Merge Conflict Resolution: Classification or Generation?

Jinhao Dong[†], Qihao Zhu[†‖], Zeyu Sun[‡], Yiling Lou[§], Dan Hao[†*]
[†]Key Laboratory of High Confidence Software Technologies (Peking University), MoE
School of Computer Science, Peking University, Beijing, China
[‡]Zhongguancun Laboratory, Beijing, China
[§]School of Computer Science, Fudan University, Shanghai, China
dongjinhao@stu.pku.edu.cn, Zhuqh@pku.edu.cn, szy_@pku.edu.cn, yilinglou@fudan.edu.cn, haodan@pku.edu.cn

***Abstract*—Collaborative development is critical to improve the productivity. Multiple contributors work simultaneously on the same project and might make changes to the same code locations. This can cause conflicts and require manual intervention from developers to resolve them. To alleviate the human efforts of manual conflict resolution, researchers have proposed various automatic techniques. More recently, deep learning models have been adopted to solve this problem and achieved state-of-the-art performance. However, these techniques leverage classification to combine the existing elements of input. The classification-based models cannot generate new tokens or produce flexible combinations, and have a wrong hypothesis that fine-grained conflicts of one single coarse-grained conflict are independent.**

In this work, we propose to generate the resolutions of merge conflicts from a totally new perspective, that is, *generation*, and we present a conflict resolution technique, MergeGen. First, we design a structural and fine-grained conflict-aware representation for the merge conflicts. Then, we propose to leverage an encoder-decoder-based generative model to process the designed conflict representation and generate the resolutions auto-regressively. We further perform a comprehensive study to evaluate the effectiveness of MergeGen. The quantitative results show that MergeGen outperforms the state-of-the-art (SOTA) techniques from both precision and accuracy. Our evaluation on multiple programming languages verifies the good generalization ability of MergeGen. In addition, the ablation study shows that the major component of our technique makes a positive contribution to the performance of MergeGen, and the granularity analysis reveals the high tolerance of MergeGen to coarse-grained conflicts. Moreover, the analysis on generating new tokens further proves the advance of generative models.

***Index Terms*—Merge Conflict Resolution, Generative Models, Conflict Representation**

I. Introduction

Collaborative software development is essential for the development of large-scale projects. To improve the development efficiency, the tasks are often divided among the team for concurrent development and different developers will create their own branches of the main repository [1]. The developers need to submit the pull request (aka merge request) regularly, and code changes will be merged into the main branch after the review of the repository maintainers [2]. However, collaboration increases the occurrence of conflicts when merging code changes in the same locations. Approximately 12% commits are used to merge different versions of code [3], and almost 46% of merge commits lead to conflicts, showing the frequency of merge operations and the accompanying conflicts. It is challenging to resolve merge conflicts, since the developers need to understand the intentions of code changes, which is a time-consuming and tricky task [1], [4]–[7].

In order to reduce the manual efforts of resolving conflicts, many researchers have dedicated into proposing automatic resolution and merge techniques [4], [8]–[14]. Given two conflict versions $\mathcal{A}$ and $\mathcal{B}$, and their common ancestor version $\mathcal{O}$, the aim of the resolution and merge techniques is to produce the accurate resolution $\mathcal{M}$. The traditional approaches can be divided into three categories, i.e., unstructured merge techniques [15], [16], structured merge techniques [17]–[19], and semi-structured merge techniques [20]–[22]. Unstructured merge relies solely on textual similarity, which provides good generalization ability and execution efficiency but can only detect conflicts and cannot resolve them automatically. Structured and semi-structured merge, on the other hand, are based on abstract syntax trees (AST). These approaches are time-consuming and can only be applied to specific programming languages, limiting their applicability. To address these problems, researchers apply deep learning models into merge conflict resolution and propose learning-based techniques, which achieve state-of-the-art performance. Since the learning-based techniques only require the text of conflicts as input, they can also be regarded as unstructured techniques so that they also have a strong ability for generalization and high efficiency. Different from the traditional unstructured techniques, the learning-based techniques can master the common patterns of merging conflicts from the history data, so they have strong conflict-resolving ability.

However, the existing learning-based techniques treat the conflict resolution problem as a classification problem, and dedicate their efforts on combining the existing contents of input, such as selecting which lines from the input or design patterns (e.g., selecting $\mathcal{O}$ or concatenating $\mathcal{A}$ and $\mathcal{B}$) to combine the conflicts at token level. Formulating conflict resolution as a classification task has three limitations. First, classification can only choose the existing tokens from the input, and cannot generate new tokens which do not exist in the input. However, it is a common phenomenon for

* Corresponding author.
‖ Co-first author.

the developers to create new tokens in the resolutions (e.g., invoking a new function to combine the contents of both versions). Second, the patterns to merge code need to be specially designed for different datasets, and the combinations produced by the patterns are limited and not flexible. For example, given the conflict versions $\mathcal{O}$ = XYZ, $\mathcal{A}$ = X′YZ′, $\mathcal{B}$ = XY′Z, and the developer resolution $\mathcal{M}$ = X′Y′Z′, where X′, Y′, Z′ are the modifications of the code snippets X, Y, Z respectively. None of the patterns which simply select one version (i.e., X′YZ′ or XY′Z) or concatenate $\mathcal{A}$ and $\mathcal{B}$ (i.e., X′YZ′XY′Z or X′Z′Y′ if removing the tokens appearing in $\mathcal{O}$) can produce the correct resolution, since this case requires multiple interleaving between $\mathcal{A}$ and $\mathcal{B}$. Third, the state-of-the-art technique MergeBERT computes and resolves the conflicts at token level. One single line-level conflict may consist of several token-level conflicts, and only one token-level conflict can be solved one time if using classification. MergeBERT assumes the different token-level conflicts are independent. However, this assumption is incorrect because they belong to one line-level conflict and rely on the contents of each other.

Therefore, in this paper, we propose to solve the merge conflict resolution problem from a totally new perspective, i.e., producing the resolutions through ***generation***. We put forward a merge conflict resolution technique, **MergeGen**, which leverages an encoder-decoder-based generative model to generate the resolutions auto-regressively. The generative models can produce new tokens that do not exist in the input. Moreover, by producing the resolutions token by token, the models can generate more flexible combinations, which the classification models cannot produce. In addition, the generative models have a strong ability of understanding and generating code. There is no need to design patterns to combine the inputs deliberately and the generative models can directly generate the code which exists in the input. Moreover, through generation, the resolutions for several token-level conflicts in one line-level conflict can be generated at one time and the resolving for them is regarded as an overall problem.

MergeGen uses a new structural and fine-grained conflict-aware representation to represent merge conflicts. The simplest representation of conflicts is simply concatenating the code of $\mathcal{O}$, $\mathcal{A}$ and $\mathcal{B}$ in conflict. It is coarse-grained and requires the model to identify the concrete fine-grained conflicts, and align the fine-grained conflict snippets of three versions on its own. Therefore, we propose to compute the conflicts at token-level and leverage a structural and conflict-aware way to represent the fine-grained token-level conflicts. We enclose and highlight each conflict with the designed structural tokens. Furthermore, we align the corresponding conflict snippets of $\mathcal{O}$, $\mathcal{A}$ and $\mathcal{B}$, and concatenate them with the structural separator. The representation can help the models identify and align the conflict regions, and produce flexible combinations using the tokens from them.

