Improving Collaboration Efficiency in Fork-based Development

Shurui Zhou
Carnegie Mellon University, USA

***Abstract*—Fork-based development is a lightweight mechanism that allows developers to collaborate with or without explicit coordination. Although it is easy to use and popular, when developers each create their own fork and develop independently, their contributions are usually not easily visible to others. When the number of forks grows, it becomes very difficult to maintain an overview of what happens in individual forks, which would lead to additional problems and inefficient practices: lost contributions, redundant development, fragmented communities, and so on. Facing the problems mentioned above, we developed two complementary strategies: (1) Identifying existing best practices and suggesting evidence-based interventions for projects that are inefficient; (2) designing new interventions that could improve the awareness of a community using fork-based development, and help developers to detect redundant development to reduce unnecessary effort.**

***Index Terms*—Fork-based Development, Distributed Collaboration, Awareness of Collaboration, Open-Source Community**

I. Problem: Inefficiencies in Social Forking

Collaboration is essential for software development at scale, in both industrial and open-source projects. Fork-based (or branch-based) development is a lightweight mechanism that allows developers to collaborate remotely. Developers could simply copy code files from the original project, while having freedom and independence to make modifications [1]. Recent advances in distributed version control systems (e.g., 'git clone') and social coding platforms (e.g., GitHub fork) have made fork-based development relatively easy and popular [2] by providing support for tracking changes across multiple forks and mechanism for integrating changes back [3]. We measured from the GHTorrent [4] data and found that over 114,120 GitHub projects have more than 50 forks, and over 9,164 projects have more than 500 forks as of June 2019, with numbers rising quickly.

Before the rise of social coding, forking traditionally referred to the intention of splitting an independent development line, competing with the original repository, often with a new name. We use the term social fork (**fork** for short) in the sense of creating a public copy of a git repository and refer to the traditional definition of the splitting of a new independent project as a **hard fork**.

While easy to use and popular in practice, fork-based development has well-known downsides. When developers each creates their own fork and develop independently, their contributions are usually not easily visible to others, unless they make an active attempt to merging their changes back into the original project. When the number of forks grows, it becomes very difficult to keep track of decentralized development activity in many forks. The key problem is that it is *difficult to maintain an overview* of what happens in individual forks and thus of the project's scope and direction. Also, in industrial contexts, researchers found that it is hard for individual teams to know who is doing what, which features exist elsewhere, and what code changes are made in other forks [5].

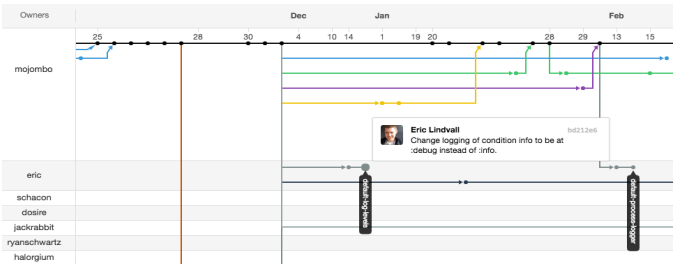

Fig. 1: GitHub's network graph shows commits across known forks, but is difficult to use to gain an overview of activities in projects with many forks [6].

Developers are interested in what happens in other forks, but cannot effectively explore them with current technology, such as GitHub's network view (see Fig. 1), which visualizes the history of commits over time across all branches and forks of a project [3]. Although the network view is a good starting point to understand how the project evolves, it is tedious to use if a project has many forks. Furthermore, the problem of **lacking an overview** of forks can lead to several additional problems and **inefficient practices**:

