applied sciences

Article

Mining on Students' Execution Logs and Repairing Compilation Errors Based on Deep Learning

Ruoyan Shi, Jianpeng Hu * and Bo Lin

School of Electronic and Electrical Engineering, Shanghai University of Engineering Science, Shanghai 201600, China; m325121516@sues.edu.cn (B.L.)
* Correspondence: mr@sues.edu.cn

Abstract: Automatic program repair techniques based on deep neural networks have attracted widespread attention from researchers due to the high degree of automation and generality. However, there is a scarcity of high-quality labeled datasets available for training program repair models. This study proposes a method of mining reasonable program repair examples from student program execution logs. Additionally, we introduce the Rookie Simulator (RS), which simulates the error patterns commonly made by novice programmers and generates a large number of program repair sample pairs. To address the issue of low repair rates for infrequent and complex error patterns in compilation errors, the study proposes the attention-enhanced capsule network for program repair (ACNPR), a program repair model that integrates compiler feedback information and utilizes capsule networks to capture complex semantic features. Experimental evaluations were conducted using publicly available datasets, including the DeepFix, TEGCER, and a real course dataset named SUES-COJ mined in this study. The results indicate that our method consistently outperforms current state-of-the-art models in terms of full repair rates.

Keywords: automatic program repair; capsule network; compilation error repair

check for updates

Citation: Shi, R.; Hu, J.; Lin, B. Mining on Students' Execution Logs and Repairing Compilation Errors Based on Deep Learning. *Appl. Sci.* **2023**, *13*, 9933. https://doi.org/10.3390/app13179933

Academic Editor: Chilukuri K. Mohan

Received: 4 August 2023
Revised: 24 August 2023
Accepted: 29 August 2023
Published: 2 September 2023

1. Introduction

During the programming process, program errors are quite common and inevitable. Specifically, syntax errors, such as the misuse of delimiters and identifiers, frequently occur in programs written by students and novice programmers. Programming assignments are a crucial method for practicing programming skills, and the scale of such laboratory courses is gradually increasing [1]. Through online systems, code execution logs of students completing programming tasks can be obtained. The execution logs comprehensively include students' processes, from starting to write and compile the program, discovering compilation errors, finding the reasons for the errors, and making modifications to eventually repairing the program's errors successfully and submitting it. Hence, the logs contain common syntax errors made by novices, as well as the modified programs. These program samples can serve as examples for program error repair, guiding the process of fixing programs.

Traditional automated program repair approaches mostly rely on search-based or semantics-based techniques [2], which heavily depend on manually designed heuristic search algorithms [3–5] or methods with semantic constraints [6,7] to generate reasonable repair patches. These approaches have low automation levels and limited generality. In recent years, more repair tools have been developed based on deep learning models that extract syntactic and semantic features from program code. By training repairers on large-scale code data, these models are capable of handling various types of code errors and have higher levels of automation and practicality potential.

The essence of deep learning–based program error repair methods is statistical analysis–based repair techniques, which rely on the quality of the dataset and the model's ability to capture program syntactic features. It is crucial to find truly applicable code samples from massive open-source data and meet the data quantity requirements for training deep learning models. Currently, there is a scarcity of labeled datasets for training program repair models, and manually annotating the corresponding correct repairs for each erroneous program is a labor-intensive task. Several studies utilize perturbation methods to randomly introduce, modify, or delete parts of critical code to generate a large number of erroneous programs and their correct repairs. However, the quality of the data obtained through this approach is inferior to real error data, leading to suboptimal repair effectiveness [8].

Relying solely on code semantics information to train repair models has limited capability in capturing features of low-frequency errors and deep semantic errors. Incorporating compiler feedback information can assist such models in effectively extracting certain types of error patterns, but these patterns are usually simple [9] and do not involve complex semantic code structures.

This study aimed to overcome the aforementioned challenges by mining pairs of samples from students' code execution logs specifically used for program repair. By so doing, real program repair examples were obtained. Furthermore, a deep learning–based approach was employed to augment the quantity of program repair samples. The augmented dataset was then used to train a program repair model capable of fixing compilation errors. The main contributions of this study are outlined as follows:

1. We propose a method for mining reasonable program error repair examples from students' program execution logs in the online experimental system. This method enables the extraction of a large number of program code pairs that encompass multiple types of errors, which can be used for training program repair models.
2. To address the issue of insufficient program error repair sample data, we introduce the Rookie Simulator (RS), which simulates the error-prone habits of novice programmers. Using deep learning techniques, the RS is trained to predict and generate additional labeled program pairs. These program pairs closely resemble the types and distribution of real errors. By training the repair model with these generated samples, improved repair effectiveness can be achieved.
3. To address the issues of low-frequency errors and unclear error description information in program repair, we introduce the attention-enhanced capsule network for program repair (ACNPR) model. This model combines compiler error description information and features such as program-feedback graphs to repair erroneous programs. Experimental results demonstrate that the ACNPR model achieves satisfactory repair results for various types of novice errors.

2. Related Work

Gupta et al. [10] first proposed using the sequence-to-sequence neural network model DeepFix to automatically fix syntax errors in programs, achieving an end-to-end automated repair tool. This method focuses on four common types of simple syntax errors and introduces mutations to programs, which are represented as sequences of line numbers and corresponding statements. The model was trained to predict individual repairs and employs an iterative approach to fix programs with multiple errors. Subsequently, Gupta et al. [11] introduced a reinforcement learning–based repair framework for fixing student programs. The framework utilizes long short-term memory networks to encode program texts and incorporates cursor positions. An agent performed a series of cursor navigation and editing operations to simulate the step-by-step program repair process executed by students. Hajipour et al. [12] proposed SampleFix, which samples various repairs for a given erroneous program. It learned the distribution of latent patches with the use of a conditional variational autoencoder and generated repair patches by editing the latent lines of the errors, effectively correcting common syntax errors. These methods rely solely on the semantic guidance of the code for program repair without utilizing feedback information from the compiler, thus limiting their repair capabilities.

Yasunaga et al. [13] utilized diagnostic feedback information from compilers and proposed the program-feedback graph to address long-distance dependencies of variables. The authors also introduced a self-supervised learning paradigm for program repair, creating a large amount of additional training data by intentionally introducing five types of random perturbation modules to unlabeled programs. Their final system, DrRepair, achieved advanced complete repair rates. Mesbah et al. [14] proposed DeepDelta, which employs deep neural networks to learn repair patterns for specific types of errors. The authors transformed error repair patterns into a domain-specific language called Delta and trained a neural machine translation network using compiler feedback information as the source and incremental changes of code repairs as the target. This method effectively generates repair patches for two commonly occurring and costly Java compilation errors. Seo et al. [15] presented a sequence-to-sequence learning framework called MultiFix, which allows for repairing multiple errors at once. It pairs the best-aligned erroneous program with the corresponding correct program generated from the edit distance calculation to label repair examples. By taking the error code with positional encodings as input, the predicted repair patch can handle multiple errors across lines, eliminating the need for iterative iterations and improving repair efficiency. The aforementioned methods utilize specific patterns for mining or perturbing to generate error programs as the training dataset. The effectiveness of repairing different types of errors is constrained by the quality and diversity of the training dataset. Our approach involved utilizing a learning-based method to produce error programs that better conformed to the distribution of actual error locations and types. This allowed us to increase the training samples while preserving a greater amount of semantic information within the actual erroneous code. Compared to the methods of random program mutation and design perturbation, our approach does not rely on manual analysis of program error types in different categories of datasets. Instead, our method was capable of automatically generating error examples corresponding to specific types. For instance, Chinese students often negligently used Chinese characters that the compiler couldn't recognize instead of English characters while programming. By utilizing the corresponding dataset, our method was able to simulate such errors and generate a substantial number of examples without the need for manually designing templates or mutating programs specific to this type of error.

Ahmed et al. [16] proposed TEGCER as an automated feedback tool for novice programmers. The authors employed supervised classification to match compilation errors with related errors submitted by other students. In total, 212 error categories were identified, and corresponding solution examples were provided. Subsequently, Chhatbar et al. [17] introduced MACER—which separates the repair process into type identification and repair application, resulting in faster repair speeds. These approaches rely on fine-grained categorization of error types and require the extraction of templates from repair examples. Since they were built based on a specific dataset to create repair categories, their effectiveness was limited when applied to datasets submitted by programmers with varying levels of programming expertise. Our approach directly completed the program repair in an end-to-end manner based on the input features, without any pre-defined manual categorization. This high level of automation allowed for simulating different categories of errors made by different types of programmers. Additionally, our approach demonstrated greater generality across different datasets.

To overcome the limitations of convolutional neural networks in capturing the relative positional relationships between different features, Sabour et al. [18] introduced a new neural network architecture called the capsule network. Instead of scalar neurons as the basic computing units, this model utilizes vector neurons, where both the input and output are vectors. Each entity is represented by the orientation of the vector, while the length of the vector represents the confidence of belonging to a specific class, and each dimension of the vector represents specific attributes of the entity. The architecture of the capsule network primarily consists of convolutional, primary capsule, and digit capsule layers, as well as a decoder. The information flow between the primary capsule and digit capsule layers is facilitated by a dynamic routing algorithm. While capsule networks are mostly applicable in image processing, Zhao et al. [19] applied such a network to text classification and achieved better performance than CNNs and RNNs in multilabel analysis tasks. Jia et al. [20] proposed a capsule network enhanced with multi-head attention for text classification. This approach enabled the model to encode long-distance dependency relationships between words and to merge them with semantic information through the capsule network. In our program repair model, we applied capsule networks to extract high-order features from two different sources: the semantic information of program code and the semantic information in compiler descriptions. By utilizing two types of capsule networks and incorporating them into the contextual semantics through an attention mechanism, our model was able to learn the deep-seated semantics of program errors. This approach enhanced the effectiveness of repairing complex semantic errors.

