Using Pre-trained Language Models to Resolve Textual and Semantic Merge Conflicts (Experience Paper)

Jialu Zhang
Yale University
New Haven, Connecticut, USA

Todd Mytkowicz
Microsoft Research
Redmond, Washington, USA

Mike Kaufman
Microsoft Corporation
Redmond, Washington, USA

Ruzica Piskac
Yale University
New Haven, Connecticut, USA

Shuvendu K. Lahiri
Microsoft Research
Redmond, Washington, USA

ABSTRACT

Program merging is standard practice when developers integrate their individual changes to a common code base. When the merge algorithm fails, this is called a merge conflict. The conflict either manifests as a *textual merge conflict* where the merge fails to produce code, or as a *semantic merge conflict* where the merged code results in compiler errors or broken tests. Resolving these conflicts for large code projects is expensive because it requires developers to manually identify the sources of conflicts and correct them.

In this paper, we explore the feasibility of automatically repairing merge conflicts (both textual and semantic) using *k-shot learning* with pre-trained large neural language models (LM) such as GPT-3. One of the challenges in leveraging such language models is fitting the examples and the queries within a small prompt (2048 tokens). We evaluate LMs and k-shot learning for both textual and semantic merge conflicts for Microsoft Edge. Our results are mixed: on one-hand, LMs provide the state-of-the-art (SOTA) performance on semantic merge conflict resolution for Edge compared to earlier symbolic approaches; on the other hand, LMs do not *yet* obviate the benefits of special purpose domain-specific languages (DSL) for restricted patterns for program synthesis.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**.

KEYWORDS

Resolving merge conflicts, language model, GPT-3, k-shot learning

ACM Reference Format:
Jialu Zhang, Todd Mytkowicz, Mike Kaufman, Ruzica Piskac, and Shuvendu K. Lahiri. 2022. Using Pre-trained Language Models to Resolve Textual and Semantic Merge Conflicts (Experience Paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534396

1 INTRODUCTION

Merge conflicts are common causes of broken pull requests, failure of continuous integration builds, and latent software defects in large projects [9]. Even a single unsuccessful merge can delay the development process from hours to days [10, 22]. It was measured that in large projects developers sometimes spend more time resolving merge conflicts than developing the features [3]. One of the key reasons for this trend is the increasing collaborative environment in large modern software. A recent study showed that with thousands of people and tens of active branches committed to the same code base, nearly 20 percent of all the merge attempts in a large project result in an unsuccessful merge [9].

Table 1: An Example of a Textual Merge Conflict.

Base O	Variant A (Upstream: Chromium)	Variant B (Downstream: Edge)	Resolution R

#include "o.h"	#include "a.h"
	#include "o.h"
		#include "b.h"
		#include "o.h"
			CONFLICT

An unsuccessful merge can originate from either a *textual* merge conflict or a *semantic* merge conflict. A textual merge conflict occurs when two developers edit the same line of code differently. Table 1 shows one such an example. The "git merge" algorithm failed to produce a merge because two independent changes, `#include "a.h"` and `#include "b.h"`, happen in the same location. Normally, in such a situation, a developer must resolve the conflict manually (i.e., after executing the `git merge` command).

Semantic merge conflicts in a divergent fork. In contrast, a semantic merge conflict occurs when there are no textual merge conflicts but nevertheless the merge still results in a broken build, failing tests, or an unintended runtime behavior. Semantic merge conflicts can manifest in all forms of merge attempts, but they often appear in so-called *divergent forks*. A divergent fork is a copy of the source repository, usually created without the intention to merge it back. The standard terminology refers to the source repository as the *upstream* project and a fork is called the *downstream* project. Once created, downstream will have their own independent development history which is rarely merged back to the upstream [22]. For example, Microsoft Edge, Opera and Brave are all based on

This work was performed when Jialu Zhang was an intern at Microsoft Research.

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3534396

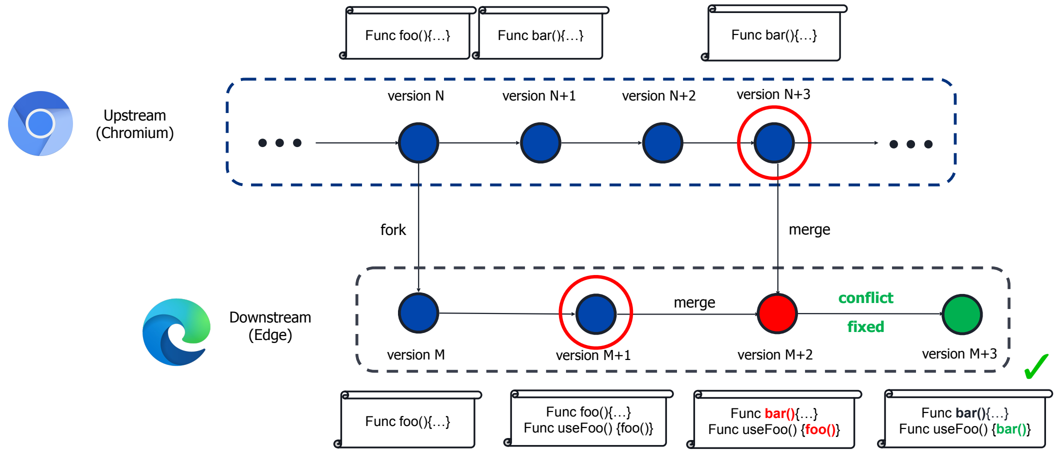

Figure 1: An Example of a Semantic Merge Conflict.

the same upstream (Chromium). Each downstream branch periodically pulls the latest updates from upstream and merges them with other branches in the downstream repository. Using divergent forks saves a large amount of developer's time by reusing functions and classes already defined and tested upstream, expedites the whole development process, and improves the maintainability of the code repository [22].

Despite its obvious benefits, in a divergent fork downstream developers frequently suffer from both textual and semantic merge conflicts. The reason is that upstream and downstream have an independent development processes and thus the downstream can be out of sync with the upstream code changes. To have a better understanding of the size and scalability issues, taking Microsoft Edge as an example, a study showed that over a three-month period there were more than 25,000 upstream commits in Chromium and they all had to be merged downstream [22]. This level of merge frequency makes it difficult or even infeasible for downstream developers to inspect every upstream change before a merge.

Fig. 1 illustrates an example of a semantic merge conflict. A function `foo()` was initially introduced upstream. After the original fork, another function `useFoo()`, which invokes `foo()`, was created downstream. Concurrently, in the upstream branch the function `foo()` was renamed to `bar()`. Later, during the next pull from upstream, there was no textual merge conflict between two versions circled in red. The definition of `foo()` downstream did not change before the merge, so the merging algorithm simply renamed it into `bar()`. Thus, a semantic merge conflict is created, as `foo()` now has no definition in the downstream and will not compile.

A previous study [22] has shown that over 800 commits were identified as attempts to solve semantic merge conflicts in Microsoft Edge. Despite their frequent presence in a modern software development process, semantic merge conflicts cannot be automatically resolved and thus require manual fixes from downstream developers [22]. Repairing semantic merge conflicts is prohibitively expensive. To find a root cause of the conflict, a developer needs to manually inspect the upstream commit history, which, for a large project, can be measured in thousands of commits.

