Automated Patch Porting across Forked Projects

Luyao Ren
Peking University
China

ABSTRACT

Forking projects provides a straightforward method for developers to reuse existing source code and tailor it to their own application scenarios, which can significantly reduce developers' burden. However, this process makes forked projects (upstream projects and their forks) share the same defects on reused code as well. With the independent development of forked projects, some defects can only be repaired in one of them, where the patches need to be ported to others as well. Manually tracking all such activities among them is hard. Previous studies reveal that porting patches across forked projects is imperative and call research in this direction. Targeting at this problem, we conducted an empirical study to analyze the characteristics of patches in forked projects. We found that 20.5% patches need to be ported among all analyzed patches, which is a non-negligible portion. Among all those patches that need to be ported, 73.2% can be easily ported by simple syntactic code transformations. However, it is still challenging for other 26.8% patches since the corresponding code has experienced different modifications in the forked projects. As a result, according to the insights from the study, we proposed a new approach, which aims to automatically identify and port patches across forked projects.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Patch porting, Open-source, Project forking, Code transformation

ACM Reference Format:
Luyao Ren. 2019. Automated Patch Porting across Forked Projects. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 3 pages.
https://doi.org/10.1145/3338906.3342488

1 INTRODUCTION

Forking is a straightforward method for developers to copy an existing software project and then tailor it to their own applications, which is widely used in both open-source and industry developments [2, 3, 13]. Since a large portion of the code in forked projects is reused, same defects could exist among those reused code. With the independent development of forked projects, some of defects can be found and repaired by developers, where the same defects may still exist in peer projects. For example, the same defect could be found and repaired in one project more than one year later than in another peer project as shown in Listing 1 and 2. Manually tracking such activities in all forked projects takes much effort of developers [8], which informs further research on automated patch porting techniques across forked projects.

To understand the characteristics of patches in forked projects, we first conducted an empirical study on the patches in forked projects from open-source repositories. We found that porting patches among forked projects are challenging. Since forked projects are developed independently, the corresponding source code may experience different modifications, which makes the patch porting hard or even inapplicable. One situation is the contexts in forked projects are changed greatly, which makes the patches cannot be applied directly, such as variables are replaced. The other situation is the defect has already been repaired but in a different way, where we need to identify whether the defect still exists in the peer projects first, such as the patch in Listing 1 should not be applied to its peer code in Listing 2 after May 21st, 2019, where different patches are applied to repair the same defect.

According to manual inspection, we proposed a preliminary approach to automatically identify and port patches across forked projects, which involves program analysis and code adaptations.

```
Commit : github.com/vim/vim/commit/d23a823
Message: patch 8.0.1496: clearing a pointer takes two lines
Source : src/edit.c
==========
    static void
ins_compl_del_pum(void)
{
    if (compl_match_array != NULL)
    {
        pum_undisplay();
-       vim_clear((void **)&compl_match_array);
+       VIM_CLEAR(compl_match_array);
    }
}
```

Listing 1: A patch in Vim(2018-02-11)

```
Commit : github.com/neovim/neovim/commit/ae846b4
Message: vim-patch:8.0.1496: VIM_CLEAR()
Source : src/nvim/edit.c
==========
static void ins_compl_del_pum(void)
{
  if (compl_match_array != NULL) {
    pum_undisplay(false);
-   xfree(compl_match_array);
-   compl_match_array = NULL;
+   XFREE_CLEAR(compl_match_array);
  }
}
```

Listing 2: Ported Patch in NeoVim(2019-05-21)

2 STUDY METHODOLOGY AND RESULTS

For the first stage of research, to analyze the characteristics of patches in forked projects, we used four pairs of forked projects. We took the upstream one or the older one as the reference project

