Can Program Synthesis be Used to Learn Merge Conflict Resolutions? An Empirical Analysis

Rangeet Pan*, Vu Le†, Nachiappan Nagappan†, Sumit Gulwani†, Shuvendu Lahiri†, Mike Kaufman†

*Iowa State University, Ames, IA, USA

rangeet@iastate.edu

†Microsoft Corporation, Redmond, WA, USA

{levu, sumitg, shuvendu.lahiri, mike.kaufman}@microsoft.com, nnagappan@acm.com

Abstract—Forking structure is widespread in the open-source repositories and that causes a significant number of merge conflicts. In this paper, we study the problem of textual merge conflicts from the perspective of *Microsoft Edge*, a large, highly collaborative fork of the main *Chromium* branch with significant merge conflicts. Broadly, this study is divided into two sections. First, we empirically evaluate textual merge conflicts in *Microsoft Edge* and classify them based on the type of files, location of conflicts in a file, and the size of conflicts. We found that ~28% of the merge conflicts are 1-2 line changes, and many resolutions have frequent patterns. Second, driven by these findings, we explore *Program Synthesis* (for the first time) to learn patterns and resolve structural merge conflicts. We propose a novel domain-specific language (DSL) that captures many of the repetitive merge conflict resolution patterns and learn resolution strategies as programs in this DSL from example resolutions. We found that the learned strategies can resolve 11.4% of the conflicts (~41% of 1-2 line changes) that arise in the C++ files with 93.2% accuracy.

Index Terms—merge conflict, program synthesis, automated fixing

I. Introduction

Textual merge conflicts occur when changes from independent commits over a common base cannot be resolved by a textual 3-way differencing tool present in version control systems such as git. Merge conflicts are frequent and annoying, and studies have shown they can have a significant impact on the development lifecycle of projects [1]. Prior works have focused on predicting merge conflicts [2]–[7], resolving them through using structured abstract syntax tree (AST) aware techniques [8]–[12], or even verifying semantic correctness of merges using program verifiers [13]. Although these approaches have paved the course for research on merge conflict prediction and resolution, the problem of merge-conflicts persists in practice. The reason can be attributed to (a) lack of structure-aware tools for most languages apart from Java [8], and (b) the black box nature of these tools that rely on sophisticated matching algorithms to relate the changes make them inaccessible to mainstream developers.

In this paper, we explore the problem of merge conflict resolution through a slightly different perspective. Instead of devising an algorithm to resolve any merge conflict, we pose the merge conflict resolution problem as that of *learning repetitive resolution patterns* from the history of a given project. In fact, Ghitto *et al.* [1] observed patterns are commonplace in conflict resolution for a large project when viewed over the history of a project. They also postulate that learning common resolution patterns in a project can be a useful aid for developers in resolving future conflicts.

With this motivation, we conduct a large-scale empirical study of textual merge conflicts that occur in *Microsoft Edge* (a fork of the *Chromium* browser), and apply program synthesis (in particular, programming by example (PBE) [14], [15]) to illustrate the feasibility of learning patterns for a large class of resolutions. We chose *Microsoft Edge* as a subject due to (a) the collaborative nature of the project with 500+ developers and (b) the large number of merge conflicts that manifest as a result of merging changes from the *Chromium* repository to this fork. The study consists of two main parts. First, we perform an empirical study to classify the nature of merge conflicts for this project in terms of the prevalence of file types, size of conflicts, type of conflicts, and the resolution patterns. Second, we design a domain-specific language (DSL) to capture the resolution patterns in this project and use program synthesis to automate the learning of resolution strategies as programs in this DSL.

In our rigorous empirical study of *Microsoft Edge*, we found that a majority (~47%) of the files with conflicts were written in C++, and a significant number of changes (~28%) are of 1-2 lines. Among these conflicts in C++, we find that 12.34% are related to headers and macros. More importantly, we observe that these small conflicts usually follow a few distinct resolution patterns. Specifically, we found 9 different patterns in all of C++ 1-2 liner changes and out of which, 6 patterns were found in `Include` and `Macro` related conflicts.

We describe one such pattern in Figure 1(a), where the header "cursor_type.mojom-shared.h" appears both in the forked and main branches and the developer excluded the one in the main branch (+ and - denote whether the statement has been taken into the resolved solution or not. Only the changes with + operators are carried on to the resolution, whereas the changes with - operators are not). Figure 1(b) follows the same pattern, with the only difference being the specific name of the header file. Similarly, figures 1(c,d) show another pattern, where the developer removes "base/logging.h"

```
Forked Branch:
+ #include "ui/base/anonymous_ui_base_features.h"
+ #include "ui/base/mojom/cursor_type.mojom−shared.h"
Main Branch:
− #include "ui/base/cursor/mojom/cursor_type.mojom−shared.h"
```
(a) Header Location Update Example 1

```
Forked Branch:
+ #include "ui/base/anonymous_ui_base_features.h"
+ #include "ui/base/mojom/cursor_type.mojom−blink.h"
Main Branch:
− #include "ui/base/cursor/mojom/cursor_type.mojom−blink.h"
```
(b) Header Location Update Example 2

```
Forked Branch:
− #include "base/logging.h"
+ #include "base/scoped_native_library.h"
Main Branch:
+ #include "base/notreached.h"
```
(c) Removal of a Specific Header Example 1

```
Forked Branch:
+ #include "base/command_line.h"
− #include "base/logging.h"
Main Branch:
+ #include "base/check_op.h"
```
(d) Removal of a Specific Header Example 2

Fig. 1: Examples of Programming by Example (PBE) to Resolve Merge Conflicts in *Microsoft Edge*.

from the forked branch because *Microsoft Edge* uses a different logging system. These examples demonstrate the existence of common resolution strategies in the history of a project and the value of learning such frequent patterns to automate future conflicts.

Motivated by these patterns, we propose a domain-specific language (DSL) that contains the right set of abstractions to *succinctly* describe the patterns underlying the conflict-resolution examples. The key idea of our DSL is to describe the resolution as a guarded concatenation of fragments of the main and fork branches. The fragments are constructed using some ordered set operators including indexing, filtering, and subtraction.

Our DSL is able to express 11.44% of conflict resolutions, which also accounts for ∼41% of 1-2 line changes, seen in C++ files for *Microsoft Edge* with 93.2% accuracy. Expressing the resolution strategies as programs can address the limitations of fully automated tools in the hands of mainstream developers as: (1) developers can examine the program and understand the justification behind suggesting the changes, and (2) if new patterns emerge and existing patterns do not work, a new program can be learned by either providing a few examples or updating the synthesized program.

Our DSL, which contains the right set of abstractions that allow succinct expression of conflict resolution patterns, facilitates the development of a program synthesizer that can automatically learn intended patterns in this DSL from few examples. The key idea behind our synthesizer is to leverage a top-down synthesis methodology that reduces the synthesis problem to authoring efficient inverse functions for the various operators in our DSL [14].