The motivation for this work stems from the need to have a tool that helps programmers in repairing semantic merge conflicts in Microsoft Edge. We developed a tool, named GMerge, that automatically suggests merge conflict resolutions. GMerge takes as input a merge conflict and merge histories from both upstream and downstream. GMerge returns a conflict resolution which indicates which lines of code need to change, and how. In the example from Fig. 1, GMerge automatically suggests that function `foo()` should be renamed to function `bar()`.

Our tool is based on *k-shot learning* with a large language model (GPT-3 [4]). GPT-3 is a large language model that uses deep learning to produce text that looks almost as if it was produced by a human. GPT-3 has been successfully deployed in many applications such as question-answering [14], text completion, source code generation [6] and in many other fields [13, 17]. The biggest difference between GPT-3 and other supervised machine learning models is that the user does not need to train the model specifically for their application. The user only provides a few "shots" (or examples to teach the model) as input to GPT-3 and GPT-3 achieves competitive results, compared to other supervised machine learning models. A *shot* is a standard term used in language models that describes a question/answer pair. Motivated by GPT-3's successful applications in other fields, this paper investigates how to utilize GPT-3's capabilities to resolve merge conflicts. A k-shot learning approach with GPT-3 has significant engineering benefits as it does not require expensive task specific fine-tuning.

To be able to successfully use GPT-3, we needed to address two main challenges: *data curation* and *prompt engineering*. The data curation process automatically extracts source code changes related to merge conflicts. These changes are extracted from both upstream and downstream commits and they are converted into an intermediate representation (IR). Prompt engineering takes data in the IR format and translates the data into input consumable by GPT-3. A key challenge with prompt engineering is that GPT-3's input is limited to 2048 tokens and thus the shot and query must fit within it. To tackle this challenge, GMerge applies string pattern analysis and heuristics in prompt engineering.

We empirically evaluated Gmerge and we have run it on Microsoft Edge semantic merge conflicts. Our evaluation shows that Gmerge learns the correct resolutions at the state-of-the-art (SOTA) 64.6% of accuracy; we use the developer's actual fixes as the ground truth. The evaluation demonstrates the effectiveness of k-shot learning, which provides a cost-effective and language-agnostic solution for real-world semantic merge conflicts.

Generalization to textual merge conflicts in a divergent fork. To establish how easily our approach can be generalized, we conducted a second case study, focusing on textual merge conflicts. The main purpose was to evaluate the effectiveness of our data curation and prompt engineering in this domain. The empirical results show that k-shot learning performance is on par with existing SOTA tools. However, note that those tools still need special purpose domain-specific languages (DSL) for program synthesis. This requires a significant amount of engineering, while our k-shot approach is relatively simpler.

In summary, we make the following contributions:

- We present a data-driven tool Gmerge that uses k-shot learning with a large language model (GPT-3) to automatically find repairs for merge conflicts.
- We propose a method of prompt engineering that translates conflict examples and queries to a small prompt for GPT-3.
- We evaluate Gmerge on both textual and semantic real-world merge conflicts problems. We obtained the state-of-the-art (SOTA) performance on semantic merge conflict resolutions for divergent forks and respectable performance on textual merge conflicts resolutions for divergent forks.

2 MOTIVATING EXAMPLES

In this section we use examples to outline basic ideas of how large language models can repair merge conflicts.

2.1 Semantic Merge Conflict in Divergent Forks

The example in Fig. 1 is not artificially contrived: semantic merge conflicts happen daily. Our first example shows a concrete instance of the merge conflict from Fig. 1, taken from the Microsoft Edge development. The downstream repository contains the following code snippet:

```
if (browser &&
    ...GetBrowserViewForBrowser(browser)->IsIncognito
        ())
    {return;}
```

The merge process for this repository succeeded, but the updated repository failed to build. The compiler returned the following message: "no member named `IsIncognito()` in `BrowserView`". This left the developers confused, because the `IsIncognito()` function was not changed since the last successful merge.

The root cause of this conflict is in the upstream. We could manually execute the `git diff` command on hundreds of commits in the upstream branch, and finally we can find a fragment of the log showing that function `IsIncognito()` was renamed to `GetIncognito()`.

```
...
- bool IsIncognito() const;
+ bool GetIncognito() const;
```

However, clearly this approach is extremely time-consuming. Instead, running Gmerge on this example, it automatically suggests the following repair for the downstream repository:

```
if (browser &&
    ...GetBrowserViewForBrowser(browser)->
        GetIncognito())
    {return;}
```

Gmerge first detects and curates all the relevant upstream changes to a JSON file as the conflict description. We describe the process in Sec. 5.2 – Fig. 5 is the JSON file generated out of this particular example. Gmerge then leverages various heuristics to translate and fit the JSON file into the small prompt input format for GPT-3. The details of the prompt engineering design are given in Sec. 5.3 and Fig. 6 depicts the input to GPT-3 created from the JSON file in Fig. 5.

2.2 Textual Merge Conflict in Divergent Forks

Textual merge conflicts happen frequently in Microsoft Edge development [16, 22]. According to the previous study, more than 1100 textual merge conflicts occurred in Edge's development in a span of three months [22], and each one of these conflicts needs a developers' manual fix. Fig. 2 shows a solution of a real-world merge conflict. The upstream file contains lines 2 and 3, and its corresponding downstream file contains lines 5 and 6 at exactly the same position, which causes a textual merge conflict. When the conflict is reported, a downstream developer had to manually inspect the cause of the conflict. The developer also had background knowledge of the whole project and knew that the header file `url_utils.h` in the forked branch has the same content as the header file `google_util.h` in the main branch. To resolve this issue, the developer kept the one in the forked branch and excluded the one in the main branch. The developer also kept both lines 3 and 6, since they are referring to two different header files.

Gmerge is able to resolve textual merge conflicts because we teach GPT-3 how historical textual merge conflicts were resolved with the examples in the prompt. We describe the details of this process in Sec. 6.3. Running Gmerge on this example automatically produces the same resolution shown below.

```
1 Upstream (Chromium):
2 - #include "components/google/core/common/google_util.h"
3 + #include "components/variations/net/omnibox_http_headers.h"
4 Downstream (Edge):
5 + #include "components/microsoft/core/common/url_utils.h"
6 + #include "components/variations/edge_features.h"
```

Figure 2: An example of a textual merge conflict resolution in a divergent fork. The line that starts with "+" means this line is kept in the final merged code. The line that starts with "-" means this line is removed from the final merged code.

3 BACKGROUND

In this section, we provide the necessary background for k-shot learning, which is the basis of our tool. We first give a general overview of k-shot learning and the following sections demonstrate how to apply k-shot learning to automate merge resolution.

Language model. A *language model* is a probability distribution over a sequence of words. A classic application of a language model is to predict the next token in some text. Language models have been used for sentiment analysis[18], question and answering[26], or even code generation[6].

Generative Pre-trained Transformer 3 (GPT-3). *GPT-3* is a large language model that is famous for its ability to produce general, human-like text [4]. It was trained on a massive corpus of unlabeled text, including Common Crawl (410 billion tokens) and Wikipedia (3 billion tokens). GPT-3 is now generally available to the public through its API[1].