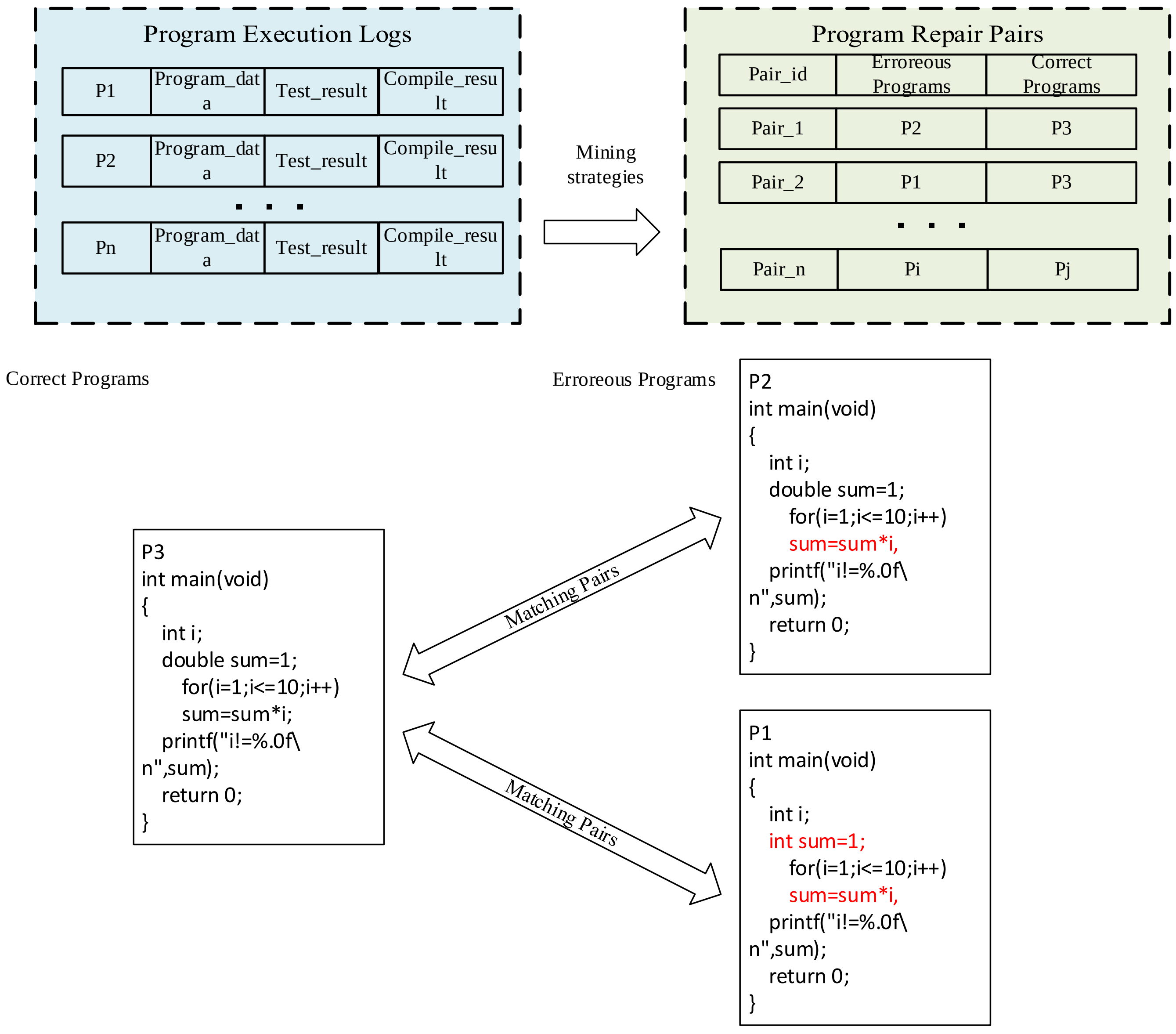
The method employed in this research involved mining real errors made by students, followed by simulating their error-prone habits to generate a large and diverse set of program repair examples for training. Compiler feedback information was incorporated into the process. By leveraging the graph neural network and capsule network, the model was able to fuse with code semantic information, enabling it to capture complex and diverse semantic information more effectively.

3. Mining on Students' Execution Logs

In computer language education courses for university students, to assess and reinforce their learning outcomes and improve learning efficiency, instructors assign programming tasks within an online experimental system. The system provides feedback and assigns scores based on the programs submitted by the students—who complete programming tasks online, including writing, compiling, and submitting their code. Therefore, the online experimental system captures the real-time execution logs of the students' programs. These records contain detailed information, ranging from the initial stages of program writing and compiling to identifying compilation errors, troubleshooting, and ultimately fixing the program errors successfully before submission. The required error-repair program pairs for training program repair models can be mined from these program execution logs.

The C program source code written by students contains code comments and unnecessary blank lines. These redundant codes have no impact on the program semantics and increase the code length, reducing the program's semantic coherence. In this study, regular expressions were used to replace multiline and single-line comments with empty characters and to match and delete lines in the program that only contain whitespace characters (spaces, line breaks, and tabs).

The program execution logs contain sequentially compiled code for different programming tasks, along with their testing results on the compiler and test case. Based on the submitted correct code that passed the tests, it is possible to retrospectively search for relevant erroneous code that was compiled earlier. Taking Figure 1 as an example, the program execution sequence of the student goes from P1 to Pn. Initially, the student executed P1, but it didn't pass the compilation. They made a modification from "int sum" to "double sum" and recompiled it, but it still didn't pass. Then, they made another modification from "sum=sum*i," to "sum=sum*i;", and it compiled successfully. The method for mining program pairs starts by sequentially searching the execution logs to find a correct program sample, denoted as P3. Then, employing various strategies, the search goes backward from P3. Among the previously submitted programs, error examples corresponding to those that failed compilation are sought. Once the error programs (P2 and P1) corresponding to P3 are found, they are paired with P3. These program pairs are then recorded as program repair examples.


Program Execution Logs
P1 | Program_data | Test_result | Compile_result
P2 | Program_data | Test_result | Compile_result
Pn | Program_data | Test_result | Compile_result
Mining strategies
Program Repair Pairs
Pair_id | Erroreous Programs | Correct Programs
Pair_1 | P2 | P3
Pair_2 | P1 | P3
Pair_n | Pi | Pj
Correct Programs
Erroreous Programs
Matching Pairs
Matching Pairs

```
P3
int main(void)
{
   int i;
   double sum=1;
      for(i=1;i<=10;i++)
      sum=sum*i;
   printf("i!=%.0f\
n",sum);
   return 0;
}
```

```
P2
int main(void)
{
   int i;
   double sum=1;
      for(i=1;i<=10;i++)
      sum=sum*i,
   printf("i!=%.0f\
n",sum);
   return 0;
}
```

```
P1
int main(void)
{
   int i;
   int sum=1;
      for(i=1;i<=10;i++)
      sum=sum*i,
   printf("i!=%.0f\
n",sum);
   return 0;
}
```

Figure 1. Example of program pairs mining. The specific error lines in the "Erroreous Programs" are highlighted in red.

We propose three different strategies for mining program repair sample pairs, and the specific descriptions of these strategies are as follows:

Strategy 1 involves searching through the execution logs based on the compilation results of the compiler to find programs that compiled successfully. To ensure that the programs have some semantic information, the first code that passes a portion of the test cases is chosen as the correct sample. Then, starting from the correct sample, the execution logs are examined to find the last instance of code that failed to compile before this program. This code is considered as the error sample. Finally, the error sample is paired with the correct sample.

Strategy 2 involves searching through the execution logs to find the first code that passes all the test cases, indicating that it semantically aligns with the programming task. This code is selected as the correct sample. Then, starting from the correct sample, the execution logs are examined to find the last instance of code that failed to compile. A text-diff tool is used to identify the differences between the two code segments. Only the differing lines within a span of up to three lines are retained, while the remaining content is replaced with the correct code sample. By replacing the erroneous code with the correct code while preserving the differing parts, the resulting code retains the differential aspects of the error code and incorporates the correct code to cover the other parts. This approach ensures that the differences between the sample pairs are concentrated.

Strategy 3, as shown in Algorithm 1, involves traversing the execution log L to find the program p that passes all test cases. Then, we iterate backward through the execution log, identifying all previously encountered code that failed to compile. For each such code, we calculate the difference with the correct sample, considering only one differing code block, with a span of no more than three lines. Any code meeting these conditions is identified as an error sample. The temporary variables pl and pt are used to store the programs encountered during the traversal process. The function *GetPreProg(L, pl)* retrieves the previous code record for program pl from the execution log L and assigns it to pt. Similarly, *GetDiffBlock*() and *GetDiffLine*() return the number of differing code blocks and the span of differing lines between the two programs. The line s = *Match(pt, p)* represents pairing the error program pt with the correct program p, creating a repair example program pair s. Finally, these program pairs are collected as the set of all program pairs, denoted as S.

The program pairs obtained through three different strategies are further screened by restricting the Levenshtein distance [21] between program pairs. This filtering process aims to remove examples with excessively large modifications, such as deleting an entire line. Finally, the program pairs undergo manual review to exclude any unreasonable repair examples.

Algorithm 1: Program pair mining algorithm

```
Input: program execution logs $L = \{ P_1, P_2, \ldots, P_n \}$
Output: program pair set $S = \{S_1, S_2, \ldots, S_n\}$
for $p$ in $L$ do
  if PassAllTests($p$) then
    $pl = p$
    while $pl$ is not $P_1$ do
      $pt$ = GetPreProg($L$,$pl$)
      if not Compiled($pt$) then
        if GetDiffBlock($pt$,$p$) = 1 and GetDiffLine($pt$,$p$) $\leq 3$ then
          $s$ = Match($pt$,$p$)
          add $s$ to the set $S$
        end if
        $pl = pt$
        continue
      else
        $p = pt$
        break
    end while
  end if
end for
return $S$
```

Table 1 shows the result of program pairs unearthed by applying the three methods to different execution logs submitted by students for various programming tasks. Strategy 1 has simple constraint conditions, resulting in the highest number of sample pairs mined and encompassing the widest range of error types. However, on average, each sample pair contains a larger number of errors and exhibits greater variability in modifications, leading to a higher proportion of unreasonable repairs. Strategies 2 and 3 yield a smaller number of sample pairs, but they demonstrate higher repair quality. Strategy 2's mined samples encompass fewer error types and have a lower diversity, resulting in lower repair complexity. On the other hand, Strategy 3 consistently produces samples of stable quality across different programming tasks, with a higher number of errors and a certain level of repair complexity.

Table 1. Result of program pairs mining.