Contributions: In summary, the paper makes the following contributions:

a) We empirically analyze textual merge conflict-related data from *Microsoft Edge* to qualify various measures, e.g., which files are most prevalent to have conflicts, how big these conflicts are, and where the conflicts lie in a file. Finally, we systematically discover and establish frequent patterns in resolving merge conflicts.
b) We build a domain-specific language (DSL) to address the `Include` and `Macro` related patterns established by the empirical evaluation.
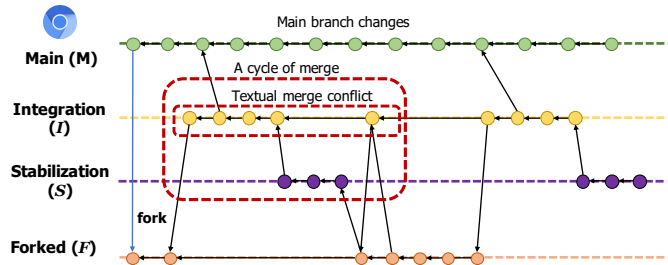

Fig. 2: The Structure of the *Microsoft Edge* Branch.

c) We leverage a top-down program synthesizer to learn the programs in this DSL to resolve the conflicts using input-output examples.

Organization: The paper is structured as follows: we discuss the dataset collection steps in §II and the empirical study in §III. Then, we describe the domain specific language and the use of program synthesis for learning patterns through examples in §IV. The results and related works are discussed in §V and §VI, respectively. Finally, §VIII concludes.

II. DATASET

A. Main/Fork Branch Structure

Figure 2 depicts the overall structure of *Microsoft Edge*, where each line denotes a branch. The *Main (M)* is the *Chromium*'s master branch and *Fork (F)* is the fork created by *Microsoft Edge* to pull the changes from the *M* to branch *F*. The arrow (⟶) depicts the child-parent (fork-main) relation between two branches. For example, the arrow from the main (M) to fork (F) denotes that F is a fork of M. There are two more branches: *Integration (I)* and *Stabilization (S)*. These branches are intermediate checkpoints while pulling the changes from M branch. In each cycle, the changes from the *M* branch are pulled to *F* branch through *I* and *S* branches. First, the conflicts are resolved at the *I* branch after merging with the *M* branch. Once conflicts are resolved, a fresh build is performed at the *S* branch and the changes in the *S* branch are then pulled to the master branch of *Microsoft Edge* (*F* branch).

A similar main-fork branch structure can be found in other projects (Brave, Colibri, Epic, Samsung Internet, LG WebOS (TV) web engine, etc.), mobile applications (android fork),

2

TABLE I: Merge Related Data for *Microsoft Edge*. M: Main Branch, F: Forked Branch

Merge Related Data	Week 1 03/30- 4/5	Week 2 4/6-4/12	Week 3 4/13-4/19	Week 4 4/20-4/26	Week 5 4/27-5/3	Week 6 5/4-5/10	Week 7 5/11-5/17	Week 8 5/18-5/24	Total	Median	SD
Merges	37	45	43	32	37	29	76	38	337	37.5	14.6
Commits in M	2031	1727	1835	2154	2068	2037	2213	1690	15755	2034.0	195.1
Commits in F	150	102	75	61	70	62	136	73	729	74.0	34.6
Conflicting Commits	27	45	38	30	36	29	33	33	271	33.0	5.8
Conflicting Files	214	269	225	272	289	276	353	165	2063	270.5	56.4
Conflicting Chunks	756	1212	533	529	5855	948	976	489	11298	852.0	1813.4

database (MariaDB), etc. In GitHub, there are more than 1.5 million forked projects in C++ alone [16].

B. Data Collection

First, we identify the merge commits by filtering the commits with two parents, one from the *Main (M)* branch and one from the *Forked (F)* branch. Next, we replay the merge commits for obtaining the conflicting files for each merge commit. Along with the conflicting file, we store the resolved version of each conflicting files.

In Table I, the summary of the merge commits is shown. We collected the merge data for eight weeks (March 30 to April 24, 2020), and we divide the data by each week. In the first row, the total number of merge commits are shown. These merge commits denote the number of times *F* branch pulled the changes from the *M* branch. The second and third row denotes the total number of commits made by *M* and *F* branches during the period of time. The fourth row denotes the number of merges that result in introducing conflicts in at least one file. The fifth row shows the total number of files having at least one conflict. Finally, in the sixth row, we report that the number of conflicting chunks that have resulted from merging the two branches' changes. We found that, on average, 80.4% of the merges and 37.2% of all commits at *F* branch result in conflicts. Also, on average, ∼258 conflicting files are generated per week. We found that the number of merges (SD: 14.6), commits in main (SD: 195.1) and fork (SD: 34.6), conflicting commits (SD: 5.8), and conflicting files (SD: 56.4) to be very consistent over the 8 weeks. In comparison to the Java related conflicts [1], *Microsoft Edge* (Mean 852) has bigger conflicting chunks than the ones found in the prior work (Mean 20). Based on the p-value and correlation value (corr), we found that conflicting chunk size ($p = 0$, $corr = 0.31$) has no impact on the number of conflicts.

III. CLASSIFICATION OF MERGE CONFLICTS

In this section, we empirically study the merge conflicts in *Microsoft Edge* and answer four questions:

A) (**File Type**) Which file types are the most prevalent to have merge conflicts?
B) (**Conflict Size**) How large are the conflicts?
C) (**Conflict Location**) What are the different types of conflicts based on the origin of location?
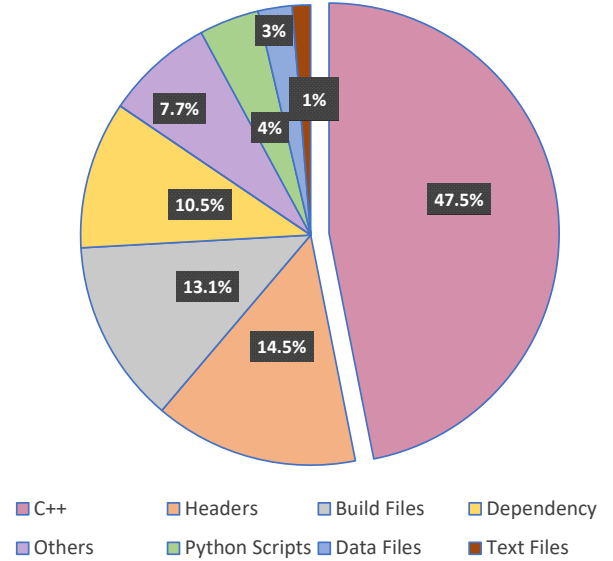D) (**Resolution Pattern**) What are the conflict resolution patterns in *Microsoft Edge*?

Fig. 3: Classification Based on the File Type.

A. Which file types are the most prevalent to have merge conflicts?

Here, we classify the conflicts based on the type of files. In Figure 3, we denote the percentage of the conflicts for each file type. We found that most of the conflicts (47.5% of the total conflicts) are in the C++ related files, and this is because the core functionality of both *Chromium* and *Microsoft Edge* are written in C++. In the following paragraphs, we briefly discuss the basic characteristics of each file type.

1) C++: The conflicts in these files are related to both logical and structural changes in the code. The structural changes include the addition, removal, or update of the header files, macros, etc. The logical changes are introduced to alter the logic of the operation. For example, addition of conditional check, loop structure, etc.