GPT-3 has been successfully deployed in many applications ranging from traditional NLP tasks such as question answering and text completion [14], to many other fields such as poem writing, and source code generation [6, 13, 17]. A key benefit of GPT-3 is its ability to adapt to novel input formats without re-training the underlying model. Often, a user of GPT-3 can exploit the "text-in", "text-out" nature of the GPT-3 interface to teach the model with examples of what the user is looking for. These examples are called "shots".

Prompt, Shot and Query. A *prompt* describes the input to GPT-3. It has two components: one or more shots and the user's query. The distinction between the two is best described with an example:

```
Question: Apple?
Answer: Red

Question: Eggplant?
Answer:
```

In the example above, the shot (first two lines) teaches the model with a "Question" and "Answer" pair. Likewise, the query follows the same pattern but prompts the model to complete the "Answer". GPT-3 is not explicitly trained on this "Question" and "Answer" pair and yet it is often capable of completing rather complex prompts.

K-shot learning. In the prior example, we only showed a single shot to the model. In contrast, if the prompt consists of k shots, it is called k-shot learning. In general, significant effort goes into developing prompts (and the corresponding shots) for complex tasks.

Zero-shot learning. Zero-shot learning is similar to k-shot learning, with the exception that no demonstration is allowed and the model is only given a brief task directive in the prompt. For example, consider a zero-shot prompt:

```
Question: What is the capital of Spain?
Answer:
```

Zero-shot learning is challenging because it requires the model to understand the task without any examples to teach it. Despite this challenge, often a model having the size of GPT-3 can still accurately answer such a prompt. In this example, the intuition is that the word “Madrid” has the highest probability given the words of sequence - “What is the capital of Spain?” in its massive training corpus.

4 SYSTEM OVERVIEW

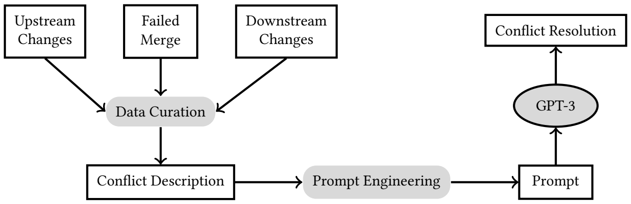

Figure 3: GMERGE Overview

The overview of the GMERGE tool is shown in Fig. 3. It takes as input three parameters: a merge conflict, and both commits in the upstream and downstream repositories that constitute the merge. As output, GMERGE returns a merge conflict resolution.

GMERGE contains two main modules: data curation and prompt engineering.

The data curation module takes as input the upstream and downstream commits along with the downstream semantic merge conflict including the compiler error messages. It generates JSON files as the intermediate representation (IR) for the prompt engineering module. We call this file a *conflict description*. This paper contains two case studies, but data curation is extensively used only in the first case study. However, when we run GMERGE on an existing benchmark where the conflicts have already been curated, GMERGE skips this module and goes directly to the next module.

The purpose of the prompt engineering module is to translate a conflict description file into the small prompt format required as input for GPT-3. The prompt engineering is a technical module and for each of our case studies we needed to apply various heuristics, described in more detail in Sections 5.3 and 6.3. This is due to the fact that conflict description files for each case study have different format.

In the next two sections, we describe applying GMERGE in two different scenarios.

5 CASE STUDY 1: RESOLVING SEMANTIC MERGE CONFLICTS FOR A DIVERGENT FORK

5.1 Insights: Applying k-shot Learning to Resolve Semantic Merge Conflicts

In GMERGE, we use GPT-3 to generate semantic merge conflict resolution via k-shot learning. Fig. 4 shows the prompt that we provide to the GPT-3. In the demonstration (line 1 to line 8), we teach the model with an example of how the historical merge conflict was resolved. In the following task directive (line 11 to line 17), we provide the model of the set of relevant changes from upstream and the conflict to be fixed in downstream. The output of the model is the resolution of the targeted merge conflict.

[1] https://openai.com/api/

```
1  Question:
2  <The historical before changes from upstream>
3  <The historical after changes from upstream>
4
5  <The historical conflict line in downstream>
6
7  Answer:
8  <The ground truth resolution provided by developers>
9
10 Question:
11 <The current before changes from upstream>
12 <The current after changes from upstream>
13
14 <The conflict to be fixed in downstream>
15
16 Answer:
```

Figure 4: An intuitive example of how to use GPT-3 to resolve semantic merge conflicts. The shot is in line 1 to line 8. The query starts from line 10.

5.2 Data Collection and Curation

Merge conflicts in downstream forks are often introduced by commits made in the upstream repository, so the goal of the data collection and curation process is to identify and extract all the source code changes in upstream that are related to the given merge conflict. In real-world software development, this is also the first step that the programmer manually performs in trying to resolve a failed merge. For large upstream repositories such as Chromium (of which Microsoft Edge is a downstream divergent fork), searching through the thousands of upstream commits is a tedious and error-prone task. In Gmerge we automate this whole process.

Gmerge takes the commit logs of the repository, the semantic merge conflicts including the compiler error messages as the inputs, and outputs a JSON file for each semantic merge conflict. This JSON file contains all code-level changes relevant to the target merge conflict. The files are designed to be self-contained, in the sense that it is sufficient to check the JSON file to gain all the information relevant for the given semantic merge conflicts. Each file has three key components: 1) the set of relevant changes from upstream, 2) the conflict to be fixed in downstream, and 3) the resolution provided by the real-world developers. The third component is only relevant for establishing the ground truth for our evaluation.

Fig. 5 shows an example JSON file that Gmerge generated for the merge conflict given in Sec. 2.1. In Fig. 5, line 2 to line 12 are the relevant changes from upstream. Line 13 is the merge conflict and line 14 is the resolution provided by the Edge developers.

How to generate the JSON file from the compiler error? In Gmerge, we localize the blamed upstream commits by tracing the compiler error message to identify the relevant keywords and prune the space of source code changes in the upstream based on these keywords. For example, shown in Sec. 2.1, Clang results in the error message "Cannot find the definition for function `IsIncognito()`." This is a typical error message for a semantic merge conflict due to function renaming. From this message, Gmerge identifies `IsIncognito()` as the keyword relevant to this error message.

Gmerge automatically extracts the keywords for each semantic merge conflict; it starts by extracting the strings in the quotation marks in the error message. To be conservative, if the keyword has a C++ class specifier in it as the prefix, e.g., `browser::IsIncognito()`, we collect both its type specifier `browser` and the function name `IsIncognito()` as the keywords. These keywords are used as the seeds for further search in the upstream commits. Gmerge extracts every deleted line in the upstream code diff that contains the keywords. Gmerge additionally extracts the lines following immediately after the deleted line, which typically contain the new name of the renamed symbol. In Gmerge we use a heuristic that collects all the lines until we reach an unchanged or deleted line.

```
1  {
2      "UpstreamChanges": [
3          {
4              "Before": "const␣bool␣incognito␣=␣
                   browser_view_->IsIncognito();",
5              "After": "const␣bool␣incognito␣=␣
                   browser_view_->GetIncognito();"
6          },
7          {
8              "Before": "bool␣IsIncognito()␣const;",
9              "After": "bool␣GetIncognito()␣const;"
10         }
11         ...
12     ],
13     "DownstreamConflict": "BrowserView::
           GetBrowserViewForBrowser(browser)->
           IsIncognito())",
14     "DownstreamFix": "BrowserView::
           GetBrowserViewForBrowser(browser)->
           GetIncognito())"
15 }
```

Figure 5: The JSON example generated by Gmerge for the merge conflict in Sec. 2.1.