Task	Mining Strategy	Amount of Sample Pairs	Amount of Error Types	Average Number of Errors per Sample Pair
Task1	Strategy 1	140	5	1.55
	Strategy 2	24	4	1.21
	Strategy 3	15	4	1.47
Task2	Strategy 1	407	7	1.81
	Strategy 2	280	4	1.18
	Strategy 3	303	6	1.48
Task3	Strategy 1	343	7	2.14
	Strategy 2	207	3	1.08
	Strategy 3	192	6	1.44

4. Error Program Generating

4.1. Code Tokenization

The type names, delimiters, library functions, keywords, and special symbols in the C language are universal, so they were preserved as individual tokens. Specifically, the identifiers within the code were retained, including variable names, function names, and other symbols, as they played a significant role in shaping the semantics of the code. The newline characters were abstracted and represented as the token <newline> to preserve the line information of the program. The various constants in the code do not alter the code syntax, and the exact values of constants are not important for the model's learning task. Therefore, the numerical constants in the code were abstracted as a token <number>, the string constants were abstracted as a token <string>, and the character constants were abstracted as a token <char>. We considered all the code in a program as a token sequence, and we used a dictionary to map all the constants that appeared in the program code. This allowed us to restore the abstracted code by replacing the mapped tokens with their original constants.

4.2. Rookie Simulator

The main workflow of the RS for program pairs generation is illustrated in Figure 2. Transformer models [22] have shown remarkable performance in natural language processing tasks, such as machine translation and speech recognition. The research conducted by Michele et al. [23] demonstrates that trained neural machine translation models can acquire a significant number of distinct bug patterns and generate patches for bug fixes. Due to the inherent similarity between syntax errors in programming and grammar errors in natural language [10], we draw an analogy to grammatical error correction in natural language. To this end, we utilize real erroneous program statements, along with their corresponding correct fix examples, as a dataset for training the Transformer translation model.

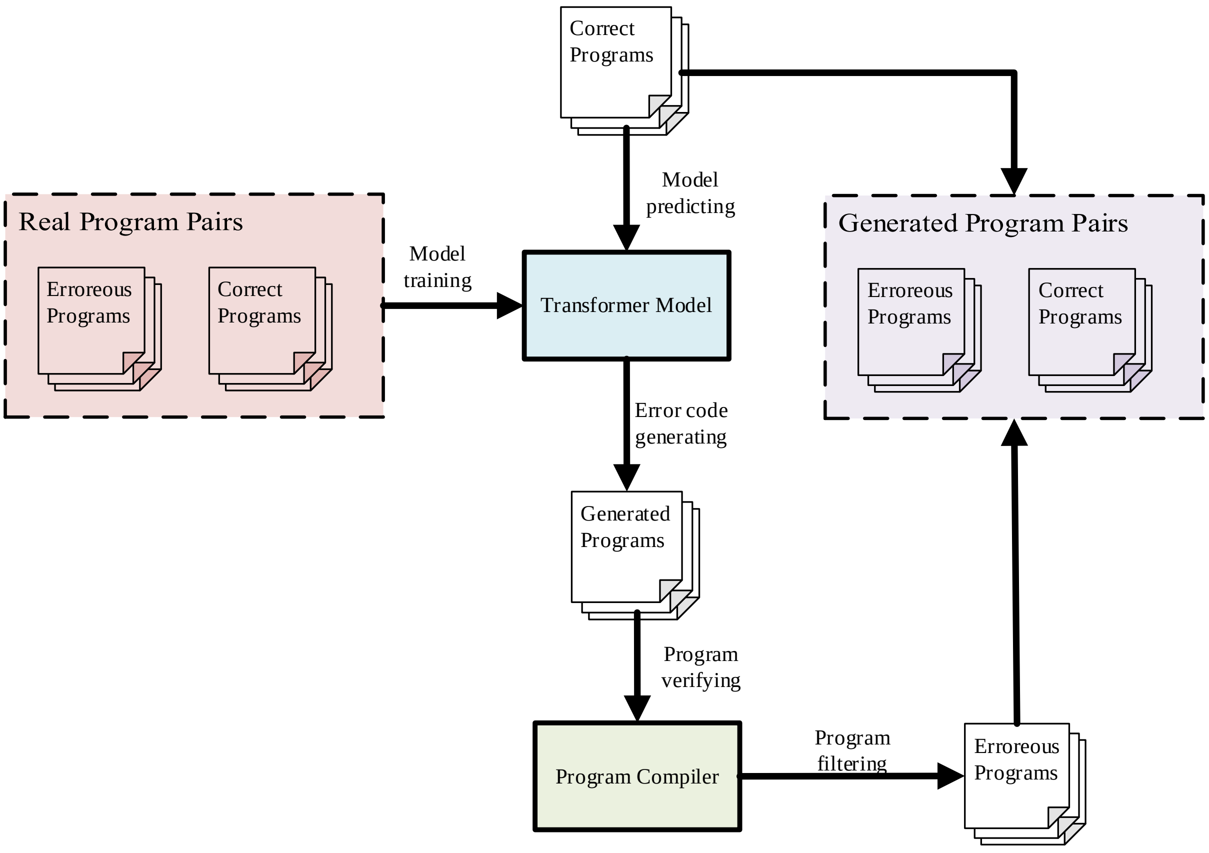
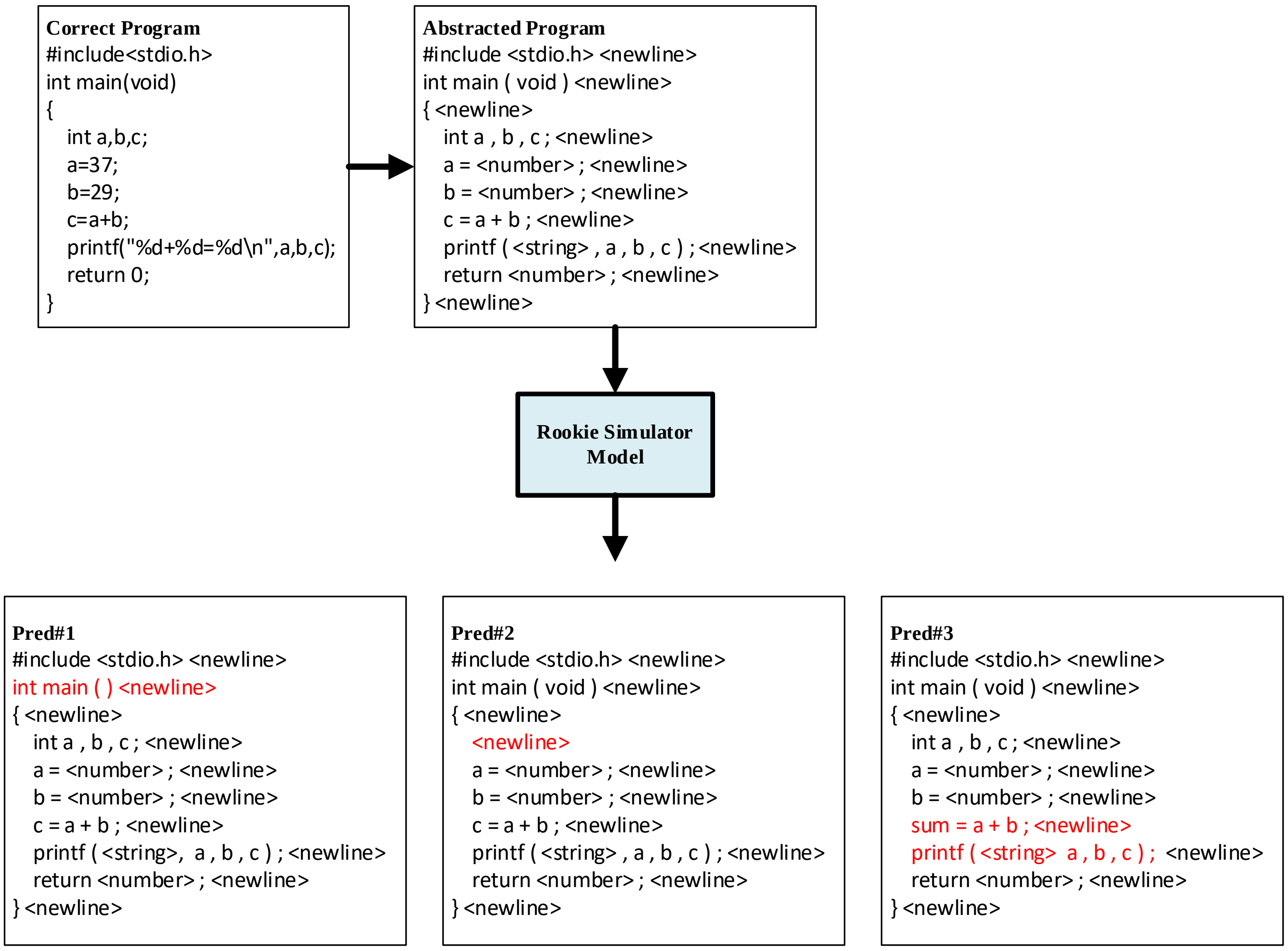


Figure 2. Flowchart of Rookie Simulator.

During the model training phase, the correct programs are used as the source data, while the erroneous programs serve as the target data. The Transformer model is trained to predict and generate erroneous program statements based on the correct code. The generated program statements are then compiled to verify their correctness. The erroneous programs that result in compilation errors are retained as the output. The retained code statements are filtered based on an edit distance [21] threshold of less than or equal to 5. These filtered statements are then paired with the corresponding source program code statements used during the prediction phase of the model, thereby creating the generated program pairs dataset.

In the prediction phase, taking Figure 3 as an example, the correct program is first abstracted and input into the trained RS model. This generates multiple predicted erroneous programs, denoted as Pred#n. Next, all generated program statements are fed into a compiler for validation. Since “int main()” and “int main(void)” are equivalent and do not result in compilation errors, Pred#1 is filtered out at this stage. Subsequently, during the edit distance filtering step, Pred#2, which has deleted a line of content, exceeds the edit distance threshold when compared to the input program. However, Pred#3 has an edit distance of 2 compared to the input program and is ultimately paired with the correct program, resulting in a generated program repair example.