- ***Lost contributions:*** Developers may fix bugs, or add useful features in forks, but unless they contribute those changes back to the original project, those contributions are easily lost, although these changes are technically public [6]. In our study, we regard a community in which more developers *attempt to contribute* their changes to upstream as more efficient. We calculate the fraction of forks that attempt to contribute any changes back among all active forks in our sample of 1131 GITHUB projects (details can be found in paper [7]). And we found that a median of 50% active forks never contribute back (see Fig. 2a).
- ***Redundant development:*** Unaware of activities in other forks, developers may re-implement functionality already developed elsewhere. Gousios et al. [2] studied that 23% pull requests (PRs) on GitHub were rejected due to redundant development. Also, this would demotivate developers from continuously contributing to the repository [8] and signif-

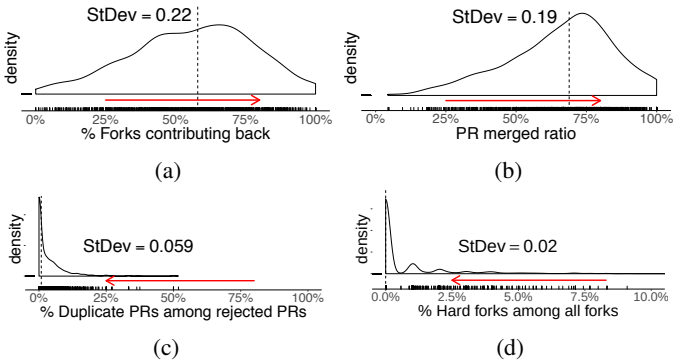

Fig. 2: Density plots of four inefficient forking practices with high variance among projects in our sample. The arrow points towards higher efficiency. The dashed line shows the median.

icantly increase the maintenance effort for maintainers [9]. Plotting the fraction of PRs rejected due to redundancies in Fig. 2c, we can observe that redundant development is a small but pervasive problem (mean 3.4 %; max 51 %).

- ***Rejected pull requests:*** When developers submit a PR that gets rejected, they can perceive this as a waste of their effort and get discouraged from contributing further [10]. One common reason for rejecting a PR is misalignment with the maintainers' vision of the project [10], [11]. From the community's perspective, a project in which most PRs are accepted can be considered as more effective with regard to contributor efforts. Observing the rate of rejected PRs among all closed PRs in our 1131 GITHUB projects plotted in Fig. 2b, we see that in most projects a majority of PRs are accepted, but also note the high variance.
- ***Fragmented communities:*** Diffusion of efforts can be observed on GITHUB in many secondary forks (*i.e.*, forks of forks) that contribute to other forks, but not to the original repository [12]. This fragmentation can seriously threaten the sustainability of open source projects when scarce resources are additionally scattered across multiple projects. In fragmented communities, we see multiple related repositories receive contributions, but those contributions are rarely shared. Hard forks are rare but potentially very expensive for a community. Fig. 2d shows that a median of 5% sampled projects have hard forks. Even though this only happens to some projects, but the problem is severe.

II. AN OVERVIEW OF PROPOSED SOLUTIONS

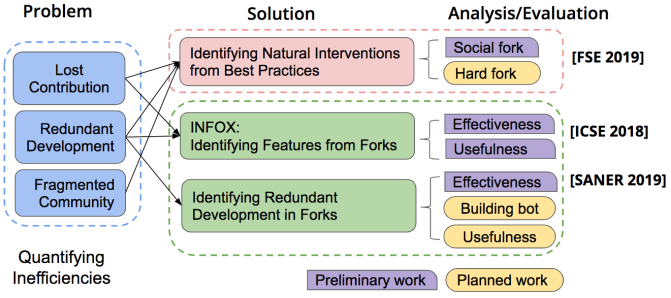

Fig. 3: Outline of proposed work. (Arrows present the mapping between solution and its targeting problems)

In this thesis, we first study the problem space by quantifying the efficiencies in fork-based development. Then we propose two strategies to mitigate these problems: First, we would like to identify existing best practices and suggesting evidence-based interventions to projects that are inefficient; second, we would like to build tools that could improve the awareness of a community and help developers to detect redundant development to reduce developers' unnecessary effort. To evaluate the effectiveness and usefulness, we propose to conduct both quantitative and qualitative studies.

