,t) as follows:

$$\text{LT-PR}(p,t) = \frac{\sum_{r \in \mathrm{cpr}(p,t)} \mathrm{lt}(r)}{|\mathrm{cpr}(p,t)|} \tag{6}$$

TABLE I
METRICS OF THE CHOSEN PROJECTS

Projects	Stars	Commits	Issues	PRs	Created at
`vscode`	73,554	48,840	67,825	5,148	2015-09-03
`atom`	48,654	36,581	14,648	4,393	2012-01-20
`brackets`	29,756	17,739	9,276	5,474	2011-12-07

LT-PR(p,t) represents the average life time of all pull requests that have been closed before t within p. For convenience, we represent the time difference $\mathrm{lt}(r)$ and LT-PR(p,t) by a unit of day.

3) Soundness: The soundness represents the degree of how carefully the government considers the quality of the proposal. For the purpose of quality assurance, most SCP recommends a rule that *every pull request must be reviewed by one or more third-person developers*. If this rule is obeyed, the number of developers involved in a pull request must be two or more. However, this rule is sometimes ignored by core members or specific members of a project. If such cases occur frequently, the soundness of the review process is declined, and thus the quality of pull requests cannot be assured.

To quantify the soundness of the government, we measure the ratio of pull requests lacking enough participants. For a pull request r, let $\mathrm{nop}(r)$ be the number of people participating in r. For a project p and timestamp t, let $\mathrm{mpr\_lp}(p,t)$ be a set of pull requests defined by:

$$\mathrm{mpr\_lp}(p,t) = \{r | r \in \mathrm{mpr}(p,t) \wedge \mathrm{nop}(r) < 2\}$$

$\mathrm{mpr\_lp}(p,t)$ represents a set of pull requests violating the above rule. That is, only one developer is involved in the pull request, and the pull request is not reviewed by anyone else.

Then, we define *the ratio of pull requests lacking participants*, R-PRLP(p,t) as follows:

$$\text{R-PRLP}(p,t) = \frac{|\mathrm{mpr\_lp}(p,t)|}{|\mathrm{mpr}(p,t)|} \tag{7}$$

V. Experimental Evaluation

A. Objectives and Settings

To address RQ2 in Section I (i.e., to see the states of successful projects), we have conducted experimental evaluation applying the proposed metrics to major GitHub projects. We visualize the evolution of each project as the time-series observation of the proposed metrics from each viewpoint of essentials of the city (see Section III-C). We chose projects of well-known code editors, `vscode` (`microsoft/vscode`), `atom` (`atom/atom`), and `brackets` (`adobe/brackets`) as targets. Those three editors are large-scale software and have matured community.

Table I shows the code editors chosen, and their basic project metrics. The columns of the table I represent the project name, the total numbers of stars, commits, issues, and pull requests, and the date when the project is created at[1].

[1]The data is as of May 10, 2019

B. Data Collection

For calculating the proposed metrics, we have extracted various data from the three projects using GitHub GraphQL API [10], [11]. The data items collected for each element are:

- star
  - the date of starred (starredAt)
- issue
  - is closed
  - the date of open
  - the date of close
- pull request
  - is closed
  - is merged
  - the date of open
  - the date of close
  - the number of added lines
  - the number of deleted lines
  - the number of participants
  - the number of comments

Note that the date for each element is required for the time-series analysis. We show an example that extracts the date of starred from GitHub through the GraphQL API. Figure 1 shows a GraphQL script that queries the date of starred of `atom`. When this script is sent to the API, the response shown in Figure 2 is returned. By the limitation of the API, we can extract only 100 entries at a time, so we repeat to send the request to multiple *pages*. The `after` entry at the line 4 of Fig 1 specifies the start point of the pagination requested, which is referred as the value of `cursor` in the response.

```
1: query stargazers{
2:   repository(owner: "atom", name: "atom") {
3:     stargazers(first: 100,
4:          after: "Y3Vyc29yOnYyOpIAzgFL8DE="){
5:       totalCount
6:       edges{
7:         cursor, starredAt,
8:       }
9:     },
10:   }
11: }
```

Fig. 1. An example GraphQL code for getting stargazed dates

```
{
  "data": {
    "repository": {
      "stargazers": {
        "totalCount": 48654,
        "edges": [
          // .... omit
          {
           "cursor": "Y3Vyc29yOnYyOpIAzgFL8uY=",
           "starredAt": "2014-05-06T15:36:09Z"
          }]
      }
    }
  }
}
```

Fig. 2. A response JSON by posting GraphQL script shown in Figure 1

C. Visualizing Evolution

Based on the data collected, we calculate the proposed metric $m(p,t)$ for each project p, varying the timestamp t in a daily basis since p is created until now.