Correct Program
```
#include<stdio.h>
int main(void)
{
   int a,b,c;
   a=37;
   b=29;
   c=a+b;
   printf("%d+%d=%d\n",a,b,c);
   return 0;
}
```

Abstracted Program
```
#include <stdio.h> <newline>
int main ( void ) <newline>
{ <newline>
   int a , b , c ; <newline>
   a = <number> ; <newline>
   b = <number> ; <newline>
   c = a + b ; <newline>
   printf ( <string> , a , b , c ) ; <newline>
   return <number> ; <newline>
} <newline>
```

Pred#1
```
#include <stdio.h> <newline>
int main ( ) <newline>
{ <newline>
   int a , b , c ; <newline>
   a = <number> ; <newline>
   b = <number> ; <newline>
   c = a + b ; <newline>
   printf ( <string>,  a , b , c ) ; <newline>
   return <number> ; <newline>
} <newline>
```

Pred#2
```
#include <stdio.h> <newline>
int main ( void ) <newline>
{ <newline>
   <newline>
   a = <number> ; <newline>
   b = <number> ; <newline>
   c = a + b ; <newline>
   printf ( <string> , a , b , c ) ; <newline>
   return <number> ; <newline>
} <newline>
```

Pred#3
```
#include <stdio.h> <newline>
int main ( void ) <newline>
{ <newline>
   int a , b , c ; <newline>
   a = <number> ; <newline>
   b = <number> ; <newline>
   sum = a + b ; <newline>
   printf ( <string>  a , b , c ) ;  <newline>
   return <number> ; <newline>
} <newline>
```

Figure 3. Example of model predictions. The modifications made to the input program in the predicted programs are highlighted in red font.

5. Repairing Compilation Errors

The feedback information provided by the compiler can assist in pinpointing the precise location of errors, offering relevant prompts pertaining to the errors, and effectively guiding program repair. To extract the underlying semantics embedded within code statements and compiler feedback information more effectively, we propose the ACNPR model—which aims to learn the intricate characteristics and syntactic-semantic relationships of erroneous programs, ultimately generating repair patches.

5.1. Data Preprocessing

To obtain a richer set of program error feature information and facilitate the localization and reasoning capabilities of the repair model, we performed data preprocessing on the program repair dataset. This included line tokenization, compiler compilation, and the construction of program-feedback graphs.

We divided the program into lines and assigned a unique number to each line. Based on the repair examples in the dataset, we identified the lines containing errors and labeled them accordingly. This process allowed us to obtain the line numbers corresponding to the errors and their corresponding repair patches on a single line basis.

We compiled the erroneous programs in the dataset using the GNU compiler collection and extracted the compiler feedback information. Each feedback from GCC contains source file information, error location information, and error description. For example, in the feedback message "e158_277497.c:10:18: expected identifier or '(' before 'return,'" "e158_277497.c" represents the name of the compiled C program, "10" indicates the line number where the error is identified by the compiler, and "18" represents the column number of the error location. The phrase "expected identifier or '(' before 'return'" is the compiler's description of the error. We extracted the line number indicating the location of the error as guidance for the repair model to locate the errors. Additionally, we extracted the error description information as a part of the model input.

The construction of program-feedback graphs is guided by Yasunaga et al. [13] and involves identifying key information (e.g., "return") from error descriptions and locating the corresponding tokens in the source code. These tokens are then connected to form the graph. For each program, a program-feedback graph is built, consisting of inter-connected subgraphs that represent different key symbols. This graphical representation effectively abstracts the relationships between the same key symbols at different locations.

5.2. Model Architecture

The given input—which includes the program code along with the corresponding compiler error information, such as line number and error description—can be trans-formed into a sequential input model. Additionally, the program-feedback graph is constructed to directly connect important symbols related to program repairs. The source code semantics, error message features, and program-feedback graph features are then aggregated. An attention-enhanced capsule network is employed to fuse higher-order features and reconstruct semantic information, and ultimately, the decoding layer outputs the predicted error line number and the corresponding repair sequence. The error line index in the model's output corresponds to the line where the predicted modification patch is located, which is different from the input compiler error line.

The overall architecture of the ACNPR model is illustrated in Figure 4. The model takes the program statement sequence $x_i = (x_{i1}, x_{i2}, \ldots, x_{ij})$, error line number i_{err}, and error description sequence $M_{err} = (M_1, M_2, \ldots, M_n)$ as input and predicts the error line number k and the corresponding repair code sequence $y_k = (Y_1, Y_2, \ldots, Y_m)$ as output. The values of line number i and the length of each line sequence j are variable and depend on the maximum values within the current batch. The loss function used in the model is the standard negative log-likelihood. The model structure mainly consists of four parts: the encoding, graph attention, capsule network, and decoding layers.

The encoding layer encodes the source code sequence and the compiler error information, resulting in a hidden state $\mathbf{h}$. The graph attention layer incorporates the symbol association information of the source code and error information, based on the constructed program-feedback graph, into the hidden state $\mathbf{h}$ through attention, yielding a corresponding state $\mathbf{g}$. The capsule network layer utilizes a capsule network to aggregate higher-order features. It fuses the captured feature values into the semantics, incorporating the contextual semantic information of the source code and that of the compiler error information through attention, resulting in an output state $\mathbf{o}$. The decoding layer decodes based on the hidden state obtained from the upper layers, predicting the probability distribution of the error line number and the corresponding repair patch.

5.2.1. Encoding Layer

Given the source code sequence $x_i = (x_{i1}, x_{i2}, \ldots)$, we encoded it at the line level using a bidirectional long short-term memory (BiLSTM) network denoted as $\text{BiLSTM}^{(1)}_{\text{code}}$, which outputs the hidden state $\mathbf{h}$. Additionally, given the compiler error description sequence $M_{err} = (M_1, M_2, \ldots)$, we encoded it using another BiLSTM network, denoted as $\text{BiLSTM}^{(1)}_{msg}$, which outputs the corresponding hidden state $\mathbf{h}_{M_l}$. Based on the error line number i_{err} indicated by the compiler, we calculated the position offset $\Delta i = i_{err} - i$ for each code line relative to this error line. To incorporate this position offset information into the source code sequence, we used position embedding. Each position offset Δi was concatenated with the corresponding hidden state $\mathbf{h}$ of the code sequence, resulting in the final representation $\mathbf{h}_{x_{ij}}$ for each line of code.

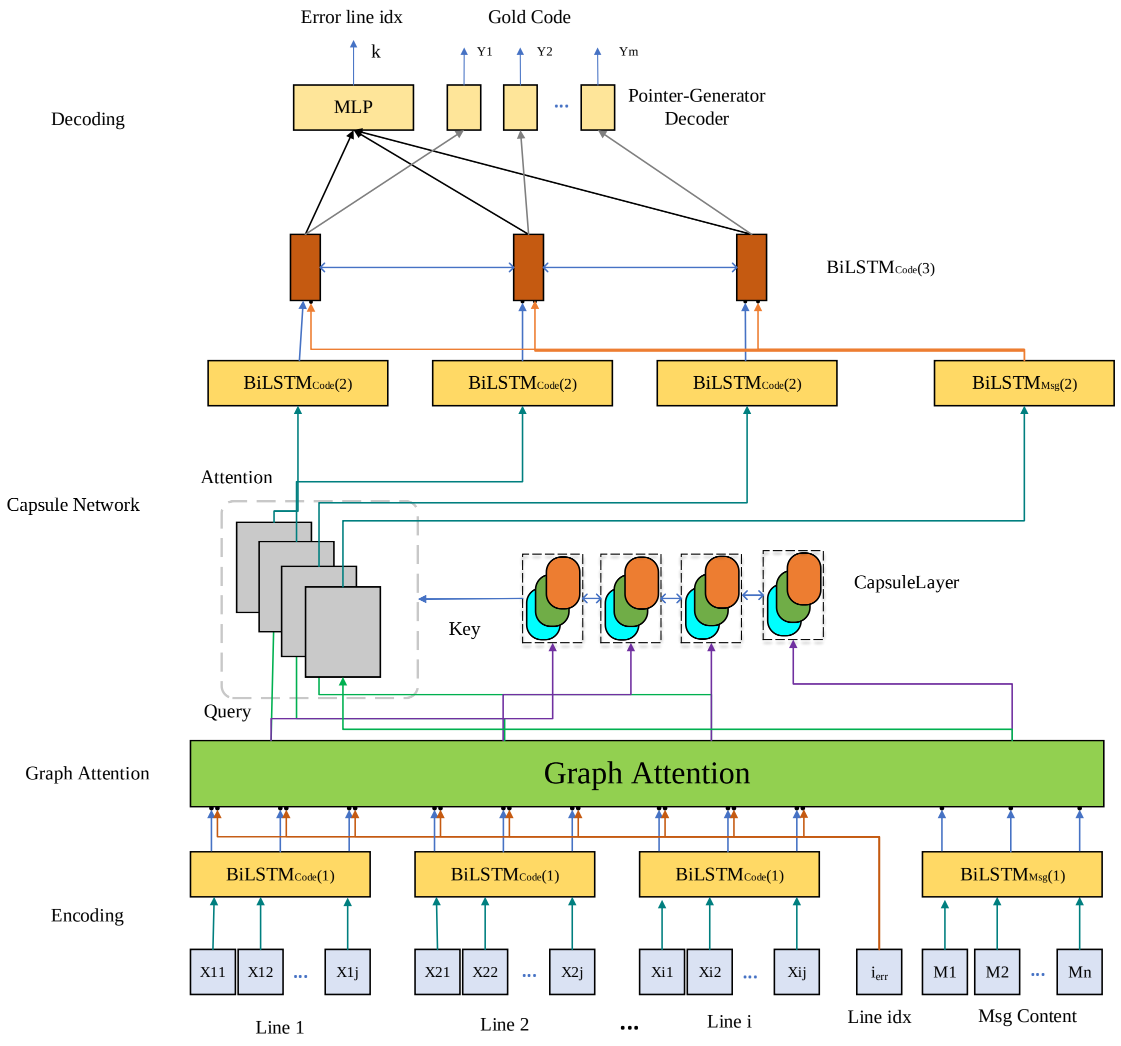


Figure 4. Model architecture of ACNPR.

Indeed, after encoding the code sequence with the position embedding and incorporating the position offset information, the resulting code context vectors contained the relevant information associated with the error position. Additionally, the compilation error description information had been encoded as well, enabling the subsequent integration of corresponding semantic information. This encoding process facilitated capturing the contextual relationships between the code lines and the error position, as well as the semantic representations of the error descriptions, ultimately aiding in the subsequent steps of the model.