2) Dependency: These files are the dependency files that are needed to build *Microsoft Edge* from scratch. Very often, dependencies, e.g., the version, controllable parameters, etc. are updated by the main and the forked branch. We found that various internal tools are used to automate the resolution of conflicts in this genre of files.

3) Headers: The definition and structures of the class are defined that are used in other files. 14.5% of the files having at least one conflicts are header related ones.

4) Build Files: With each build (including run-time compilation), a considerable number of files are auto-generated.
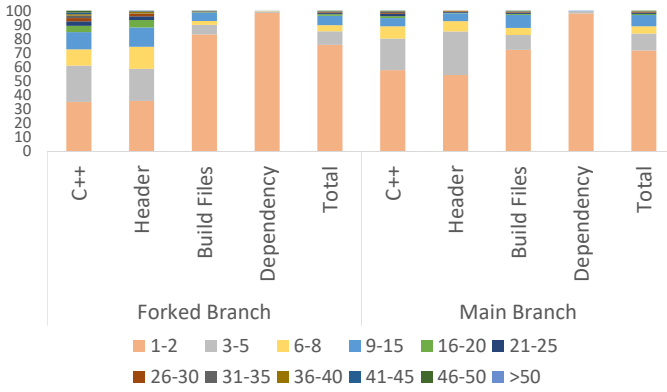
Fig. 4: Classification Based on the Conflict Size.

Fig. 5: Classification Based on the Location of the Conflicts.

When they are pushed along with other changes, they often result in having conflicts. Since they are auto-generated codes, rebuilding can resolve the conflicts, and they do not need any particular logical or structural changes for resolution.

5) Python Scripts: 4.3% of the conflicting files occur in the python scripts (.py and .pyl files). These files are utilized for running different scripting level codes.

6) Data Files: Mostly, these are matrix market files (.mm), and grid data format (.grd) files and changes in the data often cause the conflicts. However, only 2.4% of the conflicts occur in these files.

7) Text Files: Changes in the documentation related files such as *readme* files are categorized here.

8) Others: File types responsible for less than 1% of the conflicting files are grouped in this category. Mostly, comma-separated files (.csv), protobuf files representing the structure of the code (.proto), etc. are in this group.

B. How big are the conflicts?

> 💡 **Finding 1** $\Rightarrow$ 35.17% and 57.74% of the conflicts in C++ are 1-2 lines of change by the main and forked branches, respectively. Overall, 28% of the conflicts are of 1-2 lines for both main and forked branch.

Each conflicting chunk has been divided into main and fork section based on the conflict markers. Furthermore, we categorize each section based on the number of lines of changes. We created twelve groups starting from 1-2 lines to more than 50 lines and compute the same for each type of files other than the "others" category. Here, we refer to the conflict size as the number of lines reported by the conflict markers (main/fork section). We found that the majority of the conflicts are of 1-2 lines for either main (71.70%) or fork (75.72%). For C++ files, where the majority of the conflicts occur, we found that these numbers are 35.17% and 57.74% for main and fork, respectively. Also, we found that 28% of the conflicts in C++ are of 1-2 lines for both main and forked branches. Similar to what Ghitto *et al.* [1] observed in Java projects, we found that a non-trivial amount of conflicts are due to small
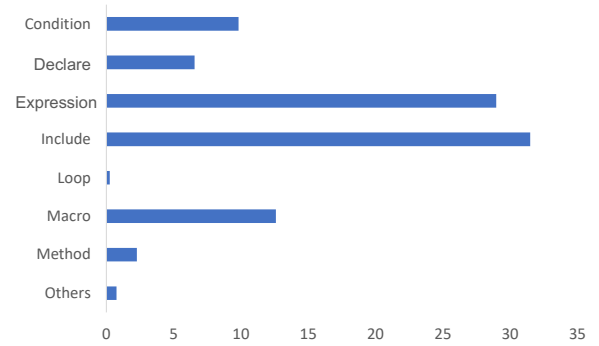
changes ($\leq$8 lines) in C++ ($\sim$85% and $\sim$75% for main and fork, respectively). The distribution of the conflicts is similar to the one found in Java. In Java, 96% of the conflicts are of <50 lines of code. Whereas in *Microsoft Edge*, 98% of the conflicts are of <50 lines of code. Furthermore, we computed the p-value (0) and the correlation value (0.35) and found that lines of code has no impact on the number of conflicts.

Given that smaller resolutions are more likely to be idiomatic and repetitive, we focus on the 1-2 lines of conflicts in both the main and forked regions. Moreover, since the majority of conflicts are in C++ files, we focus our study on the C++ files.

C. What are the different types of conflicts based on the origin of location?

Based on the evidence that suggests the majority of the conflicts are in C++ files and are 1-2 lines in size, we move forward to classify the location of the conflicts.

We utilized GumTree [17] to tokenize the code and use the abstract syntax tree (AST) to determine the changes. We found that the conflicts mostly occur in seven locations that have been depicted in Figure 5. In the following paragraphs, we briefly discuss each type of such location with appropriate examples.

1) Condition: In this type of change, the conditional logic is updated. For example, in the code snippet below, the main branch added an extra condition that resulted in a conflict during the merge. This kind of conflict is 9.82% of the total observed conflicts. This is similar to the percentage of condition related conflicts found in Java (9%) by the prior work [1].

```
Forked Branch:
  if (result != DID_NOT_HANDLE)
Main Branch:
  if (elastic_overscroll_controller_ && result !=
      DID_NOT_HANDLE)
```

2) Declare: In this type of change, the variable declarations are altered (6.55% of the observed conflicts). In the following example, the value of the variable *kSettingsVersion* has been changed to 1 from 4 by the main branch, and the data type has also been updated.

```
Forked Branch:
  static const uint32_t kSettingsMagic = 'CPds';
  static const uint32_t kSettingsVersion = 4;
Main Branch:
  static constexpr uint32_t kSettingsMagic = 'CPds
      ';
  static constexpr uint32_t kSettingsVersion = 1;
```

3) Expression: 28.9% of the conflicts originated due to changes in expressions. Change in the parameter while calling a procedure, calling a different procedure, etc., are categorized in this classification section.

> 💡 **Finding 2** ⇒ 31.49% of 1-2 lines of conflicts in C++ files are due to addition, deletion, or updating the `Include` section.

4) Include: These types of changes are the most prevalent ones in the observed dataset. Mostly, the changes include the addition, deletion, and updating of the location of the headers. For instance, in the following example, the main and forked branch added a new header file that caused the conflict to occur.

```
Forked Branch:
#include "chrome/browser/ui/views/accessibility/
    hc_with_theme_bubble_view.h"
Main Branch:
#include "chrome/browser/ui/views/accessibility/
    caption_bubble_controller_views.h"
```

We found that 31.49% of conflicts of size 1-2 lines are related to the include statement. This category of conflicts is higher than the observation made by Ghitto *et al.* [1], where 6% of the conflicts in Java are due to the import statement (similar to the include statement in C++). The prior work has analyzed all the conflict sizes, whereas our analysis is based on the 1-2 liner conflicts.

5) Loop: Only a small amount of conflicts (0.25%) result from the change in the looping structure.