For each semantic merge conflict manifested in downstream, we extract the line of code that has a compiler error. To obtain the user resolution (ground-truth) for such a merge conflict, we identify the repair commit in downstream that follows the conflict that 1) changed the same line in the same file as the semantic merge conflict and 2) the compiler error for that specific merge conflict no longer existed after that commit. Similar to upstream, we collect the repair code region in downstream in the same way.

5.3 Prompt Engineering

The prompt engineering module applies various heuristics to the JSON description of the conflicts in order to translate both merge conflict examples and queries into succinct prompts for GPT-3. The output of GPT-3 is the resolution of the merge conflict.

Prompt structure. When resolving semantic conflicts, Gmerge has two prompt structures: one-shot and zero-shot. Both of them create the same task directive in the prompt. The only difference is that in zero-shot learning no demonstration is given. Fig. 6 is the real-world example of how we use one-shot learning to resolve the merge conflict in Sec. 2.1.

Prompt format. In the prompt, the lines that start with a double `--` and `++` represent the conflict-related changes in upstream. The line starting with a single `-` appears only at the end of the query and it represents the merge conflict in downstream. The line starting with a single `+` is not in the prompt. Instead, it is the output of GPT-3 for the resolution of the merge conflict.

```
1  Question:
2  --web_app_info->app_url = url;
3  ++web_app_info->start_url = url;
4
5  -web_app_info->app_url = url;
6
7  Answer:
8  +web_app_info->start_url = url;
9
10 Question:
11 --if (browser_view_->IsIncognito()||!browser_view_->
        IsBrowserTypeNormal())
12 ++if (browser_view_->GetIncognito()||!browser_view_
        ->GetIsNormalType())
13
14 --bool IsIncognito() const;
15 ++bool GetIncognito() const;
16
17 - ...GetBrowserViewForBrowser(browser)->IsIncognito
        ())
18
19 Answer:
20 + ...GetBrowserViewForBrowser(browser)->GetIncognito
        ())
```

Figure 6: An example of merge conflict resolution using one-shot learning in GPT-3. The shot is represented in line 1 to line 8. The query starts from line 10 and ends on line 19. Line 20 is not in the prompt, and it is the output of GPT-3 for the resolution of the merge conflict.

Prompt content. GPT-3 has a fixed input size, 2048 tokens. This is in sharp contrast to the massive code diffs of thousands of commits. A single JSON file could contain thousands of lines of code changes related to a merge conflict. To leverage the power of GPT-3, one of the key technical challenges in GMERGE is to fit the examples and the queries into this small prompt.

We adopt a heuristic to ensure that we prioritize diverse representations of "UpstreamChanges" (in the JSON file) in the prompt. Our intuition is that we want each pair of "UpstreamChanges" to have a distinct string `edit sequence`. Each `edit sequence` is a list of operations that are applied to strings. Applying the `edit sequence` on the first string produces the second one. There are three kinds of operations, the addition `+`, the deletion `-` or the replacement `|` in the edit. We omit the space padding in our `edit sequence` definition.

We use the Python library `difflib.ndiff` to compute the string difference. GMERGE ensures that every selected `--` and `++` string pairs in the prompt has a distinct `edit sequence` pattern. In this way, we managed to fit the shots and the queries into the small prompt (2048 tokens). In Sec. 5.4, we demonstrate the effectiveness of our prompt content design.

5.4 Evaluation

We evaluate the efficacy of GMERGE on semantic merge conflicts by answering the following questions:

(1) How does GMERGE resolve semantic merge conflicts in divergent forks?
(2) Does prompt engineering positively affect the accuracy?
(3) Are larger language models more accurate than smaller ones?

Table 2: Evaluation of GMERGE with baselines on merge conflict resolution accuracy.

	Accuracy
Gmerge (GPT-3)	**64.6% (245/379)**
Gmerge (GPT-J)	39.1% (148/379)
StringMerge	30.1% (114/379)
Transformation.Text	25.9% (98/379)

Experiment setup. We collected all Edge merge conflicts from Aug 2020 to April 2021 wherein one of four clang compiler messages occurred. The four selected types of error messages cover more than 70 percent of semantic merge conflicts. The rest of the errors constitute a long tail. These four errors are:

- `err_no_member`
- `err_no_member_suggest`
- `err_undeclared_var_use_suggest`
- `err_undeclared_use_suggest`

In total, we obtained 379 semantic merge conflicts. We process each conflict into a JSON file for downstream evaluation. The data are available at: https://doi.org/10.5281/zenodo.5911767.

In concert, we extract the actual user-defined fix for each conflict and use that as ground truth for GMERGE's suggested fix. In particular, it will be assumed that GMERGE suggests the correct fix if the user-defined fix is a prefix of our model generated solution. We adopted this metric mainly because GPT-3 is an auto-regressive model, which often outputs text up to a fixed pre-specified length. This was the internal design for GPT-3 to complete existing text at the time when the paper was written[2]. Although the prefix metric might allow extra tokens after the correct resolution, we sampled our generated resolution and noticed that it did not happen very often. This prefix metric is also accepted by the Edge developers since the actual fix by users (ground truth) is the prefix of our resolution and humans can remove the extra tokens when surfaced as a suggestion. We ran all experiments on a Windows VM with an Intel i7 CPU and 48 GB of RAM.

Can GMERGE resolve semantic merge conflicts in divergent forks? Table 2 shows the performance of GMERGE on our dataset. GMERGE has an overall accuracy of 64.6% after ten model trials. Table 2 also includes a comparison of GMERGE to three baselines. We first compared GMERGE to a heuristic-based approach, `StringMerge`. Following this, we compared GMERGE to the state-of-the-art string-based program synthesis approach [24], `Transformation.Text`. Finally, we evaluated how the choices of language model (GPT-3 and GPT-J) affected the results.

Our first baseline `StringMerge`, is a heuristic-based approach designed by analyzing patterns in the upstream and then using those patterns to generate merge conflict resolutions in the downstream. `StringMerge` implements the main algorithm described in MrgBldBrkFixer [22] but operates on and generates merge conflict resolutions as strings rather than ASTs (we use strings because we do not have access to changes that might not be parsable into ASTs). For each conflict, `StringMerge` identifies the symbol that is responsible for the

[2] Recently, a new GPT-3 version https://openai.com/blog/gpt-3-edit-insert/ was released that allows for insertions and edits to the input text. We leave exploring this direction in our future work.

conflict by checking compiler error messages. It then infers a set of possible function/class/type renaming patches on the symbol by computing the textual difference between two strings. Finally, for each patch, `StringMerge` applies it to the downstream string to obtain a possible conflict resolution.