5.2.2. Graph Attention Layer

To enable the model to learn the correspondence between compiler error descriptions and the source code, as well as to track key symbols within the source code sequence and maintain relevant syntactic dependencies, a graph attention network [24] was used. This network allows the associated information contained within the constructed program-feedback graph to be propagated within the context, enabling the model to simulate the process of symbol tracking for program repair. Based on the adjacency relationships among symbols in the program-feedback graph, the weight calculation for the multi-head attention of each adjacent symbol to a given symbol is conducted, and the state information is updated accordingly. Each layer is computed using the following formula:

$$\mathbf{c}^n = \mathrm{Attention}_G(\mathbf{h}^{n-1}) \tag{1}$$

$$\mathbf{h}^n = \mathrm{MLP}\left(\left[\mathbf{h}^{n-1}; \mathbf{c}^n\right]\right) \tag{2}$$

where $\mathbf{h}^{n-1}$, $\mathbf{h}^n$ denote the input/output representation of each token at the n-th layer, including the code state information $\mathbf{h}_{x_{ij}}$ and the error description information $\mathbf{h}_{M_l}$. Attention$_G(\mathbf{h}_t)$ is used to calculate the attention weights of symbol t on the program-feedback graph G with its neighboring nodes and then takes their weighted average. A gating mechanism is incorporated in the graph attention layer to replace the activation function, preserving the scale of each dimension and enhancing non-linear capabilities. MLP refers to a feedforward network used to integrate the attention weights into the context vector, resulting in the final output state information represented as $\mathbf{g}_{x_i}$ and $\mathbf{g}_m$.

5.2.3. Capsule Network Layer

After updating through the graph attention layer, the hidden state includes the semantic features of the input code sequence and the compiler feedback information from the updated program-feedback graph. The dimensions of these vectors are batch size $\times$ sequence length $\times$ hidden state dimension. The capsule network is used to detect the internal states of the input code sequence features, encapsulating them in vector form and resulting in output vectors representing the corresponding probabilities and characteristics of higher-level features. The internal weights of the capsules are learned using dynamic routing, where the weights determine how vectors from the lower layer will enter the higher-level vectors. The calculation of dynamic routing can be described by the following formula:

$$\mathrm{c}_{ij} = \mathrm{softmax}(b_{ij}) \tag{3}$$

$$\hat{\boldsymbol{u}}_{j|i} = \boldsymbol{W}_{ij}\boldsymbol{u}_i \tag{4}$$

$$\boldsymbol{s}_j = \sum_i c_{ij}\hat{\boldsymbol{u}}_{j|i} \tag{5}$$

$$\boldsymbol{v}_j = f(\boldsymbol{s}_j) = \frac{\|\boldsymbol{s}_j\|^2}{1+\|\boldsymbol{s}_j\|^2}\frac{\boldsymbol{s}_j}{\|\boldsymbol{s}_j\|} \tag{6}$$

$$\mathbf{h}_v = [\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_N] \tag{7}$$

where c_{ij} represents the coupling coefficients, indicating the routing probabilities from the lower-order capsules to the higher-order capsules, and $\hat{\boldsymbol{u}}_{j|i}$ represents the output vector of the i-th capsule in the previous layer, which is obtained by multiplying the input vector $\boldsymbol{u}_i$ with the corresponding transformation matrix $\boldsymbol{W}_{ij}$. $\boldsymbol{s}_j$ represents the weighted sum of all the output vectors from the previous layer's capsules to the current layer, and $\boldsymbol{v}_j$ represents the output vector of the capsule. The function f, also known as the squash function, compresses the element values of the corresponding vectors into the range of 0 to 1. After the aforementioned calculations, the output vector $\mathbf{h}_v$ is obtained. The dimension of $\mathbf{h}_v$ is N $\times$ capsules' dimension, where N represents the number of capsules. We ensure that the dimension of the capsule network matches the dimension of the input hidden state to facilitate its merging with other semantic information. Due to the limited correlation between different batch samples, we have added a regularization layer with LayerNorm [25] after the capsule network layer. This layer only normalizes the dimensions within each sentence, aiming to reduce overfitting and accelerate the convergence of the model.

We employed an attention mechanism to enhance the propagation of higher-order feature information within the context, resulting in the generation of the final hidden states $\mathbf{g}_{x_i}{}^{final}$ and $\mathbf{g}_m{}^{final}$. The purpose of the attention calculation is to allocate importance to different capsule output vectors based on their relevance within the input, thereby enhancing the fusion of higher-order feature information and generating the final hidden state. The following steps illustrate the calculation process of attention using the hidden state ${\mathbf{g}_{x_i}}^{final}$ as an example. Similarly, the attention calculation process for ${\mathbf{g}_m}^{final}$ is derived following the same procedure.

$$\boldsymbol{d}_k = f(\boldsymbol{W}_k \mathbf{h}_v + b_k) \tag{8}$$

$$\boldsymbol{d}_q = f\left(\boldsymbol{W}_q \mathbf{g}_{x_i} + b_q\right) \tag{9}$$

$$\boldsymbol{d}_v = f\left(\boldsymbol{W}_v \mathbf{g}_{x_i} + b_v\right) \tag{10}$$

$${\mathbf{g}_{x_i}}^{final} = \boldsymbol{d}_v^{\mathbf{T}} \cdot \mathrm{softmax}\left(\frac{1}{\sqrt{n_v + n_s}} \boldsymbol{d}_q \boldsymbol{d}_k^{\mathbf{T}}\right) \tag{11}$$

After updates from the graph attention and capsule network layers, the information is propagated in the respective local contexts using two types of BiLSTM models. One type of model, referred to as $\mathrm{BiLSTM}^{(2)}_{code}$, is used for the code statement sequences, while the other, $\mathrm{BiLSTM}^{(2)}_{msg}$, is applied to the error description sequences. As a result, the final hidden state $\mathbf{r}_i$ is obtained for each line i.

$$\mathbf{r}_i = \left[\mathrm{LSTM}^{(2)}_{code}\left(\mathbf{g}_{x_i}\right)^{final}; \mathrm{LSTM}^{(2)}_{msg}(\mathbf{g}_m)^{final}\right] \tag{12}$$

The final hidden states are fused using the $\mathrm{BiLSTM}^{(3)}_{code}$ to obtain a merged embedding sequence $\mathbf{o}_{1:L}$ that encompasses embeddings from different lines.

$$\mathbf{o}_{1:L} = \mathrm{LSTM}^{(3)}_{code}(\mathbf{r}_{1:L}) \tag{13}$$

5.2.4. Decoding Layer

Given the hidden states $\mathbf{o}_{1:L}$ obtained from the previous layer, we modeled the probabilities of each line $k \in \{1, 2, \ldots, \mathrm{L}\}$ being an erroneous line using a feedforward network. Additionally, we employed a pointer-generator network [26] as a decoder to model the probability distribution of the repair sequence y_k for the erroneous lines, aiming to address the issue of numerous out-of-vocabulary words caused by different variable names. The formulas are as follows:

$$p(k|\mathbf{o}_{1:L}) = \mathrm{softmax}(\mathrm{MLP}(\mathbf{o}_{1:L})) \tag{14}$$

$$p(y_k|\mathbf{o}_{1:L}) = \mathrm{PtrGen}(\mathbf{o}_k) \tag{15}$$

During the final model prediction, the model selects the erroneous line index k with the highest probability as the error localization result. To generate potential repair patches, beam search is used to output sequences y_k with higher joint probability density. These sequences serve as candidate patches for the repair prediction.

6. Experimental Evaluation

In this study, we intend to answer the following four research questions.

1. RQ1: How does our model perform compared to other repair models?
2. RQ2: Can the dataset generated by the RS help the model with program repair?
3. RQ3: Do the different modules of the model all have a positive impact on the repair effectiveness?
4. RQ4: How do models with different architectures perform on different categories of compiler errors?

6.1. Experimental Datasets

During the program repair example generation phase, two types of datasets were used: SUES-COJ and TEGCER (proposed by Ahmed et al. [16]). The SUES-COJ dataset consists of programs submitted by undergraduate students from Shanghai University of Engineering Science for a C programming course. We adopted the aforementioned strategies to extract reasonable program pairs from execution logs. The dataset comprises a total of 4447 program pairs after the process of mining and filtering. It should be noted that some of the code samples in the dataset contain multiple line modifications. The length of the code ranges from 10 to 30 lines.

The TEGCER dataset is composed of programs submitted online by over 400 undergraduate students from the Indian Institute of Technology during the first semester of the 2015–2016 academic year for an introductory C programming course. The dataset consists of a total of 23,275 code pairs. Each code pair includes the original code statement and the modified code statement, with only one change per code pair. The dataset has undergone preprocessing and filtering operations, restricting the code length to 40 lines or less. Additionally, incorrect repair data have been removed. Ultimately, 21,994 code pairs were retained as the training dataset for the model.

In addition to the above dataset, we also utilized the program perturbation process described in the reference [13] to generate a large number of repair examples as a training set. This process involved random perturbations, including modules for delimiter, type name, keyword, variable, and variable definition. We randomly applied these modules to disrupt the correct code and paired it with the original correct code, forming perturbation-generated code error-repair sample pairs. By comparing this perturbation dataset with the dataset generated through our simulation, we were able to gather valuable insights and evaluate the effectiveness of the RS approach.

During the program repair phase, we also utilized the DeepFix dataset [10]. This dataset was sourced from student-submitted programming task codes and comprises a total of 37,415 compiled programs and 6971 programs that could not be compiled. The dataset contains a significant number of errors spanning multiple lines.

Since the DeepFix dataset does not provide paired repair examples, we used the perturbation-generated programs as the training data and the original erroneous programs as the test set. This approach allowed us to train the model on the perturbed programs and evaluate its performance on the original erroneous programs.

6.2. Error Classification

By analyzing the data from the student error programs, we classified common errors based on the error content and compiler feedback information. The specific categories can be found in Table 2.

Table 2. Program error types and compiler feedback examples.

Error Type	Error Description	Compiler Feedback Example
E1	Variable undefined	'd' undeclared (first use in this function)
E2	Missing delimiter	expected ';' before '}' token
E3	Missing variables or special symbols	expected identifier before '&' token expected '=', ',', ';', 'asm' or '__attribute__' before 'b'
E4	Misuse of expressions	l value required as left operand of assignment expected expression before '=' token
E5	Illegal character error	stray '\346' in program
E6	Misuse of data types	invalid operands to binary & (have 'int *' and 'int')
E7	Other errors	too few arguments to function 'reverse' 'else' without a previous 'if'

6.3. Evaluation Metrics

We utilize three metrics to evaluate the effectiveness of the model.