We perform a comprehensive evaluation of MergeGen on multiple programming languages, and MergeGen achieves the state-of-the-art performance from both precision and accuracy for all languages. The further ablation study validates the effectiveness of the proposed fine-grained conflict-aware representation. Moreover, the granularity analysis reveals the high tolerance to coarse-grained conflicts and the performance on generating new tokens prove the advantage of generation.

In summary, this paper makes the following contributions:

- **A new formulation** of conflict resolution as a generation task, which can produce new tokens that do not exist in input, and produce more flexible combinations, without losing the ability to generate existing tokens of input.
- **A structural and fine-grained conflict-aware representation for conflicts** which can help the models identify and align the conflict regions of three versions.
- **An extensive experiment** evaluating our approach against multiple programming languages, which validates the effectiveness of our technique.
- **A replication package** available at https://github.com/DJjjjhao/ase-merge.

II. Motivation

Existing techniques leverage classification to resolve conflicts. They focus on leveraging the existing contents of three input versions $\mathcal{O}$, $\mathcal{A}$, and $\mathcal{B}$, so they put forward different approaches to combine the code of three versions. Some researchers [23] adopt and combine the inputs at line-level, while some researchers [24] design several patterns to combine the inputs at token-level. However, leveraging the existing contents of input has two obvious limitations. In this section, we will illustrate the two limitations and show some cases to help understand. Besides the two limitations, the existing techniques ignore the ability of generative models on generating the code snippet of the input, and there is no need to design patterns to leverage the input deliberately.

A. Limitation 1: Limited Combinations through Classification

```
private static ItemStack tryFillCanister(
<<<<<<< a
       Feature
||||||| base
       final Feature
=======
       final AbstractFeature
>>>>>>> b
       feature,
<<<<<<< a
       IFluidHandler tank, Stack canister, boolean
       isCreativeMode
||||||| base
       final EnumFacing side, final Stack canister
=======
       final TileEntityFluidTank tank, final EnumFacing
       side, final ItemStack canister
>>>>>>> b
```

Resolution:

```
private static ItemStack tryFillCanister(
       AbstractFeature feature, IFluidHandler tank,
       ItemStack canister, boolean isCreativeMode
```

Fig. 1: Motivating example: limited combinations through classification

Although existing techniques propose approaches to combine the input code, the limited combination patterns cannot

cover all the situations. DeepMerge [23] combines the input at line-level, which cannot generate the resolutions that are combinations of parts within one line. MergeBERT [24] combines the tokens through nine defined patterns, which can generate the resolutions requiring the interleaving at token-level. However, the proposed patterns are simple and limited, i.e., choosing one from three versions, concatenating $\mathcal{A}$ and $\mathcal{B}$, and the previous combinations removing the tokens that occur in $\mathcal{O}$. The combinations generated by the patterns are limited and not flexible.

We present a case in Fig. 1. The SOTA approach MergeBERT relies on the conflicts at token-level, and leverages the pattern to combine the token-level conflicts. In Fig. 1, we show the results of `git merge` at token-level of a function declaration in Java. `Git merge` is the command provided by `git`, which leverages the three-way merge algorithm `diff3` to merge the two branches (i.e., $\mathcal{A}$ and $\mathcal{B}$) to their common ancestor (i.e., $\mathcal{O}$). The regions between `<<<<<<<` and `>>>>>>>` are in conflict, which we color in red; and the remaining regions are not in conflict, which we color in green. The code between `<<<<<<< a` and `||||||| base` is from $\mathcal{A}$, the code between `||||||| base` and `=======` is from $\mathcal{O}$, and the code between `=======` and `>>>>>>> b` is from $\mathcal{B}$. In addition, we show the developer resolution of the conflict, and we highlight the tokens of conflicts that are selected into the resolution. We can notice that there exist two conflicts in the code snippet. The resolution of the first conflict is `AbstractFeature`, which can be obtained by selecting the code of $\mathcal{B}$ and removing the tokens that appear in $\mathcal{O}$, which is one pattern proposed by MergeBERT.

However, for the second conflict, none of the simple patterns that combine the three versions can generate the correct resolution. The resolution consists of multiple interleaving of the tokens between $\mathcal{A}$ and $\mathcal{B}$, i.e., the three code snippets `IFluidHandler tank`, `ItemStack canister`, and `boolean isCreativeMode` in the resolution come from $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{A}$ respectively. This resolution cannot be produced by simply selecting one from $\mathcal{A}$ and $\mathcal{B}$ or concatenating $\mathcal{A}$ and $\mathcal{B}$. This case can reflect the limitation of classification on generating flexible combinations. By contrast, since the generative model generate the resolutions token by token, it can produce any format of combinations and cover all the solutions that the developers will put forward.

B. Limitation 2: Classification Cannot Generate New Tokens

The second limitation of classification-based techniques is the ability to generate new tokens which do not exist in the input. The current techniques all pay attention to how to combine the tokens of the input, so they cannot generate any new tokens. However, it is a common phenomenon for the developers to write new tokens in the resolution, such as invoking a new function to combine the contents of both versions. Here is a real example in our dataset shown in Fig. 2. The developer invokes a new function `getGraceOnArrearsAgeing` which doesn't exist in the input. This resolution cannot be generated by existing techniques definitely. Generative models

```
<<<<<<< a
product.getOutstandingLoanBalance(),
acc.emiAmountVariations,
acc.memberVariations, product,
acc.inArrears);
||||||| base
product.getOutstandingLoanBalance(),
acc.emiAmountVariations,
acc.memberVariations, product);
=======
product.getOutstandingLoanBalance(),
acc.emiAmountVariations,
acc.memberVariations, product,
product.getGraceOnDueDate());
>>>>>>> b
```

Resolution:

```
product.getOutstandingLoanBalance(),
acc.emiAmountVariations,
acc.memberVariations, product,
acc.inArrears,
product.getGraceOnArrearsAgeing());
```

Fig. 2: Motivating example: generating new tokens

can address this limitation, which can produce any tokens in the vocabulary. In addition, we use Byte-Pair Encoding (BPE) [25] tokenization to split the tokens according to the frequency, which can generate any tokens of the programming language and combine the sub-words of the input.

C. Neglecting Generative Models' ability to Generate Existing Tokens

The motivation of the existing techniques to combine the contents of the input is that many tokens of the resolutions come from the conflicts. However, they ignore the ability of the generative models to generate the tokens of the input. The pre-trained code-related models are trained on a large amount of code corpus, and have strong ability in code comprehension and generation [26]–[28]. After fine-tuning on the merge resolution task, for the tokens of input which can be adopted to the output directly, the models can learn the capacity to generate them easily.

Therefore, generation is a better direction for the merge conflict resolution task, which can not only generate the tokens of input, but also produce new tokens and more flexible combinations.