Note that the high-fidelity static analysis based approach (e.g. based on clang) is not possible in our setting due to the fact that it needs the Edge project to be built, and we only have access to the project file contents. This baseline `StringMerge` is the closest we get to a static analysis baseline and it mimics the way Edge developers manually fix such a merge conflict in the real world. Table 2 shows `StringMerge`'s performance. GMERGE performs better in terms of resolution accuracy (64.6% vs 30.1%).

Our second baseline, `Transformation.Text` is the state-of-the-art program synthesis approach specialized in string transformation [24]. `Transformation.Text` takes several pairs of strings as examples. It then synthesizes a program that takes the first string in each pair as input and outputs the second string. After that, the synthesized program is used to produce the transformation on unseen strings.

The intuition of using `Transformation.Text`, such a SOTA program synthesis approach to generate merge conflict resolution is that the upstream code changes could be represented as examples in a programming-by-example based synthesis approach. After learning such a program, `Transformation.Text` outputs its transformation on the downstream string as the conflict resolution.

Our evaluation shows that `Transformation.Text` is able to generate resolutions. However, GMERGE performs much better in terms of resolution accuracy (64.6% vs 25.9%) and surprisingly even the heuristic-based `StringMerge` outperforms it. One possible reason is that `Transformation.Text` is a generic string transformation tool, so the pattern in semantic merge conflict resolution is too complex for it to learn. Fig. 9 is such a challenging example that is difficult for the existing SOTA program synthesis approach to resolve.

Our third baseline is actually GMERGE itself, but with a different language model, namely GPT-J[25]. It is introduced to evaluate how the size of the language model affects the result. GPT-3 and GPT-J have similar architectures, but GPT-3 has 175 billion parameters while GPT-J has 6 billion. Our evaluation shows that the size of the model indeed affects its ability to resolve semantic merge conflicts. GMERGE performs much better than GMERGE (GPT-J) in terms of resolution accuracy (64.6% vs 39.1%). GMERGE (GPT-J) has performed better than `StringMerge` and `Transformation.Text`. This shows that resolving semantic merge conflicts is a non-trivial problem, and a large language model is able to automatically generate a resolution for semantic merge conflicts with high accuracy.

Ablation study over GMERGE. We now present results on experiments to analyze different design choices in GMERGE. Table 3 shows how prompt engineering affects the accuracy of merge conflict resolution in GMERGE. One of the advantages of GPT-3 is that it only needs a few examples (shots) for "training". One key question, however, is what impact does the shot have on accuracy? We evaluated GMERGE in two prompt structures: one shot and zero shot. In our setting, zero-shot learning is identical to one-shot learning, except no example is allowed and the model is given the same task directive in the prompt.

Table 3: Ablation study over GMERGE. Evaluation for different prompt designs in GMERGE.

	First Pair	Maximal Test (without heuristics)	**Maximal Test (with heuristics)**
One-shot	44.1%	60.4%	**64.6%**
Zero-shot	33.2%	35.1%	40.0%

One of the major technical challenges in GMERGE is to fit the shot and the query into the fixed-sized prompt. We have evaluated GMERGE on three different prompt structures in Table 3. “First pair” means that we only choose the first conflict-related source code change in the JSON file to form the query. “Maximal Test (without heuristics)” takes as many changes as possible in the original sequence until the size of the prompt reaches its limit. “Maximal Test (with heuristics)” takes the heuristics described in Sec. 5.3 as the filtering method to prioritize some changes in the prompt. Each of these prompt representations differs in size and content and thus this section investigates their impact on model accuracy.

The evaluation shows that having a shot as the input to the language model significantly improves the results in all prompt structures. This meets our expectations because having a shot not only clearly pinpoints the current task to the model but also provides an example of what is the expected output from the model. Moreover, the evaluation shows that providing more conflict-related code changes as the context improves the accuracy of the model. It further shows that with the heuristics, GMERGE achieved the highest accuracy of 64.6%.

We found that increasing the number of shots used in our evaluation did not achieve better accuracy. The reasons are that 1) In this case study, shots teach GPT-3 and show how the output format should be. The content in the shot is not designed to be relevant to the actual merge conflict. Based on our experience, having more shots will add noise in the prompt and it might misguide the model. 2) The prompt size is very limited to only 2048 tokens (in GPT-3). Therefore, we chose the most compact shot (line 1 to line 8 in Fig. 6) in our setting to save the token space for providing more context (related code changes in upstream and downstream) to GPT-3 in the task directive.

Are larger language models more accurate than smaller ones? GPT-3 and GPT-J have similar architecture designs, but different size of the models. In this section, we show that the size of the model has a significant impact on GMERGE's task specific accuracy.

GPT-3 and GPT-J each output one resolution at one model trial. In our experiment, we repeatedly query the language models, and if the resolution is produced in any of the trials, we mark the merge conflict as “resolved”. The intuition here is that each resolution is a possible fix and because a query is inexpensive (in comparison to a developer fixing it) we can help automate the costly process of merge resolution by using the model to synthesize these fixes. To that end, we evaluate how the number of trials affects the model accuracy. Fig. 7 shows that the overall accuracy of GPT-3 and GPT-J both increased with the number of model trials, but the overall accuracy of GPT-3 increased more sharply than GPT-J with the increase of model queries. For example, for GPT-3, having ten independent trials achieves the accuracy of 64.6% in contrast to the accuracy of 37.2% with only one trial. Compared to the GPT-3 model, we only

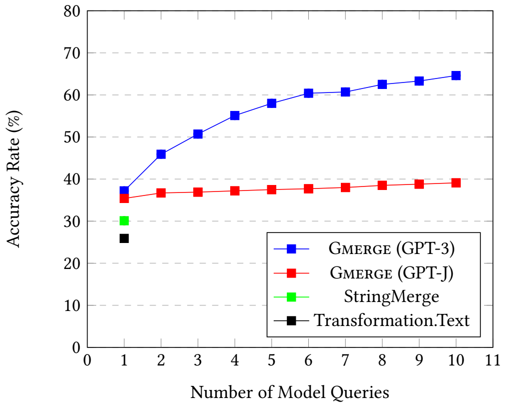

Figure 7: Accuracy comparison for GMERGE on resolving semantic merge conflicts.

observed a modest accuracy gain in the GPT-J model. `StringMerge` and `Transformation.Text` have no accuracy gain because they produce a deterministic result in every run.

Based on what we observed from the result, a larger size of the language model has better overall accuracy on both settings: one single model trial or multiple model trials. The larger model is more likely to produce extra merge conflict resolution when multiple numbers of model trials are allowed.

5.5 More Than Renaming: Challenging Examples Resolved by GMERGE

The motivating example from Sec. 2.1 was resolved using function renaming. However, in a real-world setting, the semantic merge conflicts that downstream developers face are not limited to simple function/class/type renaming. In this subsection, we illustrate the complexity of real-world merge conflicts that downstream developers face in daily development. A key observation is that the patterns here are very difficult to write down as general cases, but the fixes are intuitive once viewing the code. GPT-3 with k-shot learning is able to extract these intuitive fixes where other more rigid pattern-based approaches fail. We closely inspect the example of a broken build taken from the Edge repository.