6) Macro: C++ programs make extensive use of macros to enable code reuse. We found that 12.6% of the conflicts in C++ that are of 1-2 lines are due to changes in the macro. One such example has been depicted in the code snippet, where value corresponds to an argument has been changed by

```
Forked Branch:
IN_PROC_BROWSER_TEST_P(V4,
    ANONYMOUS_DISABLED_PERMANENT(
    MalwareWithWhitelist, 20395305)) {
Main Branch:
IN_PROC_BROWSER_TEST_F(V4) {
```

the forked branch. In *Microsoft Edge*, macros are mostly used to disable the main branch's (*Chromium*) tests that do not work in *Microsoft Edge* for various reasons, e.g., flaky test, or main branch tests a feature that *Microsoft Edge* does not ship, or related to debugging. However, a large majority of the macro conflicts are more precisely test-disablement.

7) Method: 2.3% of the changes that cause conflicts, are related to the changes in the method declaration, e.g., the return type, return value, etc. This genre of conflicts is less in C++ than in Java (10%).

8) Others: In this category, we classify any conflicts that occur other than the previously mentioned ones. Mostly, they are situated in the comment block, and only 0.8% of the conflicts under observation have been caused in these places.

D. What are the conflict resolution patterns in Microsoft Edge?

Next, we empirically evaluate every 1-2 lines of changes (both main and forked branch) and identify the patterns based on the type of operations that the developers did to resolve the merge conflict. In this context, any resolution that removes the conflict markers is treated as a conflict resolution. We used the open coding scheme to build the classification scheme. First, we label the 10% of the merge conflicts and build the classification scheme. Then, if needed, we add the classification category and revise the labels. Furthermore, the classification scheme has been verified by the third and the fifth authors, who are experts in software engineering study and merge conflicts. Finally, when no new classification category has been added, we completed the labeling process. A total of nine types of operations are found in our dataset. In the following paragraphs, we discuss each type of merge conflict, resolving patterns, and understand the root cause of such patterns with proper examples.

1) Apply Forked Specific Changes (AFSC): *Microsoft Edge* has created a fork, and periodically they pull the changes from the Chromium to update the codebase. We found that developers from *Microsoft Edge* update the code to reflect the *Forked branch* specific changes, e.g., disabling tests, adding forked branch specific header, etc. When this kind of change is reflected in the merge conflict, the forked or *Microsoft Edge* specific changes are taken into account, and the main branch-specific changes are discarded.

Example 3.1: Apply fork specific changes, where *Microsoft Edge* added a test disabling check.

```
Forked Branch:
+ IN_PROC_BROWSER_TEST_F(V4,
    ANONYMOUS_DISABLED_PERMANENT(
    CheckUnwantedSoftwareUrl, 20395305)) {
Main Branch:
- IN_PROC_BROWSER_TEST_F(V4,
    CheckUnwantedSoftwareUrl) {
```

In the code snippet, the forked branch introduced a check in the second parameter of the browser test case procedure, which has been used to disable a portion of the test cases in *Microsoft Edge*. In the resolution of the merge conflict, the developer chose the fork branch changes discarding the main branch changes. These kinds of operations are mostly prevalent in changes located in macro (42.0%), expression (22.6%), and include statements (4.80%).

2) Change in the Header (CH): A small portion of the conflicts is due to a change caused by another change in a

Fig. 6: Classification Based on the Type of Operations that Developers Do to Resolve. **AFSC**: Apply Forked Specific Change, **CH**: Change in the Header, **Concat**: Concatenation, **DDC**: Dependency on different conflicts, **FB**: Frequent Behavior, **LC**: Logical Change, RD: Remove Duplicate, and **SI**: Space Introduced.

different place (mostly in a header file). For instance, the forked branch's change has added an extra parameter while calling the procedure. As the forked branch had made a change in the header (function *OnExtensionDownloadFinished*) by adding a new parameter, the same change has been reflected in the C++ file to be consistent. While resolving the conflict, developers chose the latest changes (fork branch-related).

Example 3.2: A change in the header file caused the update of the following code.

```
Forked Branch:
+  ON_CALL(*this, OnExtensionDownloadFinished(_, _
      , _, _, _, _, _))
Main Branch:
-  ON_CALL(*this, OnExtensionDownloadFinished(_, _
      , _, _, _, _))
```

3) Concatenating Changes from two branches (Concat):

💡**Finding 3** ⇒ 39.5% of the resolution strategies involved concatenating the main and the forked branch's changes.

A vast majority of the resolution strategies are based on keeping both the changes that resulted in a conflict. For example, in case of the changes in the include statements, the developers can often take both changes without any particular order (if there is no commonality between the header files). Assume the main branch has made a change that involves the addition of the header *header1.h*, and the forked branch added a header *header2.h*. A developer often chooses to resolve the conflict by taking both the headers. Since these headers in *Microsoft Edge* usually contain independent classes, the order in which the include statements are concatenated does

not matter. However, for changes where more complex logic is involved, the concatenation requires to be more aligned with semantics rather than be on the syntactic changes. For example, in the code snippet, the resolution can be either the changes in the forked branch, followed by the changes from the main branch or the changes from the main branch, followed by the changes from the forked branch. If the latter strategy is taken, the header2 and header3 (actual name of the header has been changed) will be included in the conditional statement, so they would only be called if the conditional check returns true. However, this would create an anomaly in the code if there is a dependency on the header files irrespective of the value of the conditional check.

Example 3.3: Applying both the changes done by the forked and the main branch.

```
#if !defined(PROJECT_ANONYMOUS_BUILD)
#include <header1_name>
Forked Branch:
+ #endif
Main Branch:
+ #include <header2_name>
+ #include <header3_name>
```

4) Dependency of Different Conflicts (DDC): This type of resolution strategy accounts for a small number of scenarios (2.3%). If more than one conflict is present in the C++ file and one decision of resolution affects the resolution of another conflict, we label the fixing strategies as this group.

Example 3.4: A header file has a dependency on a section of code, which is also in the conflicting region.

```
Forked Branch:
+ #include "components/project_anonymous/core/
    common/switches.h"
- #include "components/version_info/channel.h"
```

```
Main Branch:
```

The header file (*"components/version_info/channel.h"*) has been referred to in a single place of the file, and that section was deleted while resolving another conflict. Here the reference is denoted by the class-object relation. The header file, *"channel.h"*, has a class named *channel* declared, and the object of that class has been used in the C++ file that includes the header statement. Since the object reference is no longer present, the developer decides to remove the `Include` statement as well.

5) Frequent Behavior (FB): We found that there are some repetitive patterns, where developers remove a specific header file while fixing the merge conflicts. For example, 92.8% of (13 out of 14) the scenarios involving the *"base/logging.h"* header file, the line related to the specific header file has been deleted, keeping other changes intact (Figure 1(c) and (d)). This is a project-specific pattern, which is due to the fact that fork is using the logging function differently than the main.

6) Logical Changes (LC): This genre of fixing strategies needs a logical understanding of the code. In most cases, the resolve includes a section of the main and the forked branch stitched by an appropriate logic.