III. Approach

In this section, we will give a detailed introduction of MergeGen, and the overview of our approach is in Fig. 3. MergeGen consists of two major components, i.e., the structural and fine-grained conflict-aware representation for merge conflicts and the encoder-decoder generative model which resolves the conflicts through generation. Given the $\mathcal{O}$, $\mathcal{A}$, and $\mathcal{B}$ versions of conflicts, we first represent the conflicts with the proposed structural and fine-grained conflict-aware representation. After that, we feed the conflict representation to the encoder-decoder generative model to produce the resolutions

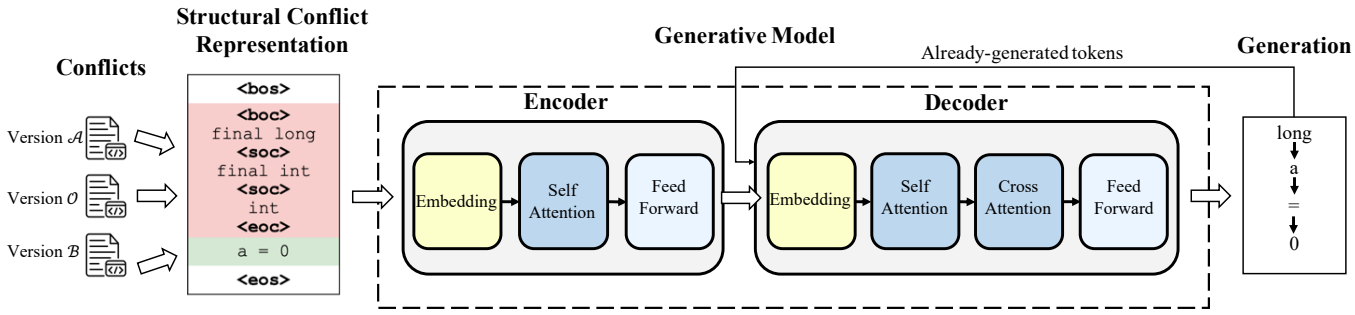

Fig. 3: The overview of MergeGen: the conflicts are first represented by our proposed structural and fine-grained representation and then put to the generative model to generate the resolutions token by token.

token by token through generation. We will first introduce the proposed representation in Section III-A and then introduce the proposed model in Section III-B

A. Structural and Fine-grained Conflict-aware Representation

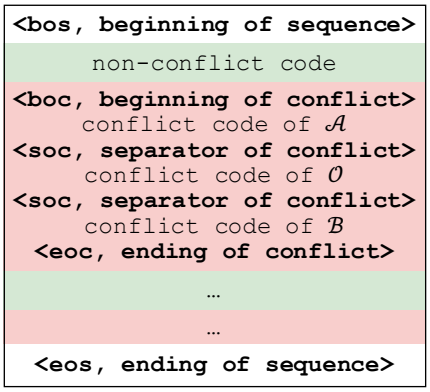

Fig. 4: The structural and fine-grained conflict-aware representation of merge conflicts

The first component of MergeGen is the specially-designed representation for the merge conflicts, defined as Repre_{merge}. The simplest way to represent the conflicts is to directly concatenate the three versions in conflict, i.e., $\mathcal{O}$, $\mathcal{A}$, and $\mathcal{B}$. However, this trivial representation requires the model to identify the concrete fine-grained conflicts, and align the fine-grained conflict snippets of three versions on its own. This is challenging for models especially when there are multiple finer-grained conflicts in one coarse-grained conflict. To solve this problem, we propose a structural and fine-grained conflict-aware representation for the conflicts, shown in Figure 4.

First, we compute the conflicts at token-level, which are finer-grained and more accurate than the line-level conflicts. Within one line-level conflict, there exist common code snippets which are actually not conflicts, and exist several smaller and more accurate conflicts. For example, given a conflict where $\mathcal{O}$ is `final int a = 0`, $\mathcal{A}$ is `final long a = 0`, and $\mathcal{B}$ is `int a = 0`. The developer resolution $\mathcal{M}$ is `long a = 0` which consists of both two smaller changes, i.e., changing the type from `int` to `long` and removing the modifier `final`. The line-level merge will identify the whole line of each version as only one conflict and selecting one version or combining several versions at line-level as the resolution is incorrect, while the token-level merge can identify two smaller conflicts and the combination of respective resolutions can obtain the correct answer. The proposed representation and the generation process of this example are shown in Fig. 3.

Then, we leverage a structural and conflict-aware way to represent the fine-grained token-level conflicts. According to Fig. 4, the proposed representation consists of two types of regions, i.e., non-conflict regions and conflict regions. The non-conflict regions are directly put into the final representation. The conflict regions are enclosed by two proposed structural tokens, i.e., ⟨boc⟩ and ⟨eoc⟩, which represent the beginning and the end of conflicts respectively. In addition, we align the corresponding conflict snippets of the $\mathcal{O}$, $\mathcal{A}$ and $\mathcal{B}$, and concatenate them with the structural separator ⟨soc⟩. This can help the models align the areas of three versions and combine the tokens of them.

Finally, we enclose the whole conflict with ⟨bos⟩ and ⟨eos⟩, which represents the beginning and end of the whole sentence, and we obtain the final representation of the merge conflicts Repre_{merge}. To sum up, the proposed conflict representation is structural and fine-grained conflict-aware, where the conflict regions are enclosed with the structure tokens and are distinguished from the non-conflict regions. The models can easily identify the conflict and non-conflict regions, and align the conflict snippets from different versions. The representation can help the models focus on the conflict regions, and produce flexible combinations using the tokens from the input.

B. Generative Model

The biggest difference of MergeGen from the previous techniques is leveraging ***generation*** to produce the resolutions. We adopt an encoder-decoder-based generative model to process our proposed representation and generate the resolutions auto-regressively. Since we require both the encoder and the decoder, we utilize the state-of-the-art encoder-decoder-based pre-trained model T5 (Text-To-Text Transfer Transformer) [29], which specializes in the text-to-text tasks. Specially, we use CodeT5, which is based on T5 architecture and

pre-trained on code-related tasks. The encoder and the decoder of CodeT5 are all based on Transformer [30] architecture.

1) Encoder: After we obtain the final representation of the merge conflicts Repre$_{merge}$, we first feed it to the encoder to obtain the high dimensional representation, which is then fed to the decoder to guide generation.

As we have introduced in Section III-A, to represent the conflicts in a structural way, we add some structural tokens. We insert the structural tokens into the vocabulary of CodeT5 and assign the randomly-initialized and trainable embedding vectors to them. The final conflict representation Repre$_{merge}$ can be tokenized with the tokenizer of CodeT5 into several tokens, which is defined as $[t_1, t_2, ..., t_{N_c}]$, where N_c is the length of the tokens of Repre$_{merge}$. Then we pass the sequence of tokens to the vocabulary and obtain the embedding vectors E_c, which can be represented by $[\boldsymbol{e}_1, \boldsymbol{e}_2, ..., \boldsymbol{e}_{N_c}]$, where $E_c \in \mathbb{R}^{d_x \times N_c}$ and d_x is the dimension of the embedding.