A challenging but typical merge conflict example is shown in Fig. 8, after a merge, the compiler sent an error message indicating that it could not find a definition for `PermissionRequestType`. However, `PermissionRequestType` has not been changed before the last successful merge in the downstream.

```
- PermissionRequestType::PERMISSION_CAMERA_PAN_TILT_ZOOM:
```

Figure 8: A challenging but typical semantic merge conflict in Microsoft Edge. This line has not been changed since last successful merge attempt. However, after the merge, the compiler can not find the definition for `PermissionRequestType`.

To correctly resolve such a merge conflict, developers need to learn how the upstream context has changed from the last successful merge attempt to this specific merge and then apply the similar changes to the downstream context. To derive this particular resolution, it was not enough to find the relevant file in the commit history and then propagate the changes: the developer needed to find three different files and manually combine the changes described in those files to derive the required resolution. We list the most relevant changes in Fig. 9.

```
- permissions::PermissionRequestType::PERMISSION_NOTIFICATIONS
+ permissions::RequestType::kNotifications

- permissions::PermissionRequestType request_type
+ permissions::RequestType request_type

- permissions::PermissionRequestType::PERMISSION_NOTIFICATIONS)
+ if (request_type == permissions::RequestType::kNotifications)
```

Figure 9: Upstream changes related to the conflict in Fig. 8.

```
+ RequestType::kCameraPanTiltZoom:
```

Figure 10: The conflict resolution given by Edge downstream developers to the conflict in Fig. 8.

The developers first detected that the root cause of this compiler error was due to fact that `PermissionRequestType` has been renamed to `RequestType` in upstream. However, applying these renaming changes to the `PermissionRequestType` in downstream still did not resolve this conflict. This was because the `PERMISSION_NOTIFICATIONS` in upstream was changed to `kNotifications`. After the developer learned how the upstream context changed and applied similar changes to the downstream context, the merge conflict was resolved by changing `PERMISSION_CAMERA_PAN_TILT_ZOOM` to `kCameraPanTiltZoom`. The correct resolution to this particular problem is given the line annotated with + in Fig. 10 and this was a repair that the developer committed. GMERGE managed to automatically learn not only the renaming cases to `RequestType` but also learn and apply the related context change near the `RequestType` in the downstream to produce the exact resolution as developers did.

The existing merge conflict resolution approaches [16, 22, 24] are not helpful in such cases, because their learning algorithms are limited when it comes to learning and combining complex string transformation. Indeed, in our evaluation, given in Sec. 5.4, we show that none of our baseline methods could resolve this merge conflict. With the prompt shown in Fig. 11, GMERGE generated line 25 as the resolution to this conflict.

5.6 Discussion

Practical value and tool development. GMERGE achieved the overall accuracy of 64.6% in solving semantic merge conflicts in Microsoft Edge. The practical value of GMERGE lies in its ability to

```
1  Question:
2  --web_app_info->app_url = url;
3  ++web_app_info->start_url = url;
4
5  -web_app_info->app_url = url;
6
7  Answer:
8  +web_app_info->start_url = url;
9
10 Question:
11 ...
12
13 --"request", permissions::PermissionRequestType::
        PERMISSION_NOTIFICATIONS,
14 ++"request", permissions::RequestType::
        kNotifications, requesting_origin);
15
16 --permissions::PermissionRequestType request_type,
17 ++permissions::RequestType request_type,
18
19 --permissions::PermissionRequestType::
        PERMISSION_NOTIFICATIONS) {
20 ++if (request_type == permissions::RequestType::
        kNotifications) {
21
22 - case permissions::PermissionRequestType::
        PERMISSION_CAMERA_PAN_TILT_ZOOM:
23
24 Answer:
25 + case permissions::RequestType::kCameraPanTiltZoom:
```

Figure 11: The merge conflict resolution to the conflict in Fig. 8. Line 1 to line 8 are the shot, line 10 to line 24 are the query to GPT-3. Line 25 is not included in the prompt, and it is the output of GPT-3 for the resolution of the conflict.

automate 64.6% of such merge conflicts. To the day, Gmerge is on the process of being productized in Microsoft Edge development.

Automation level. The motivation of our work is in the fact that it often costs developers a large amount of time to manually identify the source of conflicts and correct them. Because of this, automation is important in resolving semantic merge conflicts for a divergent fork. MrgBldBrkFixer [22] also investigated the feasibility of fixing the semantic merge conflicts in Microsoft Edge.

However, MrgBldBrkFixer is not fully automated, contrary to Gmerge. It needs downstream developers to manually classify the semantic conflicts and assign a type for these conflicts. Therefore, MrgBldBrkFixer still requires manual labor in resolving merge conflicts. This is the reason why we cannot directly use MrgBldBrkFixer as the baseline in our evaluation. We manage to automate the major algorithm described in MrgBldBrkFixer [22] into our first baseline: `StringMerge`, and we include the result comparison in Table 2.

Limitations. Gmerge cannot provide any guarantee to the resolution of merge conflicts. Gmerge relies on Clang compiler message to locate the conflict-related changes in upstream. Therefore, if the compiler fails to pinpoint where the error is or the git commit history is not accurate, Gmerge cannot handle such merge conflicts.

Future directions. We envision Gmerge in its current version to resolve merge conflicts related to compiler errors in Microsoft Edge. Encouraged by our results, generating test failure resolution is a promising future direction to pursue. The lack of compiler errors in the merged code is the prerequisite for the execution of unit and integration tests. Also, from the existing work [22], only 9.9% of merge conflicts in Edge are from test failures. Moreover, the issue of root causing the fixes for test failures (for ground truth) is a non-trivial exercise. Therefore, Gmerge serves as our first step in resolving merge conflicts related to compiler errors in Edge, but we plan to address the test failure resolution in our future work.

```
1  Question:
2  <<<<<<< HEAD
3  The historical conflict region in upstream
4  =======
5  The historical conflict region in downstream
6  >>>>>>>
7
8  Answer:
9  The ground truth resolution provided by the
        developers
10 ...
11
12 Question:
13 <<<<<<< HEAD
14 The current conflict region in upstream
15 =======
16 The current conflict region in downstream
17 >>>>>>>
18
19 Answer:
```

Figure 12: An intuitive example of how to use GPT-3 to resolve textual merge conflicts. Shot(s) are shown in line 1 to line 10. The query starts from line 12.

6 CASE STUDY 2: RESOLVING TEXTUAL MERGE CONFLICTS FOR A DIVERGENT FORK

6.1 Insights: Applying k-shot Learning to Resolve Textual Merge Conflicts

Similar to semantic merge conflict, in Gmerge, we use GPT-3 to generate textual merge conflict resolution via k-shot learning. Fig. 12 shows the prompt that we provide to the GPT-3. In the demonstration (line 1 to line 10), we teach the model with examples of how the historical merge conflicts were resolved. In the following task directive (line 12 to line 19), we provide the model with the current conflict region in upstream and downstream respectively. The output of GPT-3 is the resolution to the merge conflict.