Table 1: Study Results of Patches in Subject Projects

Ref. / Tar. Proj.	#Sampled	#Porting Needed		#Syn. Eq.
		#Resolved	#Unported	
Vim/NeoVim	50	14	2	8
FreeBSD/OpenBSD	50	2	1	2
Marlin/Ultimaker2Marlin	50	1	2	2
OpenRefine/LODRefine	50	1	18	18
Total	200	18	23	30

and the other as the target project. First, we fetched all the commits which are in the reference project's commit history but not in the target project's. Then, we collected patches from those commits based on the occurrence of keywords including "bug", "fix", "patch" in the commit message which is widely used in previous works [7, 10]. As a result, we obtained 42786 patches in total. Then, we randomly sampled 50 patches for each pair of forked projects. For each patch, we checked whether it needs to be ported to the target project by manually checking its commit message and the corresponding contexts in both the reference projects and the target projects. Here, we call a patch is "porting needed" if it has already been resolved in the target project (the patch has been ported or the corresponding defect has been fixed), or it has not been ported and its corresponding defect still exists in the target project.

The result is shown in Table 1. We found that 20.5% patches in the referenced projects need to be ported to the target projects, which is a non-negligible portion of all analyzed patches. Among all those patches that need to be ported, we found that 73.2% (30/(18+23)) patches could be directly ported without additional modifications. However, for the rest of 26.8% patches, in order to make these patches compatible in the corresponding contexts in the target projects, extra adaptations on the patches are required.

Additionally, from the result we could find that the maintenance behaviors on porting patches among different forked projects are diverse. For the forked projects like Vim and NeoVim, developers manually ported the patches frequently (14 Resolved vs 2 Unported), denoting the importance of ported patches, while for the forked projects like OpenRefine and LODRefine, developers tracked such kind of patches less frequently (1 Resolved vs 18 Unported).

Therefore, an automated patch porting technique can be helpful in reducing developers' manual effort to port patches across forked projects. Some insights are learned from the study to achieve it. We found that both the patch's current contexts and the evolution history of the contexts are important when identifying the unported patches. Additionally, comprehension of the semantics of patches is crucial for adaption on patches if the corresponding contexts are not compatible for directly porting. To understand the semantics of patches, it is necessary to extract how the specifications are changed when applying the patches based on their descriptions and code changes. We call the changes of specifications as Δ_{spec}.

3 PRELIMINARY APPROACH

According to the previous analysis, to reduce maintenance workload of developers, we proposed an initial framework that helps developers to identify and port patches in forked projects automatically. As shown in Figure 1, the framework contains three

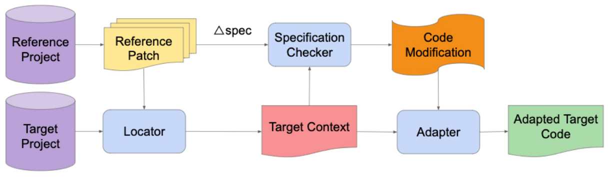

Figure 1: Framework

core parts: a locator, a specification checker, and a patch adapter. Given patches from reference projects, the locator will find the corresponding contexts of patches in target projects. Then, for each patch, the checker will extract the Δ_{spec} and examine whether the target project satisfies it or not, if not, the checker will generate corresponding code modifications based on the Δ_{spec} and the target contexts. Then, the adapter will adapt the code modifications for the target contexts to finish the patch porting.

When the corresponding contexts of patches in target projects have been modified, it may lead to inconsistencies which would be challenging. To tackle it, we intend to use some program analysis techniques and semantic code transformation techniques.

4 RELATED WORK

Ray et al. [8] studied ported edits in three BSD forked projects and found the maintenance of porting changes in forked projects is significant and heavily depends on developers, for which SPA [9] was proposed to detect and characterize porting errors. Similarly, Gabel et al. [4] proposed DejaVu to detect buggy inconsistencies in reused code, while Jablonski and Hou [5] proposed CReN to avoid copy-paste errors by tracking code changes in the IDEs. They are targeting different application scenarios compared to ours. Additionally, Tian et al. [12] proposed an approach to automatically identify bug fixing patches in Linux commit history, which potentially can be leveraged for patch identification in our approach.