7) Remove Duplicates (RD): These fixing strategies are mostly found in the condition (5.13%), declare (15.38%), expression (5.22%), and include (4%) related conflicts. Overall, 4.28% of the conflicts under observation are resolved by removing the duplicate content. Often, the main or forked branch moves a file into a different location and updates the same in the code. To resolve such conflicts, developers tend to remove the duplicate `Include` statement(s). For example, the code snippet in Figure 1(a) has the same header name *"cursor_type.mojom-shared.h"* in the main and the forked branch. However, the location of the file has been changed without changing the content of the header file. In such a scenario, the developer can choose any version of the *"cursor_type.mojom-shared.h"* file and take the rest of the changes from the main and the forked branch. The same can happen with header files outside the conflicting region.

Another scenario includes the variable declaration. In the following scenario, the variable *supervised_user_id* has not been initialized (main branch) or initialized with a default value (forked branch). However, in C++, the effect of either statement will result in the same effect. If the string is not initialized, then the default constructor will be called, and it will assign a zero size character to the string, which is similar to initializing with the empty string (*""*). To resolve such conflict, the developer can choose any version of the variable declaration and add that with the *endif* line.

```
Forked Branch:
+ #endif
+ std::string supervised_user_id = "";
Main Branch:
- std::string supervised_user_id;
```

8) Rename Related Fixes: File renaming is very common in a fork structure. In our case, resolving the rename related operation accounts for 5.29% of conflicts with 1-2 lines of change, and they are most prevalent in macros (38%). In this scenario, any of the two parties involved have changed the name of the function, and to make it consistent, every place where the function has been called, has been changed too. This kind of change often results in conflicts, and one such instance has been depicted in the following example.

Example 3.5: Microsoft Edge test has been renamed.

```
Forked Branch:
- IN_PROC_BROWSER_TEST_P(V4,
    ANONYMOUS_DISABLED_PERMANENT(CheckResourceUrl,
    20395305)) {
Main Branch:
+ IN_PROC_BROWSER_TEST_F(V4, CheckResourceUrl) {
```

The name of the test has been updated by the main to *IN_PROC_BROWSER_TEST_F* from *IN_PROC_BROWSER_TEST_P*. In such cases, the developer chose the renamed version of the function name and resolved the conflict.

9) Space Removed (SR): Introducing an extra space in the code can cause a conflict. As the default merging strategy is based on the textual difference, these scenarios are also considered as a case for merge conflict. In such scenarios, developers remove the space.

10) Others: Here, we labeled the conflicts that have no pattern of fixing or not enough information to conclude.

IV. CASE STUDY ON DEPLOYING SYNTHESIS FOR RESOLVING MERGE CONFLICTS

The empirical study has identified several patterns for resolving merge conflicts. We used these high-level patterns to guide the DSL to resolve the conflicts. We now discuss the use of program synthesis to learn programs expressing these patterns from examples. We first use some examples from the previous section to motivate the design of our DSL. We then formally define the DSL, and discuss the learning of programs in this DSL from examples using PROSE, an inductive program synthesis framework [14], [18]. The DSL that we propose possesses some syntactic sugar to be able to convey the high-level patterns. We did not want to break the patterns into assembly-level languages.

A. Motivating Examples

Recall that in Figure 1(c) and (d), the desired behavior is to remove the header "base/logging.h" from the forked branch's changes, then concatenate the changes in both branches. Additionally, we need a *guard* to only execute the resolution above when the forked branch contains "base/logging.h". Otherwise, the resolution would blindly apply to other non-applicable cases. Here is a program that expresses the pattern above:

```
Apply(FrequentPattern(x, "base/logging.h"),
    Concat(Main(x),
      Remove(Fork(x), ForkByPath(x, "base/logging.h"))))
```

In this program, `Apply` checks if the forked branch of the input x contains the header "base/logging.h" (using the

predicate `FrequentPattern`), then returns the resolved text as the concatenation of the main branch and the forked branch having the header "base/logging.h" removed.

In Figure 1(a), the header "cursor_type.mojom-shared.h" appears in both the main and forked branches, hence only the header in the forked branch is selected. Although we can use a predicate to check if both main and fork contain that particular header, such a predicate is very specific to that header name and cannot capture other headers. Instead, we introduce a predicate `DuplicateMainFork` to check if there are any headers that appear in both main and forked branches. We then resolve the merge by concatenating (a) the forked branch and (b) the main branch with these duplicated headers removed (note that `Pattern(...)` returns the duplicated headers).

```
Apply(DuplicateMainFork(x),
    Concat(Fork(x),
        Remove(Main(x), Pattern(x, "DuplicateMainFork"))))
```

B. DSL for Resolving Merge Conflicts

$$
\begin{aligned}
\text{Output } \mathbf{r} :=\ & \texttt{Apply}(\text{c}, \text{t}) \\
\text{Condition } \mathbf{c} :=\ & \texttt{And}(\text{p}, \text{c}) \mid \text{p} \\
\text{Predicate } \mathbf{p} :=\ & \texttt{DuplicateMainFork}(x) \\
& \mid \texttt{DuplicateMainOutside}(x) \\
& \mid \texttt{DuplicateForkOutside}(x) \\
& \mid \texttt{MainSpecific}(x) \mid \texttt{ForkSpecific}(x) \\
& \mid \texttt{Dependency}(x) \mid \texttt{Rename}(x) \\
& \mid \texttt{FrequentPattern}(x, path) \\
\text{Transformation } \mathbf{t} :=\ & \texttt{Concat}(\text{t}, \text{t}) \mid \texttt{Remove}(\text{s}, \text{s}) \mid \text{s} \\
\text{Selection } \mathbf{s} :=\ & \texttt{Main}(x) \mid \texttt{Fork}(x) \\
& \mid \texttt{ForkByPath}(x, path) \mid \texttt{ForkByIndex}(x, k) \\
& \mid \texttt{MainByIndex}(x, k) \mid \texttt{MainByPath}(x, path) \\
& \mid \texttt{Pattern}(x, key)
\end{aligned}
$$

Fig. 7: Syntax of our DSL. x refers to the input. $path, k, key$ denote a path constant, an integer constant, and a string constant, respectively.

Having demonstrated some program constructs in our DSL, we now discuss the DSL in detail. Figure 7 shows its syntax. At a high level, a program takes an input x, a struct that contains the main and fork sections denoted by the conflict marker, the location of the file, the rest of the content of the file, and returns the resolved text. We take the location of the file to extract the content of the imported header files and utilize them to detect the patterns, e.g., the example depicted in §3.4, where we check the rest of the content and the content of the header files along with the content between the conflict markers. The program uses `Apply` to check if some conditions $\mathbf{c}$ hold, then performs the merge resolution $\mathbf{t}$ that transforms the common base program to a resolved program.