Then we feed the embedding vectors E_c to the transformer layer. The transformer layer of the encoder consists of one multi-head self-attention layer and one fully connected feed-forward network. The encoder consists of L transformer layers, and the computation process of each layer is as follows. Given the hidden states of the last layer X_e^{l-1} ($X_e^0 = E_c$), the transformer layer computes the multi-head self-attention, i.e., A_e. The multi-head attention is the concatenation of several single-head attention $A_e(i)$, which is the weighted sum of X_e^{l-1}. The computation formula is shown in Equation 1 and Equation 2, and to make the formula concise, we simplify X_e^{l-1} to X_e.

$$A_e(i) = W_v X_e \cdot \text{softmax}\left(\frac{{X_e}^T {W_k}^T \cdot W_q X_e}{\sqrt{d_x}}\right) \quad (1)$$

$$A_e = W_o[A_e(1); A_e(2); ...; A_e(h)] \quad (2)$$

In the equations, $W_q \in \mathbb{R}^{d_x \times d_x}$, $W_k \in \mathbb{R}^{d_x \times d_x}$, $W_v \in \mathbb{R}^{d_x \times d_x}$, $W_o \in \mathbb{R}^{d_x \times h d_x}$ denote the projection parameters, and h is the number of heads. After that, the attention A_e is passed to the fully connected feed-forward network to obtain the output of the encoder at the l-th layer, i.e., X_e^l, as shown in Equation 3.

$$X_e^l = W_2 \cdot \max(0, W_1 A_e + b_1) + b_2 \quad (3)$$

In the equation, $W_1 \in \mathbb{R}^{d_x \times d_x}$, $W_2 \in \mathbb{R}^{d_x \times d_x}$, $b_1 \in \mathbb{R}^{d_x}$, $b_2 \in \mathbb{R}^{d_x}$ are learnable parameters. After passing the L layers of the encoder, we can obtain the final output of the encoder, that is, X_e^L, which then serves as the input of the decoder. To make the illustration convenient, we simplify X_e^L to X_e in the following description.

2) Decoder: After the encoder processes and encodes the conflicts, we can obtain the final high dimensional representation of the conflicts X_e. Then, the decoder feeds on X_e and generates the tokens of resolutions one by one.

The decoder is also based on the transformer structure. Besides the self-attention and fully connected feed-forward network of the encoder, the decoder has another extra layer, the cross-attention layer over the output of the encoder. Given the hidden states of the already generated $k-1$ tokens X_d^{k-1} (i.e., $[\boldsymbol{x}_d^1, \boldsymbol{x}_d^2, ..., \boldsymbol{x}_d^{k-1}]$), the process to generate the k-th token is as follows.

Similar to Equation 1 and Equation 2, the first step is self-attention over the already generated tokens which can be computed in Equation 4 and Equation 5.

$$\boldsymbol{a}_d^k(i) = W_v X_d^{k-1} \cdot \text{softmax}\left(\frac{\left(X_d^{k-1}\right)^T W_k^T \cdot W_q \boldsymbol{x}_d^{k-1}}{\sqrt{d_x}}\right) \quad (4)$$

$$\boldsymbol{a}_d^k = W_o[\boldsymbol{a}_d^k(1); \boldsymbol{a}_d^k(2); ...; \boldsymbol{a}_d^k(h)] \quad (5)$$

After that, the decoder has a unique layer (i.e., cross-attention layer) that does not exist in the encoder. The cross-attention layer is used to integrate the input information and guide the generation process. The cross-attention is also multi-head attention and the computation process is shown in Equation 6 and Equation 7.

$$\boldsymbol{a}_c^k(i) = W_v X_e \cdot \text{softmax}\left(\frac{X_e^T W_k^T \cdot W_q \boldsymbol{a}_d^k}{\sqrt{d_x}}\right) \quad (6)$$

$$\boldsymbol{a}_c^k = W_o[\boldsymbol{a}_c^k(1); \boldsymbol{a}_c^k(2); ...; \boldsymbol{a}_c^k(h)] \quad (7)$$

Similar to Equation 3 of the encoder, the cross attention $\boldsymbol{a}_c^k$ is passed to the fully connected feed-forward network to obtain the output of the decoder. Since the decoder needs to generate the token from the vocabulary, the output needs to be resized to the size of the vocabulary, as shown in Equation 8.

$$\boldsymbol{x}_d^k = W \cdot \left(W_2 \cdot \max(0, W_1 \boldsymbol{a}_c^k + b_1) + b_2\right) \quad (8)$$

$W \in \mathbb{R}^{|V| \times d_x}$ is a trainable parameter, $|V|$ is the size of the vocabulary, and the output $\boldsymbol{x}_d^k \in \mathbb{R}^{|V|}$.

Finally, the likelihood of each token in the vocabulary being chosen as the subsequent token will be computed, which can be obtained by a *softmax* layer ensuring the sum of probabilities is 1. The probability distribution across the vocabulary is denoted by $\boldsymbol{p}^k$, where $\boldsymbol{p}^k(i)$ represents the probability of the ith token to be generated. The computation of $\boldsymbol{p}^k(i)$ is performed according to Equation 9.

$$\boldsymbol{p}^k(i) = \frac{exp\{\boldsymbol{x}_d^k(i)\}}{\sum_{j=1}^{|V|} exp\{\boldsymbol{x}_d^k(j)\}} \quad (9)$$

Therefore, according to $\boldsymbol{p}^k$, the token with the highest probability in the vocabulary will be selected as the kth token.

IV. Experimental Setup

- **RQ1: Overall effectiveness.** How does MergeGen perform on generating merge conflict resolutions? We compare the effectiveness of MergeGen on generating the resolutions with the state-of-the-art techniques.
- **RQ2: Multi-language effectiveness.** How does MergeGen perform for different languages? One advantage of

MergeGen is that it belongs to unstructured techniques and can be applied to any programming languages. Therefore, we evaluate the performance of MergeGen for multiple languages to verify its generalization ability.

- **RQ3: Ablation analysis and granularity analysis.** *(1) RQ3.a: Ablation analysis.* How does the proposed merge conflict representation perform? We evaluate the performance of the variant of MergeGen without the representation to validate the benefits brought by it. *(2) RQ3.b: Granularity analysis.* How does MergeGen perform for coarse-grained conflicts? The existing state-of-the-art technique MergeBERT can only be used at token-level conflict, which is limited when the code has large scale and the computation of token-level conflicts are time-consuming.
- **RQ4: Performance on new tokens.** How effectively does MergeGen perform on generating new tokens? The ability to generate new tokens is one key advantage of MergeGen against others, so we validate it in this RQ.

A. Dataset

We evaluate our proposed model on the well-constructed multilingual dataset, which is established by the state-of-the-art technique MergeBERT and has been used to evaluate all the advanced techniques [24]. This dataset is collected from more than 100,000 open source software repositories and reserves the commits that have merge conflicts. For each commit, it includes two conflict versions $\mathcal{A}$ and $\mathcal{B}$, the common base $\mathcal{O}$, and the resolution of the developer $\mathcal{M}$. Following the common practice in deep learning [31], [32], we randomly split the dataset of each programming language into training/validation/testing set by an approximate ratio of 8:1:1.

B. Compared Techniques

We compare our technique with the SOTA merge conflict resolution techniques, including the unstructured techniques, the structured techniques, and the learning-based techniques.

Diff3 [15] algorithm is the default merge algorithm of the version control system. As the unstructured techniques, `diff3` operates only according to the code text at line-level. It identifies the conflicts where version $\mathcal{A}$ and version $\mathcal{B}$ modifies the same places of their common ancestor $\mathcal{O}$. `Diff3` cannot resolve the conflicts and require the intervention of developers.

