Towards Understanding and Fixing Upstream Merge Induced Conflicts in Divergent Forks: An Industrial Case Study

Chungha Sung
University of Southern California
Los Angeles, CA, USA

Shuvendu K. Lahiri
Mike Kaufman
Pallavi Choudhury
Jessica Wolk
Microsoft Corporation
Redmond, WA, USA

Chao Wang
University of Southern California
Los Angeles, CA, USA

ABSTRACT

Divergent forks are a common practice in open-source software development to perform long-term, independent and diverging development on top of a popular source repository. However, keeping such divergent *downstream* forks in sync with the *upstream* source evolution poses engineering challenges in terms of frequent merge conflicts. In this work, we conduct the first industrial case study of frequent merges from upstream and the resulting merge conflicts, in the context of Microsoft Edge development. The study consists of two parts. First, we describe the nature of merge conflicts that arise due to merges from upstream. Second, we investigate the feasibility of automatically fixing a class of merge conflicts related to *build breaks* that consume a significant amount of developer time to root-cause and fix. Towards this end, we have implemented a tool MrgBldBrkFixer and evaluate it on three months of real Microsoft Edge Beta development data, and report encouraging results.

ACM Reference Format:
Chungha Sung, Shuvendu K. Lahiri, Mike Kaufman, Pallavi Choudhury, Jessica Wolk, and Chao Wang. 2020. Towards Understanding and Fixing Upstream Merge Induced Conflicts in Divergent Forks: An Industrial Case Study. In *42nd International Conference on Software Engineering Companion (ICSE '20 Companion), October 5–11, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3377812.3390800

1 INTRODUCTION

Divergent forks are a common practice in open-source development, e.g., to provide customized products by adapting an open-source project. Leveraging an upstream software that defines or adheres to some standards (e.g., Android) allows the downstream software to offer better application compatibility. As an example, web browsers such as Opera, Samsung Internet, and Microsoft Edge build upon the Chromium engine; similarly, customized versions of the Android mobile operating system are offered by various smartphone vendors, together with their own applications.

Unlike a branch that is often short-lived, a divergent fork may live permanently along side the original project. However, flow of information between the original and forked repositories is asymmetric. While most divergent forks need to continuously integrate changes from the original repository, e.g., to keep up with important security patches, changes from the forked repositories seldom flow back into the original repository. To signify this asymmetric nature, we refer to the original repository as the *upstream* and the forked repository as the *downstream*.

Although popular and convenient, a divergent fork may incur significant overhead. One challenge is to keep the downstream synchronized with important updates in the upstream. As the upstream software evolves due to API changes and security patches, the downstream needs to be updated accordingly. That is, the downstream needs to perform a *merge* from the upstream. While merge conflicts are not unique to divergent forks [1, 2], the cost of root-causing and fixing the asymmetric *upstream merge induced conflicts* is significantly higher, for three reasons: (1) Changes in the upstream often occur without knowledge of the downstream development. (2) Root-causing the upstream commit responsible for merge conflict, especially build break, is non-trivial when the commit history of the upstream consists of several thousand commits. (3) A merge induced build break may also be caused by changes in the downstream, often made many commits earlier. This makes it difficult to find the right developer to assign the fix.

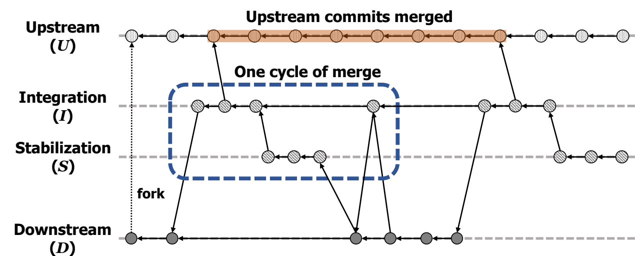

Figure 1: The branch structure of Microsoft Edge.

2 STUDY OF UPSTREAM MERGE-INDUCED CONFLICTS IN EDGE

In this section, we study the upstream merge-induced conflicts in the context of Microsoft Edge development, a recent divergent fork of Chromium.

Branch structure of Microsoft Edge. Figure 1 gives a simplified branch structure in Microsoft Edge. Each horizontal line represents one of the four branches: *Upstream* (U), *Integration* (I),

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE '20 Companion, October 5–11, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7122-3/20/05.
https://doi.org/10.1145/3377812.3390800

Table 1: List of resolution cases in upstream induced build break.

Group	Type of Resolution/Fixes in Downstream	Possible (Example) Causes from Upstream
1	Include Statement Update	- File/Directory Name/Structure is updated
2	Entire Function Definition/Call Update (e.g., function body move/add/removal)	- Function definition (with body) is added/removed - Function definition (with body) moved to different class/section (e.g., public → private)
3	Function Name Update	- Function name is updated - Entire function is removed (e.g., function deprecation)
4	Function Type/Specifier Update	- Function definition type is updated - Function definition specifier is added/removed
5	Function Param/Arg Update (e.g., param/arg add/rem/reorder)	- Function definition parameter is added/removed/reordered
6	Function Param/Arg's Type Update (e.g., param/arg type update)	- Function definition param's type is updated - Function definition param's specifier/modifier is added/removed - Hierarchy/Name of Class/Namespace /Enum definition is updated
7	Class/Namespace/Enum Reference Update (e.g., field type update)	- Hierarchy/Name of Class/Namespace /Enum definition is updated
8	Uncategorized	

Stabilization (S) and *Downstream* (D). Circles indicate commits and an arrow points from a child to a parent commit. Here the D branch denotes the master branch of Edge, and the U denotes the Chromium master. D is created by the "fork", and both branches evolve independently.