Figure 3 shows the results of the analysis. Each graph in Figure 3 contains three lines (red dotted lines, green dashed line, and blue solid line), where the horizontal axis represents the date. The three corresponds to the calculated metrics of `atom`, `brackets`, and `vscode`, respectively. We observe the results from the perspective of the five essential attributes.

1) Population: N-STAR(p,t) in Figure 3 characterizes the evolution of the population of three projects. The vertical axis plots the number of stars. From the graph, we can see that the population of the three projects grow well. Especially, the rapid growth of `vscode` is notable. Preferably, the graph of N-STARshould continuously increase; however, the gradient may become gentle as the project is matured.

In the graph, we can find some points that the population suddenly grows, which are related to special events of the projects. For example, the populations of `atom` and `vscode` suddenly grew at the middle of 2014, and late of 2015, respectively. We investigated the reason for the sudden growth. The reason was due to the first release of the projects, on Feb 27th, 2014 of `atom`[2], and on Nov 18th, 2015 of `vscode`[3].

2) Size: N-MGCL(p,t) in Figure 3 characterizes the evolution of the size of the projects, where the vertical axis represents the number of merged code lines. From the graph, we can see that the size of every editor grows well by the development the new features. Although `vscode` started later than other two editors, the size is actively evolving to catch up with the two. Of course, the size would be relate to the population of the projects mentioned in Section V-C1. The sudden growth of N-MGCL(p,t) corresponds to large-scale modifications, and it is often observed in development of new features. When a flat period appears in the graph, it means the project is inactive. This implies that there is nothing to change the project, or that the governance of the evolution falls into a bad shape.

3) Problems: N-RIS(p,t) and R-RIS(p,t) in Figure 3 characterizes the evolution of the problems in the projects. The vertical axes of N-RIS(p,t), and R-RIS(p,t) are the number of remaining issues, and the ratio of remaining issues, respectively. In N-RIS(p,t), we can see that the evolution of each editor represents its own characteristics. The number of issues of `atom` increases gradually at the beginning of the

[2]https://github.com/atom/atom/releases/tag/v0.56.0
[3]https://github.com/microsoft/vscode/releases/tag/0.10.1