1. LocalizeAcc: This represents the accuracy of the model in correctly predicting the erroneous line for a given error in the generated dataset. This metric assesses the model's ability to localize the error accurately.
2. SingleAcc: This metric measures the accuracy of the model in predicting the correct fix for a single line repair example in the generated dataset. It evaluates the model's capability to accurately generate the appropriate repair patch.
3. RepairAcc: This metric indicates the success rate of the model in producing fully compilable program fixes, as verified by the compiler, on a real dataset. It assesses the model's overall capability in generating program repairs that pass compilation integrity.

Given an erroneous program set $\mathbf{D}$, with corresponding real error positions i_{err} and the corresponding repair patches $x_{i_{err}}$, along with a validator C that can determine if program p contains errors, we define $d(p)$ as the model's prediction of the erroneous line number for the current program p, $f(p)$ as the model's prediction of the repair patch for the erroneous line in the current program p, and $g(p)$ as the final result after an iterative repair process by the model. The calculation methods for the three evaluation metrics are as follows:

$$\text{LocalizeAcc} = \frac{|\{p|p \in \mathbf{D}, d(p) = i_{err}\}|}{|\mathbf{D}|} \tag{16}$$

$$\text{SingleAcc} = \frac{|\{p|p \in \mathbf{D}, f(p) = x_{i_{err}}\}|}{|\mathbf{D}|} \tag{17}$$

$$\text{RepairAcc} = \frac{|\{p|p \in \mathbf{D}, C(g(p)) = 1\}|}{|\mathbf{D}|} \tag{18}$$

6.4. Training Details

6.4.1. Rookie Simulator

The model architecture of the RS was based on the encoder-decoder Transformer. The model consisted of four layers each for the encoder and decoder. The attention heads for multi-head attention were set to 8, and the vector dimension for the hidden layer states was 256. The feature dimension for the feedforward neural network was set to 1024.

During the training phase, the model parameters were optimized using the Adam optimizer [27]. The batch size for model training was set to 13,500 tokens. The learning rate was set to 0.001 and adjusted using a warm-up strategy, with an initial learning rate of 0.0001. To prevent overfitting, a dropout rate of 0.2 was used. Label smoothing was applied with a smoothing factor of 0.1. Gradient clipping [28] was set to 1.0 to limit the magnitude of gradients. The model was trained for 20 epochs.

During the predicting phase, we used beam search for generating predicted erroneous code statements. The beam size was set to 10, meaning the model predicts and outputs the top 10 sequences with the highest joint probabilities for the given input data.

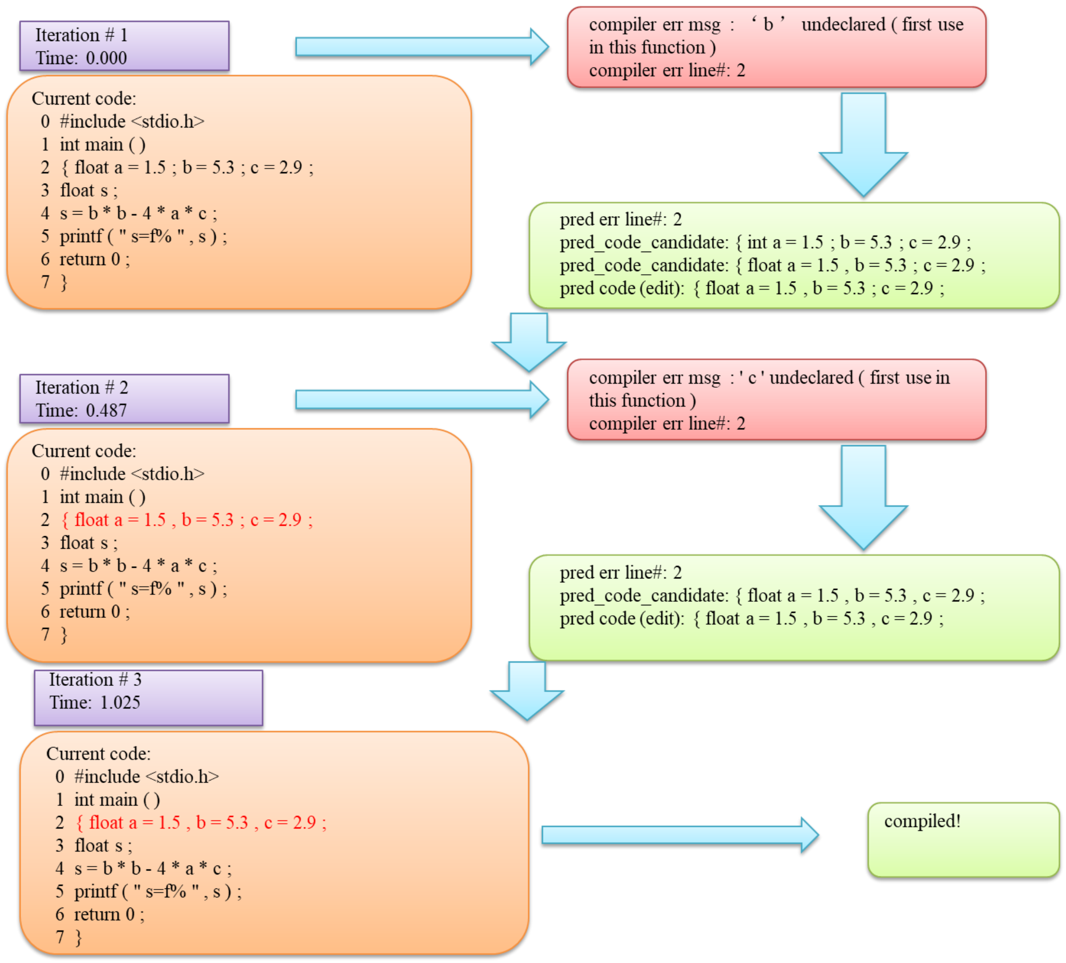
6.4.2. ACNPR Model

The parameter configuration for the ACNPR model is as follows: the dimensions of the input vector and the intermediate local context in the model were set to 200. The number of layers in BiLSTM$^{(1)}$ was set to three, while the graph attention layer was set to two. BiLSTM$^{(2)}$ was set to one layer, and BiLSTM$^{(3)}$ was set to two layers. The number of capsules in the capsule network was set to three, with a dynamic routing count of three.

During the training phase, a dropout rate of 0.3 was set. The Adam optimization method was used for parameter optimization, with a gradient clipping value of 1.0. The batch size was set to 30, and the learning rate was set to 0.00005. In the predicting phase, a beam size of 20 was used for the beam search.

For programs containing multiple lines of errors, we employed an iterative repair strategy where we took the first error message provided by the compiler as input. The GCC compiler was utilized to assess the acceptability of patches. If the patched program resulted in fewer compiler errors, the repair patch was accepted. The verification process continued iteratively, with a threshold of five iterations. Once the maximum number of iterations was reached or the patch passed the compiler verification, the repair process stopped.

The process of program iteration repair is illustrated in Figure 5. Here, "compiler err msg" represents the description of the first error provided by the compiler, and "compiler err line#" indicates the line number where the first error occurs according to the compiler's suggestion. "pred err line#" denotes the line number predicted by the model as the error location and the position to be modified for patching. It is important to note that "pred err line#" may differ from "compiler err line#", as the suggested error location by the compiler is often distinct from the actual correct modification position. "pred_code_candidate" refers to the predicted candidate patch suggested by the model, while "pred code (edit)" signifies the repair patch that has been verified and accepted through the compiler. As observed, multiple errors were detected in code line #2. In each iteration, the first feedback provided by the compiler is used as input to the model for prediction. The predicted patch is then validated by the compiler, and the final decision to apply the repair patch is made before proceeding to the next iteration.



```
Iteration # 1
Time: 0.000

Current code:
 0  #include <stdio.h>
 1  int main ( )
 2  { float a = 1.5 ; b = 5.3 ; c = 2.9 ;
 3  float s ;
 4  s = b * b - 4 * a * c ;
 5  printf ( " s=f% " , s ) ;
 6  return 0 ;
 7  }
```

```
compiler err msg :  ' b '  undeclared ( first use in this function )
compiler err line#: 2
```

```
pred err line#: 2
pred_code_candidate: { int a = 1.5 ; b = 5.3 ; c = 2.9 ;
pred_code_candidate: { float a = 1.5 , b = 5.3 ; c = 2.9 ;
pred code (edit): { float a = 1.5 , b = 5.3 ; c = 2.9 ;
```

```
Iteration # 2
Time: 0.487

Current code:
 0  #include <stdio.h>
 1  int main ( )
 2  { float a = 1.5 , b = 5.3 ; c = 2.9 ;
 3  float s ;
 4  s = b * b - 4 * a * c ;
 5  printf ( " s=f% " , s ) ;
 6  return 0 ;
 7  }
```

```
compiler err msg : ' c ' undeclared ( first use in this function )
compiler err line#: 2
```

```
pred err line#: 2
pred_code_candidate: { float a = 1.5 , b = 5.3 , c = 2.9 ;
pred code (edit): { float a = 1.5 , b = 5.3 , c = 2.9 ;
```

```
Iteration # 3
Time: 1.025

Current code:
 0  #include <stdio.h>
 1  int main ( )
 2  { float a = 1.5 , b = 5.3 , c = 2.9 ;
 3  float s ;
 4  s = b * b - 4 * a * c ;
 5  printf ( " s=f% " , s ) ;
 6  return 0 ;
 7  }
```

```
compiled!
```

Figure 5. Process of program iteration repair. The modifications made after each round of iteration are highlighted in red font.

6.5. Results Analysis

6.5.1. RQ1: Repair Performance Comparison

The performance comparison results of different repair models on the three datasets are shown in Table 3. Our method demonstrated significant improvements compared to models such as DeepFix and SampleFix, which do not have compiler feedback information. Additionally, on the raw dataset, our RepairAcc increased by 4.8%, 4.9%, and 2.3%, respectively, compared to the previously best-performing DrRepair model. It is worth noting that the DeepFix dataset consists of more complex programs with multiple lines of errors, which is why the complete repair rate is relatively lower compared to the other datasets. In the validation set of perturbation-generated data, our model outperformed the DrRepair model in terms of both LocalizeAcc and SingleAcc. Furthermore, it exhibited even greater improvement on the DeepFix dataset, indicating that our model was capable of better capturing the intricacies of program semantics.

