Co-REX:A Collaborative Transformer for Try Location Detection and Catch Exception Prediction

Jing Xiao
School of Computer Science
South China Normal University
Guangzhou, China
xiaojing@scnu.edu.cn

Tianyu Liang
School of Artificial Intelligence
South China Normal University
Foshan, China
2022024925@m.scnu.edu.cn

***Abstract*—Try Location Detection (TLD) and Catch Exception Prediction (CEP) are two interrelated tasks in runtime exception handling which is a crucial aspect in Java programming. The performance of each task can significantly benefit from the information provided by the other. However, previous methods primarily treated these tasks in isolation or addressed them separately using multi-task frameworks, which overlooked their interconnected information flow. In this work, we propose a Collaborative transformer model for Runtime EXception detector (Co-REX), which establishes a bidirectional information flow between TLD and CEP and enables mutual enhancement through joint learning. Additionally, to evaluate the performance of Co-REX, we collect a large number of Java methods from Gitee and GitHub to create a benchmark dataset. Our experimental results demonstrate that Co-REX achieves 78.1% accuracy outperforming other state-of-the-art methods by a margin of at least 2.1% for TLD. It also performs better by 3.1% improvement in terms of Top-1 accuracy than others for CEP.**

***Index Terms*—Try Location Detection, Catch Exception Prediction, Runtime Exception, Collaborative Transformer**

I. Introduction

Exception handling mechanism plays a pivotal role in modern programming languages, serving as a critical technology for ensuring the robustness and reliability of software systems [1]. For instance, the Java language employs try-catch blocks, allowing developers to efficiently identify and manage runtime exceptions, thereby facilitating exception recovery mechanisms. This approach segregates the code dealing with errors from the regular source code, significantly enhancing the program's comprehensibility and maintainability in the later stages. However, crafting high-quality exception handling code is a time-consuming endeavor. Consequently, the handling of runtime exceptions automatically through deep learning have garnered increasing attention in recent years.

In runtime exception handling, there are two critical tasks, try location detection (TLD) and catch exception prediction (CEP). Recent methods explore novel solutions based on deep learning for TLD and CEP. NexGen [13] utilizes a bidirectional long short term memory network (BiLSTM) and attention mechanism to detect potential runtime exceptions in Java code and generate appropriate handling code. [14] aims to predict the location and type of potential runtime exceptions by identifying correlations between certain code elements and Java runtime exception types. [17] develops a graph-based code representation that combines control flow graphs, data flow graphs and abstract syntax trees into a unified graph to capture both semantic dependencies and syntactic structures. Although these methods demonstrate improved experimental results for both TLD and CEP, they only focus on the information flow of each task independently, lacking an exploration of the interconnected information flow between these two tasks. For example, in Figure 1, the invocation of $parseInt()$ on the method parameter value gives a clue about $NumberFormatException$. With the mutual information about TLD locating in statement 3 with $parseInt()$, we tend to predict the exception type $NumberFormatException$ in CEP. On the contrary, when $NumberFormatException$ occupies a greater weight in CEP, we prefer to locate the statement which contains $parseInt$ or others related to $NumberFormatException$.

```
1 public static String [] TransMax(String [] params, String max){
2     for (int i = 0; i < params.length - 1; i++) {
3         if (Integer.parseInt(params[i]) > Integer.parseInt(max)) {
4             params[i] = max;
5         }
6     }
7     return params;
8 }
```

Fig. 1. An example of Java method prone to runtime exception.

To address the previous lack of mutual information exploration between TLD and CEP, we introduce Co-REX, a novel framework for jointly dealing with these two tasks. The key component in Co-REX is the collaborative transformer, which effectively integrates bidirectional information flow between the two tasks and enhance their performance in a mutual way. Specifically, in the collaborative transformer, we first employ a label attention mechanism over the labels of TLD and CEP to generate the explicit input representations, which focus on the exception location and exception type semantic information respectively. Secondly, the explicit exception location and type representations are fed into the collaborative transformer to make mutual interaction. In particular, the exception location