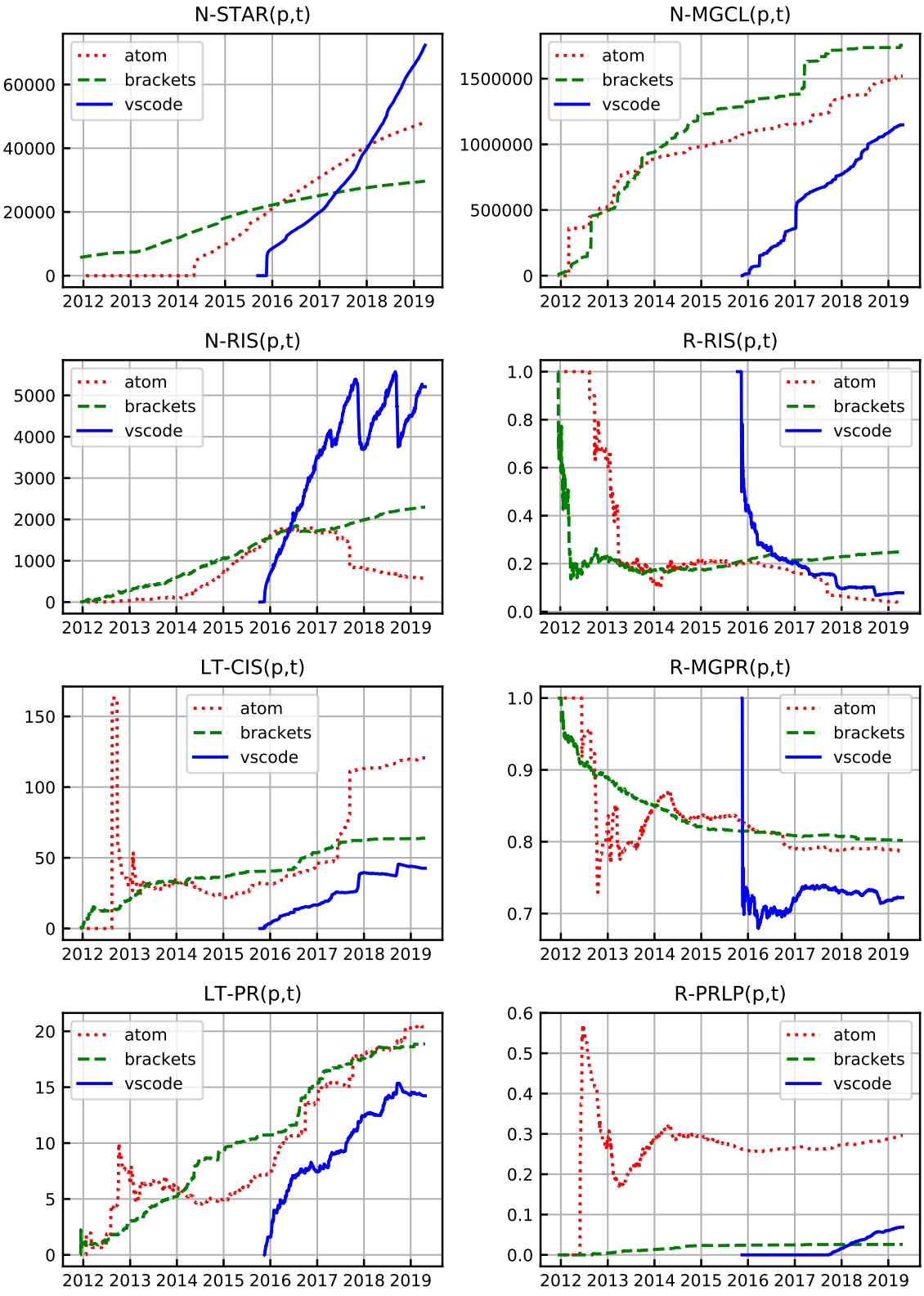

Fig. 3. Evolution of all metrics for the code editors

project until 2016. After that, it becomes flat and decreases after late 2017. The shape of the graph would tell that the problems in the project had been addressed continuously, and that the problems are decreased as the project is matured.

In the `brackets`, the issues had been stacked little by little, and the graph once becomes flat in 2016, then the issues are increasing again. The number of issues in `vscode` draws a sawtooth wave after 2017. It means that the developers regularly review issues, the core members solve many issues at once at certain timing, probably before a new release.

Although the number of remaining issues is changing uniquely for each project, the ratios is well decreased for all projects, as shown in the graph R-RIS(p, t).

The lines in N-RIS(p, t) and R-RIS(p, t) should be converged to zero. However, it is generally hard to achieve when the population grows. Because various people post various issues, the quality of the issues also varies, involving important issues as well as non-essential issues.

4) Ability of Problem Solving: LT-CIS(p, t) in Figure 3 represents the ability of problem solving in the projects, characterizing how many days are spent to solve the closed issues on average. The vertical axis shows the life time of closed issues in the number of days, indicating the velocity of problem solving. Preferably, LT-CIS(p, t) should be maintained below a certain value d, which assures that every problem would be solved around d days. However, LT-CIS(p, t) for all projects are continuously increasing. This observation may be due to the increase of the population. When a lot of issues including non-essential ones are posted, the solutions cannot catch up with the new issues, and not all issues are need to be addressed. Especially, the lifetime of `atom` suddenly increased since the end of 2017, which becomes over 120 days in 2019. The reason of the sudden growth will be investigated in our future work, taking the quality of issues into consideration.

5) Capability of Government: R-MGPR(p, t), LT-PR(p, t) and R-PRLP(p, t) in Figure 3 represent the capability of government in the projects.