JDIME [19] is a semi-structured technique which can only be applied into Java. To reduce the time cost of structured merge techniques, JDIME switches between structured and unstructured techniques with auto-tuning. Structured techniques are adopted when the conflicts are detected and unstructured techniques are adopted when the conflicts are not detected.

jsFSTMerge [18] adds the grammar of JavaScript to the semi-structured tool FSTMerge [17] to target at JavaScript. It adds the syntax information to the model, including which nodes can permute safely (e.g., method declarations) and the order of which nodes cannot be changed (e.g., statements).

MergeBERT [24] is a learning-based technique and achieves the state-of-the-art performance. This approach formulates merge conflict generation as a classification task, which defines nine patterns to interleave the inputs at token-level (e.g., choosing $\mathcal{O}$ or concatenating $\mathcal{B}$ and $\mathcal{A}$). Classification has huge limitations in merge conflict generation, i.e., it cannot generate flexible combinations of the input and cannot generate new tokens.

C. Evaluation Metrics

Since the resolution is code, following the previous works [23], [24], [33], [34], we regard the generated resolution as correct only when it exactly matches the ground truth resolution. For each conflict, we will choose the resolution with the highest probability among those generated by the models. We compute both precision and accuracy of the string match. Accuracy is the percentage of the conflicts for which the model can generate the exactly-matched resolutions among all the conflicts. Since the generated resolutions might not meet the grammar constraints of the languages, we also compute precision, which is the percentage of the correctly-resolved conflicts among the conflicts for which the model can generate the syntactically-valid resolutions. Accuracy can reflect the real performance, since it concern the performance on the whole dataset.

D. Implementation

Representation. Following the previous works [23], [24], MergeGen leverages the command `git merge` provided by `git` to obtain the conflicts at token-level of the three versions $\mathcal{O}$, $\mathcal{A}$, and $\mathcal{B}$. Since `git merge` can only operate at line-level, we put each token at one single line, and execute the command to obtain the conflicts at token-level.

Model. MergeGen is established on the base of the pre-trained model CodeT5. Since the SOTA technique MergeBERT adopts the small version of the pre-trained model CodeBERT [1] with 84M parameters, to make the comparison fair, we also leverage CodeT5-small [2] with 60M parameters [27], which are even smaller than MergeBERT-small. CodeT5 consists of 6 encoder layers and 6 decoder layers, and the hidden state size of 512. The vocabulary size is 32,102, consisting of two newly-added structure-related words (i.e., $\langle$`boc`$\rangle$ and $\langle$`eoc`$\rangle$, and we use the separator of CodeT5 as $\langle$`soc`$\rangle$) and 32,100 words from the original vocabulary. The length of the input sequence (including the conflict regions of the three versions and the surrounding context) is limited to 500 BPE tokens and the output resolution is limited to 200 BPE tokens. In the fine-tuning phase, we leverage the AdamW optimizer [35] with the learning rate of 0.0001 and the batch size of 210. We tune the hyper-parameters and select the best performing model in the validation set. The training time ranges from 0.5 to 4.5 hours for different languages. In the inference phase, we set the beam size of the generation as 3 and choose the top-1 output with the highest probability. After

[1] https://huggingface.co/huggingface/CodeBERTa-small-v1

[2] https://huggingface.co/Salesforce/codet5-small

that, following the existing work [24], we leverage tree-sitter [3] parser to check whether the output satisfies the syntax of that language. Only when it meets the syntax, we will regard it as the resolution candidate, otherwise, we will think MergeGen cannot generate a valid resolution for the conflict.

Environment. The experiments are carried out on a workstation that is equipped with a 56-core CPU and 190G memory. When fine-tuning the models, we leverage six 24G GPUs of GeForce RTX 3090.

V. Results and Analysis

In this section, we first present the overall performance of MergeGen from precision and accuracy (RQ1) in Section V-A, the results for different languages (RQ2) in Section V-B, ablation study and granularity analysis (RQ3) in Section V-C, and the performance on new tokens in Section V-D.

A. RQ1: Overall Effectiveness

TABLE I: Overall performance of merge conflict generation of MergeGen and the compared techniques on Java

Model	Precision	Accuracy
diff3	**85.5**	2.2
JDIME [19]	26.3	21.6
MergeBERT [24]	63.9	63.2
MergeGen (our tool)	69.2	**67.7**

In this research question, we compare the performance of MergeGen and other techniques. In Table I, we present the precision and accuracy of the resolutions generated by different techniques. The compared techniques include `diff3`, the state-of-the-art technique MergeBERT [24], and the state-of-the-art semi-structured technique on Java, i.e., JDIME [19].

As we can see in Table I, MergeGen achieves the best performance among the compared techniques, and the improvements of precision and accuracy towards the SOTA technique MergeBERT are 8.3% and 7.1%, proving the effectiveness of MergeGen. The accuracy of MergeGen is close to the precision, which indicates that MergeGen can produce syntactically-correct and valid resolutions for most cases.

Our results contradict the statements in MergeBERT, which claims that the generative models are not applicable to the conflict resolution tasks and the naturalness hypothesis of code [36] is not enough to understand the intentions of developers when merging programs. In fact, the code-related language models are pre-trained through auto-regressive tasks to generate tokens of code one by one (e.g., CodeGPT [28] and CodeT5 [27]). Therefore, the pre-trained models have the strong ability to understand and generate the programs [26]–[28]. In addition, in the merge conflict resolution task, many code snippets in the resolution are from the input, and it is simple for pre-trained models to directly generate the code which exists in the input. The better performance of MergeGen shows that there is no need to design patterns to combine the inputs and the generative models can learn how to generate the tokens of input which needs to be put in the resolutions.

MergeBERT formulates the conflict resolution task to a classification task, and for a line-level conflict consisting of several token-level conflicts, MergeBERT resolves each small conflict independently and cannot refer to other small conflicts. It assumes that the resolving processes for the token-level conflicts are independent, which is incorrect because they belong to one line-level conflict. By contrast, MergeGen puts all small conflicts together and generate the resolutions for them at a time, which regards resolving several relevant conflicts as an overall problem.

JDIME is one semi-structured conflict resolution tool. The semi-structured approaches leverage the language-specific features, so they can be only applied to the targeted language. From Table I, we can notice that JDIME performs much worse than the learning-based techniques MergeBERT and MergeGen. The precision and accuracy of JDIME is only 26.3% and 21.6%. This is because the structured techniques can only deal with some simple cases, which are conflicts from text, but not conflicts from syntax, such as the order of functions. They cannot resolve the real conflicts and combine the input to obtain a resolution.

As for `diff3`, the precision is really high (i.e., 85.5%), but the accuracy is too low (i.e., 2.2%). This is natural because `diff3` cannot resolve conflicts, and it can only merge the changes which don't modify the same location of the base version. Therefore, it cannot produce valid resolutions for most conflicts, so the accuracy is low. For thousands of conflicts, it can only generate valid resolutions for 140 of them.