III. IDENTIFYING EXISTING BEST PRACTICES.

In analyzing the inefficiencies of our sampled projects set, we found that projects are indeed very different regarding the degree of inefficiencies (see Fig. 2). These strong differences bring us the opportunity of improving efficiencies for open-source projects by identifying existing interventions from some projects that are more efficient. Therefore, we would like to understand: **What characteristics and practices of a project associate with efficient forking practices?**

A. Research Method

We designed a mixed-method approach to first interview 15 developers to identify candidate project characteristics and practices that may influence effectiveness in a project. Then, based on those interviews, we derived *eight hypotheses* about modularity and coordination-related factors and how they influence the inefficiencies:

$\mathbf{H}_1$. *Projects with a better modular design have a larger portion of contributing forks.*

$\mathbf{H}_2$. *Projects with a better modular design accept a higher fraction of PRs.*

$\mathbf{H}_3$. *Projects pursuing a centralized management strategy have a larger portion of contributing forks.*

$\mathbf{H}_4$. *Projects pursuing a centralized management strategy have a larger portion of merged PRs.*

$\mathbf{H}_5$. *Projects in which external developers tend to discuss or claim an issue before submitting PRs have a lower frequency of redundant development.*

$\mathbf{H}_6$. *Projects with a lower PR merge ratio have higher likelihood of having at least one hard fork.*

$\mathbf{H}_7$. *Projects with a more modular design have higher likelihood of having at least one hard fork.*

$\mathbf{H}_8$. *Projects pursuing a centralized management strategy have higher likelihood of having at least one hard fork.*

Next, we iteratively *operationalized inefficiencies and context factors (modularity and coordination)* in GITHUB repositories. Several measures are nontrivial and are built on top of significant prior research (more details are in the paper [7]). Finally, we tested our hypotheses using *multiple regression modeling*.

B. Key Findings & Insights

Based on our modeling results, we find evidence in support of both $\mathbf{H}_1$ and $\mathbf{H}_3$: projects with stronger coordination practices, as evidenced by advanced planning of what work needs to be done through issue linking, or more modular architecture, tend to have a higher fraction of contributing forks that submit patches upstream. Besides, even after controlling for confounds, we found positive effects on the average PR merge ratio, which supports $\mathbf{H}_2$ and $\mathbf{H}_4$: the more modular the architecture, or the more planned the PRs are, i.e., in response to open issues, the higher the average acceptance rate. Furthermore, our result shows that the higher the rate at which PRs are pre-communicated, the lower the overall rate of duplication among PRs ($\mathbf{H}_5$). However, the model fit is rather poor, so we conclude cautiously that: there is only weak evidence that claiming PRs before working on them associates with lower risk of duplicate work.

Regarding hard forks, our model confirms a sizeable negative effect for the PR merge ratio, strongly supporting $\mathbf{H}_6$. The centralized management index also has a statistically significant positive effect supporting $\mathbf{H}_8$. We do not find a statistically significant effect though for the modularity associating with hard forks ($\mathbf{H}_7$).

C. Planned Work – Exploring Trade-offs Regarding Fragmented Communities due to Hard Forks

We have mainly focused on social forks, while there is value to explore the benefit or trade-off of hard forks, which could lead to fragmented communities, and understand the transition between social forks and hard forks. So we propose a mix-method study that attempts to understand: **Why do hard forks occur? What characteristics of a community that associate with would lead to hard forks? What are the trade-offs between social forks and hard forks?**

IV. Designing new interventions to improve awareness.

Awareness solutions could increase the transparency in collaborative software development [13]. As there is a lot of information that is publicly available but not easily accessible, we saw opportunities for building awareness tools and mitigating inefficiencies. We designed an approach INFOX [6] to summarize un-merged code changes in forks in order to generate a better overview of the community (see Fig. 4 of INFOX overview page). We also designed an approach [14] to identify potentially redundant code changes to save developers' effort.