The vertical axis of R-MGPR(p, t) plots the ratio of merged pull requests which characterizes the acceptability. The acceptability of `vscode` is lower than other two editors after 2016. To determine the ideal value is quite difficult, since it requires to evaluate the quality of modifications performed by the pull requests. If almost all modifications are quite important ones, R-MGPR(p, t) should be close to 1. However, in general, the qualities of the modification are gradually down as the population of the project grows. In that means, the value of R-MGPR(p, t) of three editors may be natural consequence.

The vertical axis of LT-PR(p, t) represents the lifetime of pull requests at that time. In three editors, the values of LT-PR(p, t) gradually increase, and they reach around 15-20 days in 2019. This means that even if the number of pull requests increases, the capability of the community making the decision does not significantly increase. Consequently, the response time will be extended. LT-PR(p, t) is shorter than LT-CIS(p, t), because an issue is posted *before* the problem is solved, and a pull request is created at least *after* a direction of solution is proposed.

The vertical axis of R-PRLP(p, t) plots the ratio of pull requests lacking participants. The value of `atom` is around 0.3, which is significantly higher than other two editors. In general, this result is not considered to be good. Note that the cause of the result may be related to the policy of `atom`. The ideal value of R-PRLP(p, t) is zero. That is, in the community, all of the pull requests are reviewed by other developers before merging. In the context of R-PRLP(p, t), the communities of `brackets` and `vscode` obey the rule well.

D. Discussion

As shown in Figure 3, we have visualized the evolution of the three major projects by plotting the proposed metrics on time-series graphs. Each graph explains how the projects have evolved from a specific viewpoint of a city. We can see that each of the three projects has own characteristics of evolution, however, we can also see some common observations:

1) The population and the size continue to grow for a long time until now.
2) As the population and the size grow, the number of problems and the time to resolve them increase. The agility of the government also becomes slow.
3) The problems are addressed actively by the community, decreasing the ratio of unresolved issues.
4) The acceptability of the government is converged to a certain value, which is specific to a project. The same observation holds for the soundness.

The first observation indicates that the increase of the population is significantly related to the spontaneous software evolution. The more people come to a project, the more new issues and their resolutions are born. The second observation reflects the existence of *social overhead*. As more people come, the quality of issues and pull requests vary. The overhead to make a consensus for every issue would be increased. The third observation shows that in good projects, the critical problems should be eliminated constantly, which produces the dropping-to-the-right curve in the graph of R-RIS(p, t). This is similar to an ideal shape of the *burn-down chart* of Scrum [12]. The fourth observation reflects a *policy* within core members of every project, to which we should perform deeper investigation.

The great advantage of the proposed method is to quantify the state of a project p at any given time t. This allows community members to assess the current state of the project as a city, reminding them of what the problems are, and what should be done. Thus, monitoring the state by the citizens themselves is crucial for the governance of the project. Moreover, the metrics can be integrated with relevant CI (continuous integration) and CD (continuous delivery) tools. A promising idea is to implement a *bot* that triggers appropriate actions when the current state is out of acceptable state.

The limitation of the proposed method is that the metrics have not yet stepped into the *quality aspects* of elements. For example, we did not consider the importance of issue.

Although fixing a critical bug may be more important than adding a new feature, the proposed metrics do not discriminate them. In GitHub, developers can attach labels to every issue, which can be used to measure the importance. However, the convention of labeling is different among individual projects. Therefore, it is not easy to quantify the importance. Investigation of such quality aspects is left for future work.

VI. Related Work

GitHub repository has been analyzed in many previous studies. Gousious et. al. [13] summarized GitHub repositories from Feb. 2012 to Aug. 2013. However, this research does not distinguish the type of application, does not target issues, and does not consider time series, therefore it differs from our research.

Steinmacher et. al. [14] analyzed qualitative and quantitative aspects on pull requests submitted by developers other than core members. However, this research's targets are only developers who are not core members, and that it does not target other than pull requests, therefore it differs from our research.

Trockman et. al. [1] investigated the influence of the badges, which are used to make it easy to understand the transparency of github repositories. The factors related to the badges (transparency itself ot easy to understand the characteristics of the project) may also affect the analysis may be related in out future work.