B. RQ2: Multi-language effectiveness

TABLE II: Performance of MergeGen and the state-of-the-art techniques on different languages

Language	Model	Precision	Accuracy
Java	MergeBERT	63.9	63.2
	JDIME	26.3	21.6
	MergeGen	**69.2**	**67.7**
C#	MergeBERT	68.7	67.3
	MergeGen	**75.4**	**73.8**
JavaScript	MergeBERT	66.9	65.6
	jsFSTMerge	15.8	3.6
	MergeGen	**70.9**	**68.3**

MergeGen is not targeted at certain specific programming languages, and it does not require the language information like the structured tools. Therefore, MergeGen is applicable to any language and has a good generalization ability. In this research question, besides Java, we evaluate MergeGen on another two languages, that is, JavaScript and C#. The two languages are dynamically-typed and statically-typed languages respectively. In Table II, we present the precision and accuracy of MergeGen and the existing state-of-the-art technique MergeBERT. Besides, we present the state-of-the-art semi-structured techniques JDIME and jsFSTMerge, which are designed to target at Java and JavaScript respectively.

[3]https://tree-sitter.github.io/tree-sitter

According to the table, MergeGen achieves better performance than MergeBERT for each language. The improvements of precision for Java, C#, and JavaScript are 8.3%, 9.8%, and 6.0% respectively; and the improvements of accuracy for the three languages are 7.1%, 9.7%, and 4.1% respectively. Among these languages, Java and C# are statically-typed languages, which require the variables to be declared explicitly, while JavaScript is one dynamically-typed language, whose variables are inferred according to the value assigned to them. Therefore, the dynamically-typed languages have more flexible syntax, and are more error-prone at the same time. The structured and semi-structured merge techniques depend on the syntax so it is more challenging for them to match and merge at the abstract syntax tree (AST)-level on dynamically-typed languages. We can notice that the semi-structured technique jsFSTMerge performs poorly on JavaScript, which is consistent with the previous work [18], [24].

By contrast, MergeGen performs well on both dynamically-typed languages and statically-typed languages uniformly, reflecting the applicability the MergeGen on both types of languages. Since MergeGen treats the input as a sequence of tokens, it is not influenced by the type of the languages.

C. RQ3: Ablation Study and Granularity Analysis

TABLE III: Performance of MergeGen and the variant removing the merge representation

Model	Precision	Accuracy
MergeBERT (token-level)	63.9	63.2
MergeGen$_{repr-}$ (line-level)	68.9	66.8
MergeGen (token-level)	**69.2**	**67.7**

1) RQ3.a-Ablation Study: In this section, we conduct an ablation study to investigate the effectiveness of the component of MergeGen. The major component of MergeGen is the structural and fine-grained conflict-aware representation for the merge conflict. To verify the benefits brought by it, we propose a variant of MergeGen, i.e., MergeGen$_{repr-}$, by removing the merge representation. We remove the structural tokens enclosing the conflict region, and remove the separators that separate the conflict regions of three versions. Instead, we directly concatenate the code of base version $\mathcal{O}$, and two conflict versions $\mathcal{A}$ and $\mathcal{B}$. This representation can also be seen as the line-level conflicts, which simply put the coarse-grained conflict regions returned by `git merge` at line-level together. Then, we feed the line-level conflicts to MergeGen and the results are shown in Table III.

According to the table, both the precision and the accuracy decline without the merge representation, which is 68.9% and 66.8%, respectively. The performance on the more important metric, i.e., accuracy, declines more. This can indicate that the proposed merge representation is effective for producing resolutions for conflicts. Without the merge representation, the model has to identify the fine-grained conflict regions by itself and align the corresponding conflict regions of three versions, which is challenging especially when there are multiple finer-grained conflicts in one coarse-grained conflict and there are few common tokens as anchors to align the three versions. By contrast, with the merge representation, the conflicts are enclosed and marked explicitly with specially-defined structural tokens. In addition, the tokens of three versions are aligned and put together, using the separators to separate them. Through this representation, the models can directly generate the code which is not in conflict, and pay more attention to the conflict regions.

```
------version A------
IndexShard ref = new IndexShard(indexShard, true);
assert addShardReference(ref, ref.routingEntry());
return ref;

------version O------
return new IndexShard(indexShard, true);

------version B------
return IndexShard.createPrimary(indexShard);
```

Resolution:

```
IndexShard ref = IndexShard.createPrimary(indexShard);
assert addShardReference(ref, ref.routingEntry());
return ref;
```

Resolution of MergeGen$_{repr-}$:

```
IndexShard ref = new IndexShard(indexShard);
assert addShardReference(ref, ref.routingEntry());
return ref;
```

Fig. 5: The real case where MergeGen$_{repr-}$ produces resolution with poorer quality than MergeGen

We further look into one real case in our dataset where MergeGen$_{repr-}$ generates a wrong resolution, but MergeGen can generate the correct one, shown in Fig. 5. In the figure, we present the code from $\mathcal{O}$, $\mathcal{A}$ and $\mathcal{B}$, the ground truth resolution written by the developer and the resolution generated by MergeGen$_{repr-}$. We highlight the wrongly-generated tokens in italics, and MergeGen$_{repr-}$ wrongly chooses the version of $\mathcal{A}$, i.e., retaining `new` and omitting `createOnPrimary`. The code of three versions is concatenated simply, so the model has to identify the conflicts and align the corresponding conflicts of three versions by itself. In this case, MergeGen$_{repr-}$ wrongly aligns the keyword `return` of the three versions and thinks that the first sentence of $\mathcal{A}$ is not in conflict so that wrongly generates the contents of this sentence. We show our proposed structural representation of merge conflict of this case in Fig. 6. In this representation, the conflicts are aligned correctly, which can help the model focus on the conflicts and produce flexible combinations of the tokens. Since $\mathcal{A}$ does not modify the keyword `new` and $\mathcal{B}$ removes `new`, the developer tends to remove `new`. MergeGen can generate the resolution that exactly matches the ground truth.

2) RQ3.b-Granularity Analysis: MergeGen$_{repr-}$ can not only be regarded as the ablation model of MergeGen, but also be regarded as the line-level implementation of MergeGen, since the concatenation of $\mathcal{O}$, $\mathcal{A}$, and $\mathcal{B}$ is just the combination of the conflicts of three versions at line-level returned by `git merge`. From Table III, MergeGen$_{repr-}$ still performs better than MergeBERT.

```
<bos, beginning of sequence>
<boc, beginning of conflict>
    IndexShard ref = new
<soc, separator of conflict>
        return new
<soc, separator of conflict>
          return
  <eoc, ending of conflict>
IndexShard.createPrimary(
indexShard);
assert addShard(ref, ref.
routingEntry());
return ref;
  <eos, ending of sequence>
```

Fig. 6: The proposed merge conflict representation of the case in Fig. 5

The line-level conflicts are coarse-grained, and one line-level conflict might consist of several finer-grained token-level conflicts. Although fine-grained conflicts are more concrete and accurate, and can generally produce more accurate resolutions, it suffers from limitations in some cases. First, when the conflict size is huge and the `diff3` algorithm at token-level is time-consuming, the token-level approach is not applicable. Secondly, the resolution at token-level might be incorrect, for example, given $\mathcal{O}$ = `core.workflow.TlsContext`, $\mathcal{A}$ = `core.workflow.chooser.Chooser`, $\mathcal{B}$ = `core.state.TlsContext`. The resolution at token-level is `core.state.chooser.Chooser` and incorrect, while the correct one should be the concatenation of $\mathcal{A}$ and $\mathcal{B}$.