Condition The condition $\mathbf{c}$ is a conjunction of one or more predicates. Our DSL has the following predicates to capture the classification discussed in Section III:

- `DuplicateMainFork`: Some content appears in both main and forked branches. Figures 1 (a,b) illustrate this predicate, in which the header "cursor_type.mojom-shared.h" and "cursor_type.mojom-blink.h" have the same name and content in both main and fork.
- `Duplicate[Main/Fork]Outside`: Some content appears in both the main (or forked) branch's conflicting region and non-conflicting region.
- `MainSpecific/ForkSpecific`: The main (or forked) branch has some identifiers that are main (or fork) specific. For instance, the change in Example 3.1 is fork specific because it has the macro *ANONYMOUS_-DISABLED_PERMANENT*.
- `Dependency`: One or more headers have dependency (control dependency) on other conflicting regions. Example 3.4 illustrates this predicate, where "channel.h" header has a dependency on another conflicting region.
- `FrequentPattern`: This predicate captures a repetitive pattern when a specific header (identified by the literal $path$) is present in the conflicting regions. Figure 1 (c) and (d) illustrate this predicate in which $path$ is "base/logging.h".
- `Rename`: Check whether the name of the tests in the macro has been altered or not. Example 3.5 uses this predicate to capture rename operation.

Transformation During the study, we observed that the merge resolution usually involves the concatenation of some nodes selected from the main or forked branches. The resolution also occasionally removes some nodes from a branch because they appear in both branches. Based on this observation, we designed the transformation $\mathbf{t}$ that allows arbitrary combination of concatenation (`Concat`) and removal (`Remove`) of nodes. The key insight of designing a DSL amenable for synthesis is that the DSL should be expressive enough to cover common patterns, yet it should be restrictive enough to allow for efficient learning. In our DSL, while `Concat` allows concatenation of two other transformations $\mathbf{t}$, `Remove` only allows removal of some selected nodes $\mathbf{s}$ from some other selected nodes $\mathbf{s}$. Restrictions like these reduce the search space for transformation significantly, while still allowing for the expression of all patterns in Section III.

Our DSL allows selecting the whole main or forked branch via `Main` and `Fork`. It also supports selection using index or path (`[Main|Fork]By[Index|Path]`). The former selects the kth node from a branch while the later selects a node based on path. Furthermore, some patterns in our study require selecting nodes that are relevant to the predicates. By selecting Main and Fork, we want to identify only the main related section of the conflict marker. The nodes are AST nodes. However, because our DSL only works with include and macros, we simplify the nodes to contain only relevant information. For instance, an include statement `include <foo>` corresponds to a node that has two children, one for "include" and another one for `<foo>`.

We allow such selection via `Pattern`, where a string

key is used to identify the predicate name. In the example in Section IV-A, `Pattern(x, "DuplicateMainFork")` selects nodes related to the predicate `DuplicateMainFork` (i.e., the nodes appearing in both the main and forked branches). Internally, our program calculates a dictionary that maps each pattern to some nodes. The dictionary is used both in evaluating the predicate and selecting the pattern nodes.

C. Synthesizing Programs from Examples

We now discuss the synthesis process to learn programs in our DSL from examples using PROSE [18]. PROSE framework is a meta-synthesizer, i.e., a system that allows people to implement inductive synthesizers more easily by re-using shared components, which would otherwise have to be reimplemented in different synthesizers. Prior to PROSE, authors of inductive synthesizers such as FlashFill [19], FlashExtract [20] had to implement their synthesizers from scratch. Newer synthesizers such as Refazer [21], BluePencil [15] leverage PROSE to speed up the development. We follow the same strategy. To create a new synthesizer in PROSE, one needs to (1) design the DSL and its semantics, (2) implement the witness/learning functions, and (3) define a ranking scheme for programs. All other low-level details such as building the version-space algebra, intersection of program sets, are taken care of by the PROSE framework. Readers may refer to [14] for a more detailed discussion on inductive program synthesis, as well as exemplar synthesizers on other domains.

The input (also called *specification*) to the inductive synthesizer is some examples in the form of $\{\sigma_i, o_i\}_{i=1}^n$ where σ_i is the input x, o_i is the expected resolved output, n is the number of examples. The PROSE methodology requires each DSL function to be accompanied with a *witness function* (WF, also called inverse function), which takes the output and returns the set of all input configurations to that function that can generate that output. Notice that a WF does the *inverse* of its normal function, which takes the inputs and returns the output. PROSE uses the WFs to decompose the input specification into "smaller" specifications for each of the parameters, effectively decomposing a complicated learning task into several smaller, simpler tasks. The divide-and-conquer process continues until it reaches the leaf nodes in the grammar, at which point the leaf nodes are either variables or literals.

Given a specification $\{\sigma_i, o_i\}_{i=1}^n$, the WF of `Apply(c,t)` reduces it to: (a) $\{\sigma_i, \text{true}\}_{i=1}^n$ for condition `c`, where we need to learn a conjunction of predicates that returns true on inputs σ_i, and (b) $\{\sigma_i, o_i\}_{i=1}^n$ for transformation `t`, where we need to learn a transformation that converts inputs σ_i to outputs o_i.

Learning Conditions The task here is to find predicates matching the patterns in the input σ_i. We simply select all predicates, whose evaluation on the input is `true`. One exception is `FrequentPattern`, where we also need to synthesize all possible values for *path* such that applying the predicate on *path* returns `true`.

Learning Transformations The goal here is to learn a sequence of transformations that transforms σ_i to o_i for all

i. While learning `Concat`, its WF decomposes the output o_i into all pairs o_{i1}, o_{i2} such that `Concat(`o_{i1}, o_{i2}`) =` o_i. This yields two sub-tasks with specifications $\{\sigma_i, o_{i1}\}_{i=1}^n$ and $\{\sigma_i, o_{i2}\}_{i=1}^n$. If PROSE finds transformations `t1` and `t2` for these sub-tasks, it returns `Concat(t1, t2)` as the result of learning this function. E.g., in Figure 1(d) where the resolved output contains two includes for "base/command_line.h" and "base/check_op.h", `Concat`'s WF assigns each include to a sub-task. PROSE recursively finds if there is a transformation `t1` that produces "base/command_line.h", and `t2` that produces "base/check_op.h".

The learning of `Remove` is similar; its WF decomposes the output o_i into all pairs o_{i1}, o_{i2} such that `Remove(`o_{i1}, o_{i2}`)` $= o_i$. In the above example, one way to obtain the include for "base/command_line.h" (o_i) is to select the forked branch (which contains "base/command_line.h" and "base/logging.h" - o_{i1}) and removes "base/logging.h" (o_{i2}) from it.

The learning of `[Main|Fork]By[Index|Path]` functions simply returns the index or the path of the output in the (main—fork) input, depending on whether the function is index-based or path-based. For instance, to produce "base/check_op.h", two possible selections are `Main(`x`)`, where we select the main branch, or `MainByIndex(`$x, 0$`)`, where we select the first line of the main branch.

To learn *key* for `Pattern`, we iterate over all patterns and return those whose nodes match the output. E.g., in Figure 1(a), the pattern DuplicateMainFork produces the duplicate node "ui/base/cursor/mojom/cursor_type.mojom-shared.h", hence the key "DuplicateMainFork" is selected.