6.2 Study Setup: Benchmark Description

The problem of textual merge conflict resolution for divergent forks has been studied in the paper [16] and the benchmark is publicly available. This benchmark is collected from the Microsoft Edge development repository in an eight-week period (March 30 2020 to April 24 2020). For each merge conflict in the benchmark, it is either a C++ header file conflict or a Macro related conflict, and the size of the merge conflict has up to two lines of changes. Each conflict has three parts, the upstream conflict region, the downstream conflict region and the ground truth: the actual resolution by developers. In total, this benchmark has 122 textual merge conflicts due to header file conflicts and 38 conflicts due to Macro conflicts.

Note that in this benchmark, the repositories that contain the merge conflicts are not included and released, thus making Gmerge

```
1  Question:
2  <<<<<<< HEAD
3  #include "chrome/browser/ui/views/accessibility/
        caption_bubble_controller_views.h"
4  =======
5  #include "chrome/browser/ui/views/accessibility/
        hc_with_theme_bubble_view.h"
6  >>>>>>>
7
8  Answer:
9  #include "chrome/browser/ui/views/accessibility/
        caption_bubble_controller_views.h"
10 #include "chrome/browser/ui/views/accessibility/
        hc_with_theme_bubble_view.h"
11
12 Question:
13 <<<<<<< HEAD
14 #include "base/notreached.h"'
15 =======
16 #include "base/metrics/histogram_macros.h"'
17 >>>>>>>
18
19 Answer:
20 #include "base/metrics/histogram_macros.h"'
21 #include "base/notreached.h"'
22
23 Question:
24 <<<<<<< HEAD
25 #include "build/build_config.h"
26 =======
27 #include "media/media_buildflags.h"
28 >>>>>>>
29
30 Answer:
```

Figure 13: An example of prompt in textual merge conflict resolution by Gmerge.

infeasible to do the data curation. We run Gmerge on the existing benchmark [16] to resolve merge conflicts using only the prompt engineering module.

6.3 Prompt Engineering

Using the prompt format shown in Fig. 12, we evaluated how different selection strategies of demonstrations affected the final result in Gmerge. We have two prompt engineering methods here. First, we randomly selected a few examples from each category to form the shot. We adopted the same categories used in the existing work [8] to handle textual merge conflicts. This resulted in five header file examples and two Macro examples in the shot. We named this prompt engineering method as the "Randomly selected shots". Second, we use our domain-specific knowledge to pick two typical examples as the shot. We name this prompt engineering method as the "Representative shots".

Fig. 13 shows an example of using the "Representative shots" method to resolve textual merge conflicts in Gmerge. Line 1 to line 10 is the first shot, and line 12 to line 21 is the second shot. The task directive (the target merge conflict) is shown from line 23 to line 30. In Gmerge, it outputs:

```
#include "build/build_config.h"
#include "media/media_buildflags.h"
```

and this is the exact resolution provided by the Edge developers.

Table 4: Merge conflict resolving accuracy for Gmerge on Edge header file and Macro textual merge conflict dataset.

	SOTA [16]	Randomly Selected Shots	Representative Shots
Header File	91.8%(112/122)	49.6% (58/117)	60.0% (72/120)
Macro	94.4%(35/38)	100% (36/36)	100% (36/36)

6.4 Evaluation and Discussion

Table 4 shows the accuracy of resolution on Edge header file and Macro textual merge conflict dataset. The prompt engineering method "Representative shots" has fewer shots than the "Randomly selected shots" but it has slightly better accuracy than "Randomly selected shots". This shows the importance of prompt engineering in using language models to resolve merge conflicts, including selecting the most effective demonstrations.

We did evaluate the zero-shot learning in our textual conflict at first, but the result is largely worse. This is expected because GPT-3 does not have any insights about the task if only few lines of the conflict regions are given in the task directive. Also, conflict markers (namely strings <<<<<<< and >>>>>>>) are seldom seen in GPT-3's training set, making GPT-3 even more difficult to output meaningful resolution.

Compared to the existing SOTA work [16], which requires a careful design of domain-specific language, our tool Gmerge has better accuracy on Macro related textual merge conflicts. For header file related merge conflicts, Gmerge has a modest accuracy of 60.0%. This is mainly because Gmerge does not have the domain-specific knowledge for the repository in the input, which is used to resolve header file related merge conflicts. Fig. 14 is such an example that can only be resolved by using prior domain-specific knowledge.

```
Upstream (Chromium):
 + #include “base/notreached.h”
Downstream (Edge):
 - #include “base/logging.h”
 + #include “base/mojom/scoped_native_library.h”
```

Figure 14: A merge conflict that cannot be resolved without prior domain-specific knowledge. The "base/logging.h" should always be removed from the merge resolution because Edge uses a different logging system.

To resolve such a merge conflict, the existing solution [16] crafted a new domain-specific language that captures the patterns from historical data as resolution strategies, and used program synthesis to learn such repeated resolutions. Applying the learned strategies to the new unseen merge conflicts will automatically synthesize a resolution. However, without access to the historical data of the full repository, the following " always deleting `logging.h`" pattern cannot be inferred by Gmerge.

Gmerge uses a pre-trained GPT-3 model to resolve merge conflicts, which has significant engineering benefits as it does not require expensive, task-specific fine-tuning. However, for some specific types of textual merge conflicts, for example, the conflict resolution is strictly composed of lines from the input [8], fine-tuning the model could be a promising direction to pursue. We plan to explore this possibility in future work.

In summary, this case study shows that GMERGE has competitive performance on problems where current SOTA approaches require special-purpose domain-specific languages (DSL) for program synthesis. It highlights that language models still do not obviate the need for domain-specific investments for the merge conflict problem. Furthermore, even when the data curation is not feasible due to the lack of repository information, we show that prompt engineering is still useful to improve the accuracy of GMERGE.

7 RELATED WORK

Semantic merge conflict. Semantic merge conflicts occur when the merged code cannot be successfully compiled. This problem was first introduced by Horwitz *et al.* [12] and later formalized by Yang *et al.* [28] in the 1990s. Recent studies [7, 9] have shown that such a bad merge in the code delayed the development cycle or caused damage by simply leaving bugs in the code. As a result, semantics merge conflict detection [21] and resolution approaches [19, 22] have been proposed. FSTMERGE [2] is the first semi-structured merge tool. JDIME [1] automatically tunes a semi-structured merge based conflict locations detection and resolution. NLX_REG [17] uses a large language model to synthesize regular expressions. SAFEMERGE [21] prevents merge conflicts by defining formal specifications for the based code, variants of the code and the final merged code. However, it did not directly produce the merge conflict resolutions as GMERGE does.

The closest work to ours is the MrgbldBrkFixer [22]. It analyzes the AST diffs for changes in the upstream to construct a patch for merge conflicts. However, MrgbldBrkFixer still requires developers' manual work to classify the build breaks, and the tool heavily relies on the AST analysis for C++ code only. In contrast, GMERGE is scalable, fully automated and language-agnostic by leveraging large scale language models.