Every cycle of merge, the downstream pulls the changes from the upstream in two phases. First, textual (syntactic) conflicts are resolved in the I branch after pulling the recent version of the upstream. Then, any build errors (e.g., compiler errors) or test failures are resolved in the S branch. Finally, the source code is merged back to D master.

Merge Conflicts. We classify resolutions of the conflicts during merges for three months from April to June in 2019, and there are 2,218 commits to resolve all the merge induced conflicts. We consider three types of conflicts: textual conflict, build break and test failure. The number of fix commits for textual conflict is 1183, for build breaks is 815, and for test failures is 220. And, we have studied how build breaks are resolved and identified three categories: *Fixes of ill-formed files*, *Build script file fixes* and *Structural fixes in C++ files*. Especially, we focus on *Structural fixes in C++ files* because, among all classes of fix commits, developers found it to be the most laborious to identify the root-cause and prepare the fix for the downstream code.

Structural fixes in C++ files. Consider a method *Foo* that was defined in the upstream. Then, the downstream created some new call-sites for *Foo*. At some point, however, *Foo* was renamed to *Bar*, a new parameter was added, and then all the call-sites were properly changed in the upstream. When merging such a change into the downstream, compilation will cause a *build* break due to the missing symbolic link. During our study, we observe many such conflicts. Furthermore, the root-cause is often not obvious to downstream developers, and developers have to manually inspect the upstream commits (which can be a few thousands), analyze the change impact, and then create a suitable patch.

We further identify common sub-categories of the *Structural fixes in C++ files* into eight groups as shown in Table 1. While the example causes from the upstream are provided to help understand the breaks, they are not meant to be exhaustive. Since the classification is inherently manual, there exists a set of commits for which we could not find the exact patterns with possible causes; therefore, it is classified as *Uncategorized* (Group 8).

3 FEASIBILITY OF AUTOMATIC FIXES

To evaluate the feasibility of automated fixes of merge induced build breaks, we develop a prototype tool named MrgBldBrkFixer. Our preliminary targets are Group 3, 4, 6 and 7 from Table 1.

As shown in Figure 2, the input consists of (i) a set of upstream commits, C, that constitutes the merge, and (ii) a build break error, ϵ, in a downstream C++ file f. The output is the patched downstream file f' aimed to resolve the build break. Internally, there are four steps: (1) Identify the symbol σ in f that is responsible for the build break error ϵ. (2) Prune the upstream commits in C to remove the ones not relevant to σ, to obtain $C' \subseteq C$. (3) Analyze changes to definitions and uses (Defs and Uses) in the files modified in C' based on AST-aware diffing results, to infer a set of possible *renaming patches*, denoted Π. (4) For each patch $\pi \in \Pi$, apply π to the AST node (in f) that contains σ, to obtain f'.

Using real development data of Microsoft Edge collected in a three-month period, we perform a feasibility study and, the result shows that 40% of the build breaks targeted by MrgBldBrkFixer can be repaired automatically.

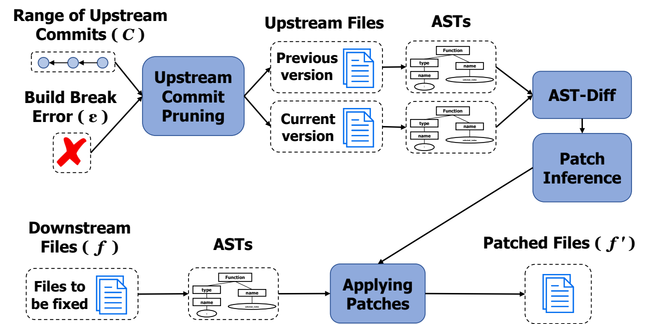

Figure 2: Overview of the automated patching.

4 CONCLUSIONS

We have presented the first industrial case study of upstream merge induced conflicts in a divergent fork, namely the Microsoft Edge. We identified structural fixes in source files, that require substantial manual effort to root-cause and fix due to the scale of upstream commits. We provided a simple analysis based on constructing a patch for such conflicts, and our preliminary results are encouraging.

REFERENCES

[1] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. 2011. Proactive Detection of Collaboration Conflicts. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*. 168–178.

[2] C. R. B. de Souza, D. Redmiles, and P. Dourish. 2003. "Breaking the Code", Moving Between Private and Public Work in Collaborative Software Development. In *International ACM SIGGROUP Conference on Supporting Group Work*. 105–114.