The existing SOTA technique MergeBERT leverages classification to select a combination pattern to combine the resolutions of conflicts. Therefore, the resolutions they can generate rely on the granularity of the conflicts. If MergeBERT is applied at line-level, it can only generate the combinations of the lines. By contrast, since MergeGen produces resolutions through generation, the flexibility of the resolutions it can generate will not be influenced by the granularity of the input. The effect of finer-grained representation is to make the models know which parts are the real conflicts and important, and focus on producing resolutions for them.

MergeGen$_{repr-}$ is the line-level version of MergeGen, i.e., leveraging the conflicts of $\mathcal{O}$, $\mathcal{A}$, $\mathcal{B}$ at line-level. It performs still better than MergeBERT, reflecting that MergeGen can achieve good performance at both line-level and token-level, and are not limited by the granularity of the input. In other words, MergeGen has high tolerance to coarse-grained conflicts. However, MergeBERT can only operate on token-level.

D. RQ4: Effectiveness on Generating New Tokens

When resolving the conflicts, the developers will not only adopt the existing tokens of the conflicts, but also create new tokens, such as invoking a new function to combine the contents of both versions. None of the previous approaches can produce the resolutions containing new tokens, but since MergeGen is a generative model, it can definitely generate new tokens. In this research question, we evaluate the effectiveness of MergeGen in generating new tokens. We choose data from the test set in which the ground truth resolutions include new tokens that are not present in conflicts. This subset of data constitutes an average of 6.5% of the total dataset for each language. Subsequently, we compute the precision and accuracy of the resolutions produced by MergeGen for this subset. For comparison, we also assess the precision and accuracy of data with resolutions that do not incorporate any new tokens.

The average precision and accuracy achieved by MergeGen in each language are 34.5% and 33.7%, respectively, for the subset containing new tokens. For comparison, the precision and accuracy for the subset containing no new tokens are 74.5% and 72.4%, respectively. The performance of generating resolutions containing new tokens is not as good as generating those containing only existing tokens. This is reasonable because the vocabulary of new tokens is really large and generating the correct new tokens is more difficult than generating the tokens from the little vocabulary of existing tokens in conflicts. Even so, MergeGen outperforms the existing classification-based techniques, which cannot generate any new tokens.

VI. Discussion

A. Threats to Validity

Threats to internal validity mainly lie in the implementation of MergeGen. To reduce the threat, we build our model based on existing mature and widely-used tools/libraries. Our model is established on the basis of the CodeT5 API provided by HuggingFace [4]. The APIs of pre-trained models provided by HuggingFace are the most popular in NLP community and are widely used by researchers and developers [24], [37]. In addition, we use the most widely-used tool `git merge` to obtain the conflicts. We also use the popular code parser tool tree-sitter to judge whether the generated resolution satisfies the syntax. Moreover, the first two authors have checked the implementation carefully. ***Threats to external validity*** mainly lie in the dataset we use to evaluate MergeGen and the compared techniques. To migrate the threat, we utilize the well-established multilingual benchmark, which has been constructed on the open source projects [24]. Furthermore, we follow the common practice in deep learning [31], [32] to split the dataset and conduct a comprehensive evaluation including multi-language evaluation, ablation study, granularity analysis, and analysis on new tokens. Nonetheless, threats in terms of practicality and realism persist in the evaluation process, i.e., we do not split the dataset following the development timeline. ***Threats to construct validity*** mainly lie in the evaluation metrics. To reduce the threat, we follow the previous works [23], [24] to use precision and accuracy to evaluate the effectiveness of MergeGen. In addition, following them, we regard the output as a candidate only when it passes the

[4]https://github.com/huggingface/transformers

syntactic check. Furthermore, following the previous works about code generation [33], [34], we regard the generated program correct only when it exactly matches the ground truth.

B. Limitations

This section introduces the limitations of MergeGen. First, because of the limitation of computing resources, we set a maximal length to the conflicts and resolutions, and the parts exceeding the limit will be discarded. Therefore, the effectiveness of MergeGen will be influenced when dealing with too long code, which is a common limitation of learning-based techniques. The length limit we set can occupy 95% of the data in our dataset, and with enough resources MergeGen can process any length of the code in theory. Although the fine-tuning process requires many computing resources, a common issue among deep learning models including MergeBERT, MergeGen can operate efficiently after being fine-tuned once. Second, although MergeGen has the ability to generate new tokens which do not exist in the input, it is challenging to deal with the case where the developers write the new words with no connection to the context. Third, MergeGen will have limited performance when there is no sufficient information to judge which version should be selected. For example, $\mathcal{O}$ is `int a = 0`, $\mathcal{A}$ is `int a = 1`, and $\mathcal{B}$ is `int a = -1`. No evidence can be leveraged to make the judgement, and more context is required.

VII. Related Work

Recently, a growing number of people of the software development community become interested in merge conflict resolution. Researchers and developers realize the importance of effective conflict resolution techniques in promoting collaborative software development and improving efficiency. To reduce the efforts of developers to resolve the conflicts, researchers have proposed various techniques to automatically generate resolutions for the merge conflicts. The conflict resolution and merge techniques can be mainly divided into four categories, unstructured merge techniques, structured merge techniques, semi-structured techniques, and the state-of-the-art learning-based techniques.

Unstructured Techniques. Unstructured merge techniques operate only according to code text, and do not consider the syntax and the structure information of the programming languages. The merge algorithm adopted in the version control system such as `git` is just unstructured merge technique, i.e., `diff3` algorithm. `diff3` identifies the conflicts where version $\mathcal{A}$ and version $\mathcal{B}$ modifies the same places of their common ancestor $\mathcal{O}$. Since `diff3` has no priori knowledge, they cannot resolve the conflicts and the developers have to intervene the resolution process. Despite the poor conflict resolution ability, the unstructured merge techniques have a strong generalization ability and can be applied into any languages, since the techniques operate only based on text. Therefore, the model version control systems leverage the unstructured merge technique, i.e., `diff3` algorithm.

Structured Techniques. Structured merge techniques represent the programs as trees or graphs (e.g., abstract syntax trees, AST), which can improve the precision compared to the unstructured techniques. They conduct tree match and tree merge to merge the conflicts. In addition, the structured merge techniques integrate the information of the specific language, such as the syntax of that language. For example, the information about the program elements whose order can be changed (e.g., the function declaration) is effective in merging many conflicts [17], [19]. Westfechtel et al. [38] and Buffenbarger et al. [20] leverage the context-free and context-sensitive information during the process of merging the conflicts. Researchers have proposed various approaches based on comparison and merge of structure towards specific languages, such as Java [21] and C++ [22]. The limitations of structured techniques are obvious, i.e., they require the knowledge of the specific language and have poor generalization ability [17], [19]. In addition, the matching and merging on the tree and graph is time-consuming, which has high computational complexity because of catching the renamings and shifted code [11].