Textual merge conflict. Textual merge conflicts have been long known as a severe and challenging problem, as reported in prior studies [9, 16]. As a result, textual merge conflict mining and detection approaches [1, 5, 11, 15] have been proposed. Going one step further than bad merge prevention [27], we have recently witnessed great progress via program synthesis[16] and machine learning [8] to directly resolve merge conflicts. Deepmerge [8] and MergeBERT [23] required customized and expensive machine learning models. Pan *et al.*[16] studied the historical data of bad merge to design special purposed domain-specific languages (DSL) for program synthesis to a single C++ project. IntelliMerge [20] studied refactoring caused merge conflicts in software development and evolution in Java programs.

8 CONCLUSION

In this paper, we explored the feasibility of leveraging k-shot learning with large language models for resolving various merge conflicts (both textual and semantic). Our results demonstrate that language models have the potential to be useful for this important problem in software engineering by providing cost-effective solutions ranging from SOTA performance on some domains (e.g., semantic merge conflicts in Edge), while providing competitive performance on other domains (e.g., textual merge conflicts in Edge). Our work also illustrates the importance of prompt engineering for these language models as an avenue for research, including automating the most effective prompts given data from a domain.

ACKNOWLEDGMENTS

We thank ISSTA reviewers for their insightful comments. We thank Pallavi Choudhury and Jessica Wolk for their data collection and valuable feedback for this work. We also thank Matt Elacqua for proofreading this work. Jialu Zhang is supported in part by NSF grants, CCF-1715387. Ruzica Piskac is supported in part by NSF grants, CCF-2131476, CCF-1553168 and CNS-1565208.

REFERENCES

[1] Sven Apel, Olaf Leßenich, and Christian Lengauer. 2012. Structured Merge with Auto-Tuning: Balancing Precision and Performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (Essen, Germany) *(ASE 2012)*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/2351676.2351694

[2] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. 2011. Semistructured Merge: Rethinking Merge in Revision Control Systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 190–200. https://doi.org/10.1145/2025113.2025141

[3] Christian Bird and Thomas Zimmermann. 2012. Assessing the Value of Branches with What-If Analysis *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 45, 11 pages. https://doi.org/10.1145/2393596.2393648

[4] Tom B. Brown et al. 2020. Language Models are Few-Shot Learners. In *NeurIPS 2020, December 6-12, 2020, virtual.* https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[5] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Proactive Detection of Collaboration Conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/2025113.2025139

[6] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[7] Cleidson R. B. de Souza, David Redmiles, and Paul Dourish. 2003. "Breaking the Code", Moving between Private and Public Work in Collaborative Software Development *(GROUP '03)*. ACM, New York, NY, USA, 105–114. https://doi.org/10.1145/958160.958177

[8] Elizabeth Dinella, Todd Mytkowicz, Alexey Svyatkovskiy, Christian Bird, Mayur Naik, and Shuvendu K. Lahiri. 2021. DeepMerge: Learning to Merge Programs. arXiv:2105.07569 [cs.SE]

[9] Gleiph Ghiotto, Leonardo Murta, Márcio Barros, and André van der Hoek. 2020. On the Nature of Merge Conflicts: A Study of 2,731 Open Source Java Projects Hosted by GitHub. *IEEE Transactions on Software Engineering* (2020). https://doi.org/10.1109/TSE.2018.2871083

[10] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. 2016. Work Practices and Challenges in Pull-Based Development: The Contributor's Perspective *(ICSE '16)*. 285–296. https://doi.org/10.1145/2884781.2884826

[11] Mário Luís Guimarães and António Rito Silva. 2012. Improving Early Detection of Software Merge Conflicts. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press.

[12] S. Horwitz, J. Prins, and T. Reps. 1988. Integrating Non-Intering Versions of Programs. In *POPL* (San Diego, California, USA). ACM, 133–145.

[13] Naman Jain, Skanda Vaidyanath, Arun Shankar Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram K. Rajamani, and Rahul Sharma. 2022. Jigsaw: Large Language Models meet Program Synthesis. In *ICSE*. IEEE, 1219–1231.

[14] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. 2021. WebGPT: Browser-assisted question-answering with human feedback. arXiv:2112.09332 [cs.CL]

[15] Hung Viet Nguyen, My Huu Nguyen, Son Cuu Dang, Christian Kästner, and T. Nguyen. 2015. Detecting semantic merge conflicts with variability-aware execution. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015). https://doi.org/10.1145/2786805.2803208

[16] Rangeet Pan, Vu Le, Nachiappan Nagappan, Sumit Gulwani, Shuvendu K. Lahiri, and Mike Kaufman. 2021. Can Program Synthesis be Used to Learn Merge Conflict

Resolutions? An Empirical Analysis. In *ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE.

[17] Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2021. Multi-Modal Program Inference: A Marriage of Pre-Trained Language Models and Component-Based Synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 158 (oct 2021), 29 pages. https://doi.org/10.1145/3485535

[18] Victor Sanh et al. 2021. Multitask Prompted Training Enables Zero-Shot Task Generalization. arXiv:2110.08207 [cs.LG]

[19] Danhua Shao, Sarfraz Khurshid, and Dewayne E. Perry. 2009. SCA: A Semantic Conflict Analyzer for Parallel Changes *(ESEC/FSE '09)*. ACM, New York, NY, USA, 291–292. https://doi.org/10.1145/1595696.1595747

[20] Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, Zhi Jin, and Qianxiang Wang. 2019. IntelliMerge: A Refactoring-Aware Software Merging Technique. 3, OOPSLA (2019). https://doi.org/10.1145/3360596

[21] Marcelo Sousa, Isil Dillig, and Shuvendu K. Lahiri. 2018. Verified Three-Way Program Merge. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 165 (Oct. 2018). https://doi.org/10.1145/3276535

[22] Chungha Sung, Shuvendu K. Lahiri, Mike Kaufman, Pallavi Choudhury, and Chao Wang. 2020. Towards understanding and fixing upstream merge induced conflicts in divergent forks: an industrial case study. In *ICSE-SEIP 2020: Seoul, South Korea, 27 June - 19 July, 2020*. ACM. https://doi.org/10.1145/3377813.3381362

[23] Alexey Svyatkovskiy, Todd Mytkowicz, Negar Ghorbani, Sarah Fakhoury, Elizabeth Dinella, Christian Bird, Neel Sundaresan, and Shuvendu Lahiri. 2021. MergeBERT: Program Merge Conflict Resolution via Neural Transformers. arXiv:2109.00084 [cs.SE]

[24] Gust Verbruggen, Vu Le, and Sumit Gulwani. 2021. Semantic programming by example with pre-trained models. In *OOPSLA*. ACM. https://doi.org/10.1145/3485477

[25] Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax.

[26] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2021. Finetuned Language Models Are Zero-Shot Learners. arXiv:2109.01652 [cs.CL]

[27] Jan Wloka, Barbara G. Ryder, Frank Tip, and Xiaoxia Ren. 2009. Safe-commit analysis to facilitate team software development. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 507–517. https://doi.org/10.1109/ICSE.2009.5070549

[28] Wuu Yang, Susan Horwitz, and Thomas Reps. 1990. A Program Integration Algorithm That Accommodates Semantics-Preserving Transformations. *SIGSOFT Softw. Eng. Notes* 15, 6 (Oct. 1990), 11 pages. https://doi.org/10.1145/99278.99290