Ranking Scheme Since there are often multiple programs that satisfy a given set of examples, inductive synthesizers use a ranking scheme to select the most likely program. For instance, in Figure 1(c), here are two possible choices for transformation **t** (among many others):

```
∘ Concat(MainByIndex(x,0),ForkByIndex(x,0))
∘ Concat(Main(x), Remove(Fork(x),
        ForkByPath(x, "base/logging.h")))
```

Although both programs are consistent with this scenario, our ranking scheme selects the latter program, which happens to be the intended one. We defined our ranking function to be a small weighted combination of a set of features that we discovered were relevant for our domain. For instance, we prefer programs that are smaller and use fewer constants since such programs generalize better to unseen inputs; hence, our feature set includes the number of operators and the number of constants in the program. Another feature accounts for favoring path-based programs to index-based programs because the former is more general, or for favoring selection from one of the branches since that is more likely.

V. RESULTS

We now present our experimental results of using program synthesis to automate merge conflict resolution.

Apart from the 8-week dataset that is used to design our DSL and synthesizer, we also collected merge conflicts for

TABLE II: Results of Using Program Synthesis on *Microsoft Edge* over 8 weeks data.

Pattern Name	Pattern Count	User Resolution
Concat	88	83 (94.3%)
AFSC (Include)	5	4 (80.0%)
RD	12	11 (91.67%)
FB	14	13 (92.8%)
DDC	6	5 (80.0%)
Rename	15	14 (93.33%)
AFSC (Macro)	22	21 (95.4%)
Total	162	151 (93.2%)

another 4 weeks (March 2 to March 29, 2020) to evaluate our approach.

In this section, we answer the following questions:

1) **RQ1:** What portion of the merge conflicts can be automated?
2) **RQ2:** How accurately can we assist developers in resolving the merge conflicts?

RQ1: What portion of the merge conflicts can be automated?
Our empirical study found that ~28% of the C++ conflicts are of 1-2 lines of changes for both the main and fork branch. Among those, `Include` and `Macro` related conflicts account for 31.5% and 12.6%, respectively. For `Macro` related conflicts, we focus on the test-disabling macros by selecting the conflicts having *Microsoft Edge* specific keywords such as `DISABLED`, `ANONYMOUS` (removed for double-blind) in the argument. We found that ~78% of the `Macro` related conflicts are of test disabling. For `Include` related issues, 8 cases (mentioned as `Others` in the classification) had files deleted when the developers resolved the conflict. We omitted those scenarios from our dataset because we were not able to get the resolution. Overall, our approach handles 11.44% of the total conflicts in C++ files.

RQ2: How accurately can we assist developers while resolving the merge conflicts?

> 💡 **Finding 4** ⇒ Overall, our approach handles 11.44% of the total conflicts in C++ files, which is 40.9% of all the 1-2 line changes in C++.

Table II presents our results on various `Include` and `Macro` scenarios on our 12-week dataset. In this table, the column `User Resolution` represents the total number of resolutions provided by our approach that match with the resolution performed by the developers. To perform the experiment, we extract the solutions after developers resolve all the conflicts in the commits and match that with our proposed resolution. We found that overall, our programs can assist the developers to resolve the merge conflict with 93.2% accuracy. If the conditions depicted in our DSL in §IV-B do not match the type of merge conflict that appears in the new examples, then our approach will not suggest. Also, these programs achieve 91.17% accuracy on the unseen 4-week dataset that we collected after our initial study. Because the conflicts in the 4-week dataset are not classified, we did not report its detailed results by classification in Table II. Instead, we report

the frequency of patterns detected while applying programs on this dataset. We found that Concatenation is applied 73.5% of the cases, Apply for Fork Specific (both `Include` and `Macro`) 8.8%, Frequent Behavior 5.88%, and Remove Duplicate 11.8%. These results indicate that program synthesis can be applied to resolve merge conflicts.

VI. RELATED WORK

Merge Conflicts: In this paper, we studied the problem of repeated patterns in textual merge conflict resolution in large projects. Ghiotto *et al.* [1] et al. perform a large-scale study of merge conflicts over Java projects and characterize the nature of merge conflicts and patterns in resolutions. The study envisions the need for tools that can capture the patterns in such resolutions. In addition to complementing the study by focusing on forks, our work can be seen as the first realization of the vision in this paper for a large project by using program synthesis to capture the frequent patterns. The problem of resolving merge conflicts soundly has received a lot of attention, going back to the work of program integration [22], where static analysis on the three input programs is performed to create a merge satisfying a conflict-freedom property. However, such approaches were never implemented or evaluated on real-world benchmarks. More recently, Sousa *et al.* [13] use program verifiers to check for the semantic conflict-freedom notion of correctness of a given merge, but do not synthesize the merge. Structured and semi-structured merge tools [8], [9] (such as JDIME) resolve some class of textual conflicts by lifting the textual 3-way differencing algorithm to the case of abstract-syntax trees. Since we are analyzing C++, we cannot perform a direct comparison due to the lack of such tools for C++. However, we believe our technique is complementary to such structured algorithms. One can apply the structured algorithms to first soundly resolve conflicts, and then turn to a synthesis guided approach to fall back on the learned resolution for the remaining. In addition, we believe that the structured techniques will not apply to most of the cases we encounter in this paper, due to the asymmetric nature of the fork related changes (e.g. AFSC), and renaming or movement of header files. Finally, the work of Sung *et al.* [23] studies and proposes fixing build breaks introduced by a merge in main/fork structure but does not deal with the problem of textual merge conflicts.

Bug Fixing: Recently, Bader *et al.* [24] applied a hierarchical clustering technique to address the fixing bugs by learning through examples. In this study, the AST based differencing technique has been used to identify the edits and clusters the edits based on the patterns. Finally, these fixes are applied to the code to address 7 different patterns in Java, e.g., null pointer exception, boxed primitive constructor, new class instances, etc. However, in our work, we empirically evaluated the merge conflicts in *Microsoft Edge* and take 1-2 examples for each pattern to build programs that can assist developers in resolving the `Include` and `Macro` related conflicts.

Program Synthesis: Prior work applied program by examples

(PBE), a form of program synthesis whose specification is examples, to various domains [19], [25], [26]. Gulwani introduced FlashFill, a system that synthesizes string transformation scripts from examples [19]. FlashExtract [25] allows end users to extract hierarchical data from semi-structured files by simply highlighting some examples. Anderson and Lawall [27], Anderson *et al.* [28] proposed techniques to learn version update patches for Linux files from input-output examples. Meng *et al.* [26], [29] developed SYDIT and LASE to synthesize code edit scripts from examples. Our work targets a new, different domain that has not been explored by PBE: merge conflict resolutions. Unlike prior work, which usually operates on a single snapshot of a file or program, our synthesizer learns patterns and transformation from various places: changes in the main branch, changes in the forked branch, and also file content outside the conflicting regions.

VII. THREATS TO VALIDITY

Construct Validity: Replaying merges and identifying conflicts have been done using the git-based system. Given the broad and extensive use of such system, we believe that errors in measurement during the dataset collection are avoided.