A. INFOX: Identifying Features in Forks

We design an approach to identify unmerged cohesive code changes (named features) from forks (called INFOX).

1) Method: INFOX takes the diff between the latest commit of the upstream and the latest commit of each fork from GITHUB, for which gathers the non-merged changes from fork. Then it proceeds in three steps:

Fig. 4: INFOX overview

- Identify a dependency graph among all added or changed lines of code by parsing and analyzing the code for multiple kinds of dependencies.
- Cluster the lines of the change based on the dependency graph using a community-detection technique, mapping each line of code to a feature, so that lines with many connections in the graph are mapped to the same feature.
- Label each cluster by extracting representative keywords with an information-retrieval technique.

2) Evaluation:

a) Effectiveness: We test the effectiveness of INFOX in a controlled setting by quantitatively comparing clustering results of INFOX and the state-of-the-art approach against a ground truth of known features in a number of open-source projects. Detailed of experiment design is described in paper [6]. The result shows that INFOX could reach 90% accuracy on a set of known features.

b) Usefulness and Actionable Insights: We contacted open-source developers who maintain forks that contain un-merged code changes to validate identified features and explore whether the generated summaries provide meaningful insights. We interviewed 11 developers from 7 different projects (response rate of 13.6%). In summary, **participants generally agreed that INFOX could identify correct clusters at certain splitting or joining steps**.

Furthermore, we looked particularly for signs that developers learned new insights while exploring the overview. Of the 11 participants, 8 gained different kinds of new information from the overview page:

- *Finding redundant development.* Two participants found other forks that are working on the same feature implementation as they did before.
- *Find interesting and potentially reusable feature.* 6 participants identified specific features of interest that are important to the project or they could reuse in their own forks.

In summary, even though we interviewed only a small number of participants, **we found frequent and concrete evidence of new insights gained from the overview page, including redundant development and reusable contributions**.

B. Identifying Redundancies in Fork-based Development

In this section, we focus on another inefficiency – redundant development in fork-based development. Our goal is (1) to

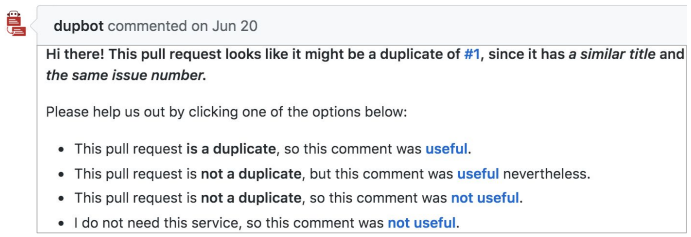

Fig. 5: Mock-up bot: sending duplicate warnings.

help project maintainers to automatically identify redundant PR order to decrease the workload of reviewing redundant code changes, and (2) to help developers detect redundancies as early as possible by comparing code changes with other forks in order to eliminate wasted effort and encourage developers to collaborate.

1) Training a Classifier to Detect Redundant Changes: To identify potential clues that might help us to detect if two changes are duplicates, we randomly sampled 45 PRs that have been labeled as *duplicate* on GITHUB from 5 projects. For each pair of duplicate PRs, we manually inspected the text and code change information to extract clues indicating the potential duplication and summarized the clues characterizing the content of a code change as follows:

- **Change description** is a summary of the code changes written in natural language, *e.g.*, commit messages, PR title & description.
- **Reference to issue tracker** is a common practice that developers explicitly link the code change to an existing issue or feature request in the issue tracker.
- **Patch content** is the differences of text changes in each file by running *'git diff'* command.
- **A list of changed files** is a set of changed files in the patch.
- **Code change location** is a range of changed lines in the corresponding changed files.

Next, we calculate the similarity between a pair of changes for each clue and then predict the probability of the two changes being duplicate through a classification model (see details in paper [14]).