Table 3. Performances of different models.

Dataset	Model	LocalizeAcc	SingleAcc	RepairAcc
TEGCER	DeepFix	-	-	27.5%
	DrRepair	97.7%	78.5%	70.2%
	ACNPR	**98.9%**	**83.2%**	**75.0%**
SUES-COJ	DeepFix	-	-	29.1%
	DrRepair	97.9%	79.6%	72.0%
	ACNPR	**99.5%**	**84.5%**	**76.9%**
DeepFix	DeepFix	-	-	27.0%
	SampleFix	-	-	45.3%
	DrRepair	97.9%	74.8%	66.0%
	ACNPR	**99.2%**	**84.6%**	**68.3%**

6.5.2. RQ2: Performance of the RS

We used the RS separately on the TEGCER and SUES-COJ datasets to generate repair examples, resulting in the RS-generated dataset. We then conducted a statistical analysis to determine the quantities and proportions of different error types in both the raw and generated data. The results are presented in Tables 4 and 5.

For common high-frequency error types—E1, E2, and E3—the RS can generate a large number of corresponding erroneous code samples while maintaining the high frequency of occurrence of these errors. Errors E4 and E6 are commonly observed in certain programming tasks but have a generally low overall frequency. Due to their complexity, it is challenging to extract semantic features for these types of errors, making it difficult to accurately simulate and reproduce them. However, the RS is still capable of generating a small number of code samples for these errors. Error type E5 is more commonly observed in some environments but occurs very rarely in others. The RS is able to accurately capture the error patterns of programmers in different environments, allowing these errors to match their corresponding occurrence frequencies across different environments.

Table 4. Chart of error types of SUES-COJ dataset.

Error Type	Raw Dataset		RS-Generated Dataset	
	Quantity	Proportion	Quantity	Proportion
E1	781	17.6%	12,515	17.5%
E2	901	20.3%	24,679	34.6%
E3	1039	23.4%	16,051	22.5%
E4	451	10.1%	3008	4.2%
E5	586	13.2%	10,218	14.3%
E6	160	3.6%	1608	2.3%
E7	529	11.9%	3284	4.6%

Table 5. Chart of error types of TEGCER dataset.

Error Type	Raw Dataset		RS-Generated Dataset	
	Quantity	Proportion	Quantity	Proportion
E1	4670	16.8%	6827	11.5%
E2	6882	24.8%	21,395	36.0%
E3	9720	35.0%	15,186	25.6%
E4	519	1.9%	2915	5.0%
E5	74	0.3%	54	0.1%
E6	591	2.1%	2490	4.2%
E7	5346	19.2%	10,512	17.8%

Overall, the RS can effectively simulate various real error types by generating both extremely common errors and rare errors. Unlike perturbation-generated error programs, the programs generated by the RS are not limited by error patterns. It can generate corresponding error types based on different datasets, and the simulated error frequencies for different error types are moderately aligned with the actual frequencies in the given dataset. The quality of the augmented dataset generated using this method is closer to that of the raw dataset. When the raw dataset contains a wider range of error types, the RS-generated error programs also encompass a broader range of information. Consequently, the repair model trained on the augmented dataset will possess more semantic information from error codes and will be capable of fixing a more diverse set of errors.

The comparative results of different models trained using perturbation-generated and RS-generated datasets are presented in Table 6. The model trained on training data generated using RS simulation performs slightly worse in terms of LocalizeAcc and SingleAcc compared to the model trained on randomly perturbed data. However, it shows better repair rates on real-world data, with an improvement of 1% to 4% in RepairAcc under the same model architecture. The reason behind this improvement is that RS-generated data captures more complex error patterns and is closer to real-world error patterns compared to randomly perturbed data. By using RS-generated data along with the ACNPR model, the optimal repair rate can be achieved.

Table 6. Results of different generated datasets.

Dataset	Model	LocalizeAcc	SingleAcc	RepairAcc
TEGCER	DeepFix	-	-	27.5%
	DeepFix + RS	-	-	31.4%
	DrRepair	97.7%	78.5%	70.2%
	DrRepair + RS	97.2%	76.3%	73.3%
	ACNPR	98.9%	83.2%	75.0%
	ACNPR + RS	98.6%	80.3%	77.9%
SUES-COJ	DeepFix	-	-	29.1%
	DeepFix + RS	-	-	33.3%
	DrRepair	97.9%	79.6%	72.0%
	DrRepair + RS	97.6%	78.0%	75.4%
	ACNPR	99.5%	84.5%	76.9%
	ACNPR + RS	98.1%	80.2%	77.9%

6.5.3. RQ3: Ablation Experiments

We conducted ablation experiments on three datasets to evaluate the effectiveness of each component. The results of the ablation experiments are shown in Table 7. The base model represents a model that retains only the encoding and decoding layers. The graph attention layer helps the model infer the positions of keywords and variables, thereby improving the model’s performance. The capsule network layer consists of two parts: CodeCaps, which extracts deep semantic features from the code, and MsgCaps, which further extracts similar error message features. Both parts contribute to improving the repair effectiveness of the model to a certain extent.

Table 7. Results of ablation experiments.

Dataset	Model	RepairAcc
TEGCER	Base	63.5%
	Base + GA	70.2%
	Base + GA + CodeCaps	73.3%
	Base + GA + CodeCaps + MsgCaps	77.9%
SUES-COJ	Base	62.8%
	Base + GA	70.4%
	Base + GA + CodeCaps	75.5%
	Base + GA + CodeCaps + MsgCaps	77.9%
DeepFix	Base	58.5%
	Base + GA	63.7%
	Base + GA + CodeCaps	67.6%
	Base + GA + CodeCaps + MsgCaps	68.3%

The experimental results demonstrate that the GA module contributes to an improvement in RepairAcc of 6.7%, 7.6%, and 5.2% for the three datasets, respectively. The capsule network layer contributes to an improvement in RepairAcc of 7.7%, 7.5%, and 4.6% for the same three datasets. In particular, for the TEGCER dataset, the capsule network layer exhibits a larger enhancement in repair accuracy, with the MsgCaps module contributing more than the CodeCaps module. Conversely, for the other datasets, the contributions are reversed. This discrepancy may arise from the higher diversity of errors in the TEGCER dataset, as well as the more complex nature of feedback information provided by the compiler. Hence, the MsgCaps module is better suited to extract deep features from the compiler's descriptive information in this particular case.

6.5.4. RQ4: Repair Preferences

To investigate the repair preferences of different modules, we conducted repair validation on the SUES-COJ dataset. The experimental results are shown in Table 8.

Table 8. Results of the repair preferences.

Error Type	Model	RepairAcc
E1	Base	58.5%
	Base + GA	76.7%
	Base + GA + CodeCaps	79.5%
	Base + GA + CodeCaps + MsgCaps	82.2%
E2	Base	91.8%
	Base + GA	94.5%
	Base + GA + CodeCaps	94.9%
	Base + GA + CodeCaps + MsgCaps	95.3%
E3	Base	58.5%
	Base + GA	63.7%
	Base + GA + CodeCaps	67.6%
	Base + GA + CodeCaps + MsgCaps	68.3%
E4	Base	70.2%
	Base + GA	75.3%
	Base + GA + CodeCaps	80.5%
	Base + GA + CodeCaps + MsgCaps	85.3%

Table 8. *Cont.*

Error Type	Model	RepairAcc
E5	Base	43.5%
	Base + GA	43.5%
	Base + GA + CodeCaps	47.8%
	Base + GA + CodeCaps + MsgCaps	52.2%
E6	Base	65.6%
	Base + GA	67.4%
	Base + GA + CodeCaps	75.4%
	Base + GA + CodeCaps + MsgCaps	75.4%
E7	Base	35.0%
	Base + GA	35.0%
	Base + GA + CodeCaps	42.2%
	Base + GA + CodeCaps + MsgCaps	55.6%

The graph attention layer is particularly effective in improving the repair effectiveness for error types that require cross-referencing critical information across code and error description. Error types E1, E2, E3, and E4 contain relevant keyword information in the error description. The graph attention layer aids the model in connecting the key content in the compiler description with the corresponding occurrences in the code, even when they span multiple lines.

The capsule network layer significantly improves the repair effectiveness for error types with complex error patterns and challenging repair mode determination based on the compiler description information. Error types E3, E4, E5, E6, and E7 contain diverse repair patterns. The capsule network layer is adept at extracting deep semantic information and generating higher-order features, enabling the model to make more informed decisions regarding deep-level repair patterns.

7. Discussions and Limitations

With the expansion of programming courses, many educational institutions have begun adopting blended learning models that combine both online and offline methods [29]. The introduction of online intelligent learning systems allows for targeted improvement of students' learning outcomes by providing immediate feedback and assessment [30]. During programming, students can utilize our tool to receive timely and relevant error feedback after submitting erroneous programs, including more precise error locations and suggested repair patches. Additionally, the error examples submitted by students can further contribute to the training of the repair model. Currently, the parallel processing efficiency of our tool needs further enhancement to cope with a large volume of requests.

It is worth noting that the program sample mining method in this paper also relies on manual review to exclude invalid repair examples, aiming for higher-quality program sample pairs. Additionally, the multiline program repair in this study depends on the compiler for verification. However, the reduction of compiler feedback information during the iterative repair process sometimes cannot accurately represent the effectiveness of partial repairs. Further improvement in optimizing the compiler's feedback information could potentially enhance the success rate of program repair.

Our approach has significant limitations in terms of its ability to repair certain types of errors. Firstly, our method is only applicable to single-file programs and does not examine included files. Regarding #include header file declarations, we consider them as a whole input to the model, hence we are unable to repair errors in header file declarations. Secondly, although the specific values of string and numeric constants typically do not impact program syntax, during the abstraction process, some erroneous program contents may be mistakenly identified as constants and anonymized. This, in turn, hampers the effectiveness of repairing such errors.

Due to the increased computational complexity caused by the dynamic routing algorithm and iterative computing process in capsule networks, the training time is prolonged [31]. In order to address specific or novel error types, we need to augment the training data with samples containing these new error types and retrain the model. As a result, our repair model requires larger computational resources when facing larger-scale datasets.