Recently, Meng et al. [6] proposed SYDIT, which could automatically adapt a given code change to the target code. However, it does not identify the locations where patches need to be ported and thus does not conform to the patch porting application. Barr et al. [1] proposed an approach to automatically transplant functionality code across projects. It is different with ours as they migrate a code snippet with an independent functionality while our goal aims to automatically identify and port patches. Thung et al. [11] proposed an automatic recommendation system to guide the backporting of Linux device drivers. It focused on porting code across versions of the same project while our scope is across forked projects.

5 DISCUSSION AND FUTURE WORK

Based on the study result, we found out a non-negligible portion of the patches from the reference projects that need to be ported to the target projects. To automatically port those patches, key challenges lie in identifying the compatibility of the patches and porting patches with preserving the desired semantics. Motivated by this, we proposed a framework integrating three components to approach these challenges. For the next stage of research, we intend to implement our framework and conduct a user study by making pull requests of applicable patches on forked projects.

REFERENCES

[1] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated Software Transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 257–269. https://doi.org/10.1145/2771783.2771796

[2] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering (CSMR '13)*. IEEE Computer Society, Washington, DC, USA, 25–34. https://doi.org/10.1109/CSMR.2013.13

[3] Neil A. Ernst, Steve M. Easterbrook, and John Mylopoulos. 2010. Code forking in open-source software: a requirements perspective. *CoRR* abs/1004.2889 (2010). arXiv:1004.2889 http://arxiv.org/abs/1004.2889

[4] Mark Gabel, Junfeng Yang, Yuan Yu, Moises Goldszmidt, and Zhendong Su. 2010. Scalable and Systematic Detection of Buggy Inconsistencies in Source Code. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 175–190. https://doi.org/10.1145/1869459.1869475

[5] Patricia Jablonski and Daqing Hou. 2007. CReN: A Tool for Tracking Copy-and-paste Code Clones and Renaming Identifiers Consistently in the IDE. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange (eclipse '07)*. ACM, New York, NY, USA, 16–20. https://doi.org/10.1145/1328279.1328283

[6] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic Editing: Generating Program Transformations from an Example. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 329–342. https://doi.org/10.1145/1993498.1993537

[7] Audris Mockus and Lawrence G. Votta. 2000. Identifying Reasons for Software Changes Using Historic Databases. In *Proceedings of the International Conference on Software Maintenance (ICSM'00) (ICSM '00)*. IEEE Computer Society, Washington, DC, USA, 120–. http://dl.acm.org/citation.cfm?id=850948.853410

[8] Baishakhi Ray and Miryung Kim. 2012. A Case Study of Cross-system Porting in Forked Projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 53, 11 pages. https://doi.org/10.1145/2393596.2393659

[9] Baishakhi Ray, Miryung Kim, Suzette Person, and Neha Rungta. 2013. Detecting and Characterizing Semantic Inconsistencies in Ported Code. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. IEEE Press, Piscataway, NJ, USA, 367–377. https://doi.org/10.1109/ASE.2013.6693095

[10] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes?. In *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR '05)*. ACM, New York, NY, USA, 1–5. https://doi.org/10.1145/1082983.1083147

[11] Ferdian Thung, Xuan-Bach D. Le, David Lo, and Julia L. Lawall. 2016. Recommending Code Changes for Automatic Backporting of Linux Device Drivers. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*. 222–232. https://doi.org/10.1109/ICSME.2016.71

[12] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying Linux Bug Fixing Patches. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 386–396. http://dl.acm.org/citation.cfm?id=2337223.2337269

[13] Shurui Zhou, Ştefan Stănciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wąsowski, and Christian Kästner. 2018. Identifying Features in Forks. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM Press, New York, NY, 105–116. https://doi.org/10.1145/3180155.3180205