2) Evaluation – Effectiveness: We evaluate the effectiveness from two perspectives: (1) helping maintainers to identify redundant PRs to decrease the code reviewing workload, (2) helping developers to identify redundant code changes implemented in other forks to save the development effort. The result shows that our approach could achieve 57-83% precision for identifying duplicate PRs and help developers save 1.9-3.0 commits per PR on average.

3) Planned Work – Evaluation of Usefulness: We plan to implement a bot for GITHUB and explore the research questions: **To what extent do developers agree with our duplication detection result?** The bot will monitor the coming PR of each repository and informs the project maintainer when the duplication is detected. We plan to measure the success rate of our detection result, in which success means the duplicate PR pairs we detect is correct. Also, we will analyze how participants react after receiving the notification, such as how positive the feedback is to evaluate the usefulness of our bot. A mock-up of the bot is shown in Fig. 5.

V. CONCLUSION

In summary, we would like to understand *how efficiently developers use forks in different communities, and to what degree project characteristics and practices of open-source communities associated with inefficiencies*. Specifically, we designing measures to quantify inefficiencies, and then we propose two strategies to mitigate these problems.

ACKNOWLEDGMENTS

The author is advised by Prof. Christian Kästner at Carnegie Mellon University. This project has been supported in part by the NSF (awards 1318808, 1552944, and 1717022). We thank James D. Herbsleb, Andrzej Wąsowski, and Laura Dabbish for their comments and advice on this project.

REFERENCES

[1] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *Proc. Europ. Conf. Software Maintenance and Reengineering (CSMR)*. IEEE, 2013, pp. 25–34.

[2] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 2014, pp. 345–355.

[3] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in github: transparency and collaboration in an open software repository," in *Proc. Conf. Computer Supported Cooperative Work (CSCW)*. ACM, 2012, pp. 1277–1286.

[4] G. Gousios, "The ghtorent dataset and tool suite." IEEE Press, 2013, pp. 233–236.

[5] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wąsowski, "Three cases of feature-based variability modeling in industry," in *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MoDELS)*. Springer, 2014, pp. 302–319.

[6] S. Zhou, Ş. Stănciulescu, O. Leßenich, Y. Xiong, A. Wąsowski, and C. Kästner, "Identifying features in forks," in *Proc. Int'l Conf. Software Engineering (ICSE)*. New York, NY: ACM Press, 5 2018.

[7] S. Zhou, B. Vasilescu, and C. Kästner, "What the fork: A study of inefficient and efficient forking practices in social coding," in *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, 2019.

[8] I. Steinmacher, G. H. L. Pinto, I. S. Wiese, and M. A. Gerosa, "Almost there: A study on quasi-contributors in open-source software projects," in *Proc. Int'l Conf. Software Engineering (ICSE)*, 2018, pp. 1–12.

[9] Ş. Stănciulescu, S. Schulze, and A. Wąsowski, "Forked and Integrated Variants in an Open-Source Firmware Project," in *31st International Conference on Software Maintenance and Evolution*, ser. Proc. Int'l Conf. Software Maintenance and Evolution(ICSME), 2015.

[10] I. Steinmacher, G. Pinto, I. S. Wiese, and M. A. Gerosa, "Almost there: A study on quasi-contributors in open-source software projects," in *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 2018, pp. 256–266.

[11] A. E. Azarbakht, "Longitudinal analysis of collaboration in forked open source software development projects," Ph.D. dissertation, Oregon State University, 2017.

[12] K. H. Fung, A. Aurum, and D. Tang, "Social forking in open source software: An empirical study." in *CAiSE Forum*. Citeseer, 2012, pp. 50–57.

[13] C. Treude and M.-A. Storey, "Awareness 2.0: staying aware of projects, developers and tasks using dashboards and feeds," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 1. IEEE, 2010, pp. 365–374.

[14] L. Ren, S. Zhou, C. Kästner, and A. Wąsowski, "Identifying redundancies in fork-based development," in *Proc. Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER)*, 2019, pp. 230–241.