Internal Validity: The empirical evaluation, the pattern recognition, and the domain-specific language built from the patterns are done as a part of a case study on *Microsoft Edge*. Also, the patterns were recognized and tested on 12 weeks of data. Though we believe that learning from the past [1] can have a potential impact on the resolution of merge conflicts, new patterns can emerge in the future, and that can be addressed in two possible ways—one, learning the new pattern from the existing DSL. Second, update the DSL to accommodate the new patterns. A possible threat is that it is possible that some scenarios are not detected by our empirical evaluation in §III due to the 12 week timeline. Another possible threat can be overfitting. To remediate the overfitting, we have taken two steps. First, we built our ranking function to choose smaller and more general programs based on the Occam's Razor principle. Simultaneously, the DSL has to be expressive enough to cover as many tasks as possible while being concise enough to be learned efficiently. Keeping both concerns in mind, we consciously bias our DSL to choose the smaller programs by punishing the longer ones using the ranking function. Second, the overfitting can also be removed by applying more examples for the learning process. For this study, we used 2-3 examples for each pattern. Our solution can be extended to incorporate more examples to remove overfitting if needed in the future. However, since our approach and the DSL has been based on the patterns and common resolution strategies found in the empirical evaluation, there can be scenarios, where our approach may not work and needs re-training. For example, all the fixing strategies are based on a simple fact that the line(s)/Nodes from either the main or fork section of the conflict will be removed or concatenated. There might be scenarios where none of the strategies are applicable—for instance, taking both the main and fork section of the

conflict and changing some parameters. In such scenarios, our approach will not be able to generate any solution. However, this situation can be tackled by extending the underlying implementation of the `Selection` operations in our DSL. Currently, `[Main/Fork]ByPath` works for selecting an include path node. Updating these operations by adding the selection of full AST nodes can help to identify such complex patterns. Also, these patterns can change in the future, e.g., the pattern illustrated in the Frequent Behavior (an example of a project-specific pattern), where due to the different logging module in the main and fork, the main one gets removed. If there is a change in the logging functionality and developers prefer to remove the fork version of the logging, then a new programs need to be generated with new examples.

External Validity: The current results are based on *Microsoft Edge*. Though *Microsoft Edge* is a widely used system, drawing general conclusions from empirical studies in software engineering is difficult because any process depends to a large degree on a potentially large number of relevant context variables [30]. For this reason, we cannot assume a priori that the results of a study generalize beyond the specific environment in which it was conducted [30]. Researchers become more confident in a theory when similar findings emerge in different contexts [30]. Towards this end, we intend that our case study will help contribute towards building a body of knowledge in the area of using program synthesis for merge conflict resolution by replicating across different context variables and environments.

VIII. CONCLUSION AND FUTURE IMPACT

In this paper, we investigate the problem of merge conflicts on a large system. Empirically, our findings identify the type of files, size of conflicts, location of conflicts in a file, and resolution patterns. In our case study, we found that a majority of conflicts in *Microsoft Edge* occur in C++ files, and overall, 28% of the merge conflicts are 1-2 line changes. We empirically characterize the different types of operations that developers perform while resolving merge conflicts. We also identified that a non-trivial section of these 1-2 liner conflicts is due to the `Include` and `Macro` related structural changes.

We propose an expressive domain-specific language to identify the structural merge conflicts in *Microsoft Edge*. The use of program synthesis provides the flexibility to learning complex patterns from a few examples (1-2 examples). Our study highlights that learning project-specific patterns is beneficial, and we are able to address 11.44% of merge conflicts in C++ files with 93.2% accuracy. Applying program synthesis for assisting with merge conflict resolution can be beneficial for two reasons: 1) flexibility to accommodate new patterns, 2) adding explainability to the resulting suggestions.

In the future, we plan to extend this study by applying our approach to other large scale projects. We also plan to explore combining program synthesis with machine learning trained on a large dataset to understand if we can deploy within projects without sufficient history, that is learn from one project and apply to another project - cross project learning.

REFERENCES

[1] G. Ghiotto, L. Murta, M. Barros, and A. van der Hoek, "On the nature of merge conflicts: A study of 2,731 open source Java projects hosted by GitHub," *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 892–915, 2020.

[2] R. P. Buse and W. R. Weimer, "Automatically documenting program changes," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 33–42.

[3] S. Rastkar and G. C. Murphy, "Why did this code change?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1193–1196.

[4] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk, "On automatically generating commit messages via summarization of source code changes," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014, pp. 275–284.

[5] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on software engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[6] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 135–146.

[7] M. Owhadi-Kareshk, S. Nadi, and J. Rubin, "Predicting merge conflicts in collaborative software development," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–11.

[8] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured merge: Rethinking merge in revision control systems," 2011, pp. 190–200.

[9] O. Leßenich, S. Apel, and C. Lengauer, "Balancing precision and performance in structured merge," *Automated Software Engineering*, vol. 22, no. 3, pp. 367–397, 2015.

[10] J. Matsumoto, Y. Higo, and S. Kusumoto, "Beyond gumtree: a hybrid approach to generate edit scripts," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 550–554.

[11] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine, "Dex: A semantic-graph differencing tool for studying changes in large code bases," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, pp. 188–197.

[12] D. Reuling, U. Kelter, J. Bürdek, and M. Lochau, "Automated n-way program merging for facilitating family-based analyses of variant-rich software," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 3, pp. 1–59, 2019.

[13] M. Sousa, I. Dillig, and S. K. Lahiri, "Verified three-way program merge," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–29, 2018.

[14] O. Polozov and S. Gulwani, "Flashmeta: a framework for inductive program synthesis," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015, pp. 107–126.

[15] A. Miltner, S. Gulwani, V. Le, A. Leung, A. Radhakrishna, G. Soares, A. Tiwari, and A. Udupa, "On the fly synthesis of edit suggestions," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[16] GitHub, "GitHub Fork Projects in C++," https://tinyurl.com/yxopll6f, 2020.

[17] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.

[18] M. PROSE, "Microsoft Program Synthesis using Examples SDK," https://github.com/Microsoft/prose, 2020.

[19] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2011, pp. 317–330.

[20] V. Le and S. Gulwani, "Flashextract: A framework for data extraction by examples," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 542–553. [Online]. Available: https://doi.org/10.1145/2594291.2594333

[21] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 404–415.

[22] S. Horwitz, J. Prins, and T. Reps, "Integrating noninterfering versions of programs," *ACM Trans. Program. Lang. Syst.*, vol. 11, no. 3, pp. 345–387, 1989.

[23] C. Sung, S. K. Lahiri, M. Kaufman, P. Choudhury, and C. Wang, "Towards understanding and fixing upstream merge induced conflicts in divergent forks: An industrial case study," *ICSE Software Engineering in Practice (ICSE SEIP)*, 2020.

[24] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.

[25] V. Le and S. Gulwani, "Flashextract: a framework for data extraction by examples," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 542–553.

[26] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: generating program transformations from an example," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 329–342, 2011.

[27] J. Andersen and J. L. Lawall, "Generic patch inference," *Automated software engineering*, vol. 17, no. 2, pp. 119–148, 2010.

[28] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S.-C. Khoo, "Semantic patch inference," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2012, pp. 382–385.

[29] N. Meng, M. Kim, and K. S. McKinley, "Lase: Locating and applying systematic edits by learning from examples," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 502–511.

[30] V. Basili, F. Shull, and F. Lanubile, "Building knowledge through families of experiments," in *IEEE Transactions on Software Engineering*, 1999, pp. 456–473.