Recently, there has been significant research interest in leveraging large-scale models for code generation tasks [32–34]. While large models have shown promising results, their resource-intensive nature for training and deployment poses challenges. In light of these challenges, future research directions could focus on using large models to assist smaller models and enhance the accuracy of their fine-grained repair capabilities. This approach could involve a multi-model framework where a large model is used to guide and augment the training process of smaller models. By exploiting the strengths of large models, such as their ability to capture complex code semantics, smaller models can benefit from their knowledge and refine their repair functions. This approach may lead to more efficient and effective code repairs with better accuracy.

8. Conclusions

This paper presents a method for mining reasonable error-repair program sample pairs from students' program execution logs in the online experimental system, providing a way to acquire a labeled dataset of real program errors. To address the issue of limited error patterns and unrealistic error contexts in the perturbed generated dataset, this study proposes the RS to generate a large number of program pairs with more semantic information that better align with real error contexts. To tackle low-frequency errors and complex error patterns in program repair, the ACNPR model is introduced, which integrates compiler feedback information. Through comparative experiments conducted on three different datasets, the effectiveness of the repair model is validated.

Our approach utilizes a dataset obtained from students' submissions in programming tasks, which predominantly consists of errors commonly made by students and novice programmers. This allows us to provide timely assistance to students during their programming process by locating specific errors and suggesting appropriate modification strategies. This approach aims to enhance students' learning efficiency while reducing the workload for teachers. However, it is important to note that at present, our method focuses primarily on fixing compilation errors. Therefore, the predicted repair patches cannot guarantee the successful completion of programming tasks. In the future, we plan to incorporate semantic constraints targeting program functionality to generate repair patches that are more likely to be semantically correct. This will effectively guide students in completing programming tasks and foster a more proactive learning experience.

Author Contributions: Conceptualization, R.S. and J.H.; methodology, R.S.; software, R.S. and B.L.; validation, R.S. and B.L.; writing—original draft preparation, R.S.; writing—review and editing, J.H.; project administration, J.H.; funding acquisition, J.H. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Shanghai Collaborative Innovation Center of data intelligence technology, grant number 0232-A1-8900-23-09.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Tracy, C.; Stu, Z.; Ellen, W.; Lecia, B. Booming Enrollments: Good Times? In Proceedings of the 46th ACM Technical Symposium on Computer Science Education, Kansas City, MO, USA, 4–7 March 2015; pp. 80–81. [CrossRef]
2. Jiang, J.; Chen, J.; Xiong, Y. Survey of Automatic Program Repair Techniques. *J. Softw.* **2021**, *32*, 2665–2690. [CrossRef]
3. Weimer, W.; Nguyen, T.V.; Le Goues, C.; Forrest, S. Automatically finding patches using genetic programming. In Proceedings of the 31st International Conference on Software Engineering, Vancouver, BC, Canada, 16–24 May 2009; pp. 364–374. [CrossRef]
4. Qi, Y.; Mao, X.; Lei, Y.; Dai, Z.; Wang, C. The strength of random search on automated program repair. In Proceedings of the 36th International Conference on Software Engineering, New York, NY, USA, 31 May–7 June 2014; pp. 254–265. [CrossRef]
5. Oliveira, V.P.L.; Souza, E.F.D.; Goues, C.L.; Camilo-Junior, C.G. Improved representation and genetic operators for linear genetic programming for automated program repair. *Empir. Softw. Eng.* **2018**, *23*, 2980–3006. [CrossRef]
6. Nguyen, H.D.T.; Qi, D.; Roychoudhury, A.; Chandra, S. SemFix: Program repair via semantic analysis. In Proceedings of the 35th International Conference on Software Engineering, San Francisco, CA, USA, 18–26 May 2013; p. 7722212781. [CrossRef]
7. Mechtaev, S.; Yi, J.; Roychoudhury, A. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In Proceedings of the 38th International Conference on Software Engineering, New York, NY, USA, 14–22 May 2016; pp. 691–701. [CrossRef]
8. Wang, H.; Liu, H.; Li, Z.; Liu, Y.; Sun, F.; Chen, X. A Token-based Compilation Error Categorization and Its Applications. *J. Softw. Evol. Proc.* **2023**, *35*, e2512. [CrossRef]
9. Manish, M.; Sandhya, S.; René, J.; Yuriy, B. Do automated program repair techniques repair hard and important bugs? In Proceedings of the 40th International Conference on Software Engineering, New York, NY, USA, 27 May–3 June 2018; pp. 2901–2947. [CrossRef]
10. Gupta, R.; Pal, S.; Kanade, A.; Shevade, S. Deepfix: Fixing common C language errors by deep learning. In Proceedings of the 31st AAAI Conference on Artificial Intelligence, San Francisco, CA, USA, 4–9 February 2017; pp. 1345–1351. [CrossRef]
11. Gupta, R.; Kanade, A.; Shevade, S. Deep reinforcement learning for syntactic error repair in student programs. In Proceedings of the 33rd AAAI Conference on Artificial Intelligence, Honolulu, HI, USA, 27 January–1 February 2019; pp. 930–937. [CrossRef]
12. Hajipour, H.; Bhattacharyya, A.; Staicu, C.A.; Fritz, M. SampleFix: Learning to Generate Functionally Diverse Fixes. In Proceedings of the Joint European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases. ECML PKDD 2021, Bilbao, Spain, 13–17 September 2021; pp. 119–133. [CrossRef]
13. Yasunaga, M.; Liang, P. Graph-based, self-supervised program repair from diagnostic feedback. In Proceedings of the 37th International Conference on Machine Learning, Vienna, Austria, 13–18 July 2020; pp. 10799–10808.
14. Mesbah, A.; Rice, A.; Johnston, E.; Glorioso, N.; Aftandilian, E. Deepdelta: Learning to repair compilation errors. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, New York, NY, USA, 26–30 August 2019; pp. 925–936. [CrossRef]
15. Seo, H.T.; Han, Y.S.; Ko, S.K. MultiFix: Learning to Repair Multiple Errors by Optimal Alignment Learning. In Proceedings of the Association for Computational Linguistics: EMNLP 2021, Punta Cana, Dominican Republic, 7–11 November 2021; pp. 4850–4855. [CrossRef]
16. Ahmed, U.Z.; Sindhgatta, R.; Srivastava, N.; Karkare, A. Targeted Example Generation for Compilation Errors. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019; pp. 327–338. [CrossRef]
17. Chhatbar, D.; Ahmed, U.Z.; Kar, P. MACER: A Modular Framework for Accelerated Compilation Error Repair. In Proceedings of the 21st International Conference on Artificial Intelligence in Education, Ifrane, Morocco, 6–10 July 2020; pp. 106–117. [CrossRef]
18. Sabour, S.; Frosst, N.; Hinton, G.E. Dynamic Routing Between Capsules. In Proceedings of the 2017 Advances in Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA, 4–9 December 2017; pp. 3880–3890.
19. Zhao, W.; Ye, J.; Yang, M.; Lei, Z.; Zhang, S.; Zhao, Z. Investigating Capsule Networks with Dynamic Routing for Text Classification. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, 31 October–4 November; pp. 3110–3119.
20. Jia, X.; Wang, L. Attention enhanced capsule network for text classification by encoding syntactic dependency trees with graph convolutional neural network. *PeerJ Comput. Sci.* **2022**, *8*, e831. [CrossRef] [PubMed]
21. Yujian, L.; Bo, L. A normalized Levenshtein distance metric. *IEEE Trans. Pattern Anal.* **2007**, *29*, 1091–1095. [CrossRef] [PubMed]
22. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, L.; Polosukhin, I. Attention is all you need. In Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA, 4–9 December 2017; pp. 6000–6010.
23. Michele, T.; Cody, W.; Gabriele, B.; Massimiliano, D.P.; Martin, W.; Denys, P. An Empirical Investigation into Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. In Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 832–837. [CrossRef]
24. Velickovic, P.; Cucurull, G.; Casanova, A.; Romero, A.; Lio, P.; Bengio, Y. Graph attention networks. *OpenReview.Net* **2018**. [CrossRef]
25. Lei Ba, J.; Kiros, J.R.; Hinton, G.E. Layer Normalization. *arXiv* **2016**, arXiv:1607.06450. [CrossRef]
26. See, A.; Liu, P.J.; Manning, C.D. Get to the point: Summarization with pointer-generator networks. In Proceedings of the 55th Annual Meering of the Association for Computational Lunguistics, Vancouver, BC, Canada, 30 July–4 August 2017; pp. 1073–1083. [CrossRef]

27. Kingma, D.P.; Ba, J. Adam: A method for stochastic optimization. In Proceedings of the 3rd International Conference on Learning Representations, San Diego, CA, USA, 7–9 May 2015. [CrossRef]
28. Pascanu, R.; Mikolov, T.; Bengio, Y. On the difficulty of training recurrent neural networks. In Proceedings of the 30th International Conference on Machine Learning, Atlanta, GA, USA, 16–21 June 2013; pp. 1310–1318.
29. Natasa, H.B.; Vedran, M.; Ivica, B. A blended learning approach to course design and implementation. *IEEE Trans. Educ.* **2009**, *52*, 19–30. [CrossRef]
30. Alammary, A.; Sheard, J.; Carbone, A. Blended learning in higher education: Three different design approaches. *Australas. J. Educ. Technol.* **2014**, *30*, 440–454. [CrossRef]
31. Kosiorek, A.; Sabour, S.; Teh, Y.W.; Hinton, G.E. Stacked Capsule Autoencoders. In Proceedings of the 33rd Conference on Neural Information Processing Systems, Vancouver, BC, Canada, 8–14 December 2019; pp. 15512–15522.
32. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. CodeBERT: A pre-trained model for programming and natural languages. *arXiv* **2020**, arXiv:2002.08155.
33. Jiang, N.; Lutellier, T.; Tan, L. Cure: Code-aware neural machine translation for automatic program repair. In Proceedings of the 43rd International Conference on Software Engineering (ICSE), Madrid, Spain, 25–28 May 2021; pp. 1161–1173. [CrossRef]
34. Berabi, B.; He, J.; Raychev, V.; Vechev, M. TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer. In Proceedings of the 38th International Conference on Machine Learning, Virtual, 18–24 July 2021; pp. 780–791.