Semi-structured Techniques. To leverage the precision of structured merge techniques and utilize the generalization ability and efficiency of unstructured merge techniques, researchers have proposed semi-structured techniques to find a balance between them. Apel et al. [17] propose FSTMerge, which is based on structured approaches and increases the generalization ability. It allows the users to add the syntax of new language to help resolve the conflicts of that language. Tavares et al. [18] adds the grammar of JavaScript to FSTMerge and obtains jsFSTMerge which can solve the conflicts of JavaScript. Apel et al. [19] propose to select between unstructured and structured merge techniques dynamically. Structured techniques are adopted when the conflicts are detected and unstructured techniques are adopted when the conflicts are not detected. Leßenich et al. [11] find that the structured merge has high computational complexity, and employs a lookahead mechanism to reduce the cost of structured merge. Previous studies find that although semi-structured techniques can reduce the number of detected conflicts [17], [39], they will neglect some conflicts and cause false negatives [9].

Learning-based Techniques. More recently, with the promising achievements of deep learning techniques in code-related tasks [32], [33], [40], researchers also adopt deep neural networks to merge conflict resolution task and propose the learning-based approaches [23], [24], which achieve the state-of-the-art performance. The learning-based techniques only leverage the text of conflicts and can also be regarded as unstructured techniques. Despite operating only based on text, the learning-based techniques have a better ability to resolve the conflicts, because they have mastered the common pattern of how to resolve by learning from the history conflict-resolution pairs. Existing learning-based techniques leverage classification to resolve conflicts. They focus on leveraging the existing contents of three input versions $\mathcal{O}$, $\mathcal{A}$, and $\mathcal{B}$, so they put forward different approaches to combine the code

of three versions. DeepMerge [23] is the first leaning-based technique, which adopts and combines the inputs at line-level. Another learning-based technique MergeBERT [24] designs several patterns to combine the inputs at token-level. Although achieving not bad performance, the existing learning-based techniques are limited because of leveraging classification to generate resolutions. They cannot generate new tokens or produce flexible combinations, and have a wrong hypothesis that fine-grained conflicts of one single coarse-grained conflict are independent.

VIII. Conclusion

In this work, we propose an automatic merge conflict resolution technique, MergeGen. Different from previous works, MergeGen produces the resolutions through ***generation***, instead of classification. Generation has great advantages towards classification, that is, it can not only generate the existing tokens of the input similar to classification, but also generate new tokens and more flexible combinations. MergeGen first represents the merge conflicts through a structural and fine-grained conflict-aware representation, and then leverages the encoder-decoder-based generative model to produce the resolutions token by token. Through the extensive evaluation, MergeGen achieves the state-of-the-art performance on all studied languages. The ablation study verifies the effectiveness of the proposed representation, and the granularity analysis reveals the tolerance to coarse-grained conflicts. Moreover, the analysis on generating new tokens further prove the advance of generative models.

ACKNOWLEDGMENTS

This work was supported by National Natural Science Foundation of China under Grant No. 62232001.

References

[1] C. Bird and T. Zimmermann, "Assessing the value of branches with what-if analysis," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[2] G. Gousios, M.-A. Storey, and A. Bacchelli, "Work practices and challenges in pull-based development: the contributor's perspective," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 285–296.

[3] 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, 2018.

[4] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 168–178.

[5] C. Costa, J. Figueirêdo, J. F. Pimentel, A. Sarma, and L. Murta, "Recommending participants for collaborative merge sessions," *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1198–1210, 2019.

[6] M. L. Guimarães and A. R. Silva, "Improving early detection of software merge conflicts," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 342–352.

[7] N. Nelson, C. Brindescu, S. McKee, A. Sarma, and D. Dig, "The lifecycle of merge conflicts: processes, barriers, and strategies," *Empirical Software Engineering*, vol. 24, pp. 2863–2906, 2019.

[8] S. Apel, J. Liebig, C. Lengauer, C. Kästner, and W. R. Cook, "Semistructured merge in revision control systems." in *VaMoS*, 2010, pp. 13–19.

[9] G. Cavalcanti, P. Borba, and P. Accioly, "Evaluating and improving semistructured merge," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–27, 2017.

[10] B. K. Kasi and A. Sarma, "Cassandra: Proactive conflict minimization through optimized task scheduling," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 732–741.

[11] O. Leßenich, S. Apel, C. Kästner, G. Seibt, and J. Siegmund, "Renaming and shifted code in structured merging: Looking ahead for precision and performance," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 543–553.

[12] T. Mens, "A state-of-the-art survey on software merging," *IEEE transactions on software engineering*, vol. 28, no. 5, pp. 449–462, 2002.

[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] F. Zhu and F. He, "Conflict resolution for structured merge via version space algebra," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.

[15] *diff3*, Linux man-pages project. [Online]. Available: https://linux.die.net/man/1/diff3

[16] Git, *git-merge*, GitHub. [Online]. Available: https://git-scm.com/docs/git-merge

[17] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "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*, 2011, pp. 190–200.

[18] A. T. Tavares, P. Borba, G. Cavalcanti, and S. Soares, "Semistructured merge in javascript systems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1014–1025.

[19] S. Apel, O. Leßenich, and C. Lengauer, "Structured merge with auto-tuning: balancing precision and performance," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 120–129.

[20] J. Buffenbarger, "Syntactic software merging," in *Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers*. Springer, 2005, pp. 153–172.

[21] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Jdiff: A differencing technique and tool for object-oriented programs," *Automated Software Engineering*, vol. 14, pp. 3–36, 2007.

[22] J. E. Grass, "Cdiff: A syntax directed differencer for c++ programs," in *C++ Conference, Portland, Oregon, Aug. 1992*, 1992.

[23] E. Dinella, T. Mytkowicz, A. Svyatkovskiy, C. Bird, M. Naik, and S. Lahiri, "Deepmerge: Learning to merge programs," *IEEE Transactions on Software Engineering*, 2022.

[24] A. Svyatkovskiy, S. Fakhoury, N. Ghorbani, T. Mytkowicz, E. Dinella, C. Bird, J. Jang, N. Sundaresan, and S. K. Lahiri, "Program merge conflict resolution via neural transformers," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 822–833.

[25] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.

[26] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[27] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[28] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[29] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[31] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu, "Commit message generation for source code changes," in *IJCAI*, 2019.

[32] J. Dong, Y. Lou, Q. Zhu, Z. Sun, Z. Li, W. Zhang, and D. Hao, "Fira: fine-grained graph-based code change representation for automated

commit message generation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 970–981.

[33] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "Treegen: A tree-based transformer architecture for code generation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 05, 2020, pp. 8984–8991.

[34] L. Dong and M. Lapata, "Language to logical form with neural attention," *arXiv preprint arXiv:1601.01280*, 2016.

[35] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[36] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[37] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, 2020, pp. 38–45.

[38] B. Westfechtel, "Structure-oriented merging of revisions of software documents," in *Proceedings of the 3rd international workshop on Software configuration management*, 1991, pp. 68–79.

[39] G. Cavalcanti, P. Accioly, and P. Borba, "Assessing semistructured merge in version control systems: A replicated experiment," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2015, pp. 1–10.

[40] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th conference on program comprehension*, 2018, pp. 200–210.