Yang et. al. [15] investigated the relationship among reviewers by using contribution and review comments in 4 OSS projects. Impact analysis of the relationship between reviewers for the rise and fall of the project will be out future work.

Constantiou et al. [16] summarize the number of commits, the number of lines, and the number of projects for the ruby project on GitHub in time scale. This study focuses on a specific language, therefore it is different from the our study that targets specific application types.

Pinto et. al. [17] analyzed the behavior of "casual contributors" by some development language in GitHub.The research target is only developer, therefore that is different from our research.

Borges et. al. [18] focused on GitHub starring and analyzed the characteristics of projects with many stars. However, the analysis target is only star, which is different from our research subject.

Our study is similar to these related work in that it analyzes the GitHub repository. However, our research is different in that it focuses on the time series change of participants involved in the repository and its activities.

VII. Conclusion

To understand the mechanism of spontaneous software evolution, we have presented a method that investigate evolution of projects in social coding platforms (SCPs) . Introducing the concept of project as a city, we first identify essential attributes for the spontaneous evolution, then develop metrics that quantify the state $S(p, t)$ of a project p at time t. We also conduct an experimental evaluation where we visualize the evolution of projects of famous code editors within GitHub.

We are currently continuing the experimental evaluation with more projects to find good/bad patterns of evolution corresponding to good/bad practices. These patterns would be crucial information to govern the project for achieving sustainable and spontaneous evolution. Investigation of quality aspects is also important future work.

Acknowledgment

Part of this work was supported by JSPS KAKENHI Grant Numbers 17K00196, 17K00500, 17H00731, 18H03242, 18H03342, and 19H01138.

References

[1] A. Trockman, S. Zhou, C. Kästner, and B. Vasilescu, "Adding sparkle to social coding: An empirical study of repository badges in the npm ecosystem," in *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, (New York, NY, USA), pp. 511–522, ACM, 2018.

[2] "GitHub: The world's leading software development platform." https://github.com.

[3] "Bitbucket: The Git solution for professional teams." https://bitbucket.org.

[4] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, pp. 1060–1076, Sep. 1980.

[5] K. Matsumoto, H. Hata, M. Nakamura, H. Tamada, A. Ihara, S. Morisaki, M. Tsunoda, K. Toda, M. Ohira, and A. Monden, "Development of fundamental technologies to accelerate spontaneous software evolution." JSPS Kakenhi Grant-in-Aid for Scientific Research (A) JP17H00731, 2017.

[6] Microsoft, "Visual Studio Code." https://github.com/microsoft/vscode.

[7] M. Nakamura and L. du Bousquet, "Constructing execution and life-cycle models for smart city services with self-aware iot," in *IEEE 12th International Conference on Autonomic Computing (ICAC2015)*, pp. 289–294, July 2015.

[8] "Atom: The hackable text editor." https://github.com/atom/atom.

[9] Adobe, Inc., "An open source code editor for the web, written in JavaScript, HTML and CSS." https://github.com/adobe/brackets.

[10] GitHub, "GraphQL API v4." https://developer.github.com/v4/.

[11] "GraphQL June 2018 edition." https://graphql.github.io/graphql-spec/June2018/.

[12] G. Dinwiddie, "Feel the burn, getting the most out of burn charts," *Better Software*, vol. 11, no. 9, pp. 26–31, 2009.

[13] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 345–355, ACM, 2014.

[14] I. Steinmacher, G. Pinto, I. S. Wiese, and M. A. Gerosa, "Almost there: A study on quasi-contributors in open-source software projects," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 256–266, May 2018.

[15] X. Yang, N. Yoshida, R. G. Kula, and H. Iida, "Peer review social network (peRSoN) in open source projects," *IEICE Transactions on Information and Systems*, vol. E99-D, pp. 661–670, 3 2016.

[16] E. Constantinou and T. Mens, "Socio-technical evolution of the ruby ecosystem in github," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 34–44, Feb 2017.

[17] G. Pinto, I. Steinmacher, and M. A. Gerosa, "More common than you think: An in-depth study of casual contributors," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, pp. 112–123, March 2016.

[18] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of github repositories," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 334–344, Oct 2016.