DOI reference number: 10.18293/SEKE2024-028

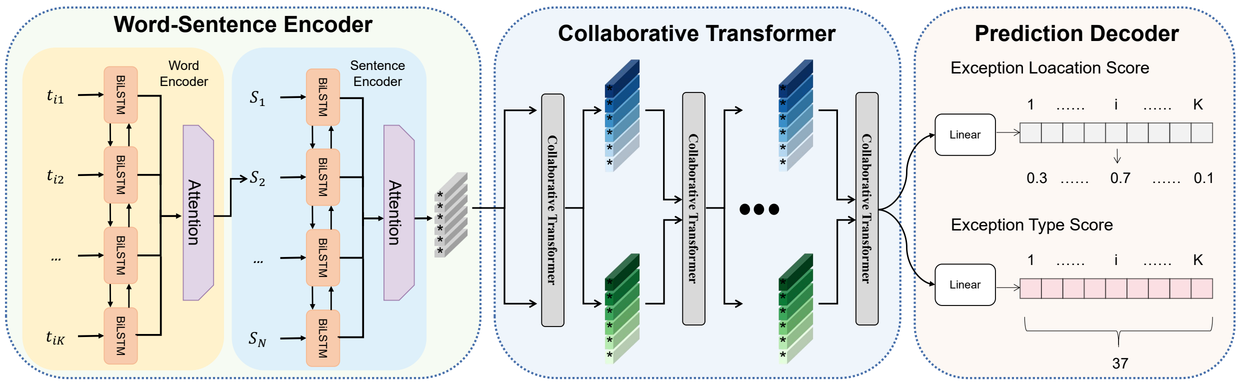

Fig. 2. The overall architecture of the proposed Co-REX model.

representations are treated as queries and exception type representations are considered as keys and values to obtain the type-aware location representation. Meanwhile, the exception type representations are used as queries and location representations are treated as keys and values to get the location-aware type representations. These above operations can establish the bidirectional connection across TLD and CEP and allow them to benefit from enhanced mutual information. Moreover, to evaluate the proposed Co-REX model, we have compiled an extensive dataset, containing more than 500K Java methods, by gathering a vast array of Java projects from Gitee and GitHub, followed by rigorous data cleansing efforts. The main contributions of this paper are as follows:

- We introduce Co-REX, a novel collaborative transformer model that enhances predictive accuracy and robustness by enabling dynamic bidirectional information sharing between TLD and CEP.
- We create a benchmark dataset with more than 500K Java methods for runtime exception handling and the dataset is available at https://github.com/lyc-mz/Co-REX.
- Extensive experimental results demonstrate that Co-REX outperforms state-of-the-art approaches in terms of predictive performance.

II. Related Work

In the field of software engineering, significant progress has been made in the study of exception handling [1]. Recent research [3], [4] has revealed a plethora of undocumented runtime exceptions within Java libraries and highlighted the challenges that developers may face in handling exceptions. This underscores the necessity for automated techniques to aid developers in identifying code that is prone to runtime exceptions.

Static error detection technologies are continually evolving. Traditional pattern-based techniques [5], [6], while effective, require extensive manual effort to adapt to updates in code and emerging error types. In contrast, studies [7], [8] have automated the mining of API usage rules from extensive source code, identifying violations as potential API misuse defects, yet they lack specificity for high-frequency APIs. Techniques proposed by [9] that use static and dynamic analysis to detect uncaught exceptions are effective but may face scalability issues in large-scale Java projects.

In exploration on automated exception handling techniques, [10], [11] introduced heuristic search methods based on code context, while [12] focused on recommending exception handling code through GitHub code searches. However, these innovative approaches overlooked other significant factors outside the code context, such as the developers' experience and specific project requirements. NexGen [13] advanced the prediction of try block locations and the generation of exception handling code, while D-REX [14] offered token-level, finer-grained predictions that target hard-to-capture runtime exceptions, providing highly interpretable suggestions to developers. Despite their effectiveness, there is a risk these methods might overemphasize a global understanding of the code at the expense of pinpointing critical exception locations. Similarly, methods like [15] and [16], although they are capable of leveraging program contexts to provide relevant suggestions, they are not sufficient for handling more complex or unusual exception scenarios. In terms of capturing code semantics, [17] represented code as control flow graphs (CFGs) and data flow graphs (DFGs). [18], [19] explored the complementarity of different code representations such as tokens, abstract syntax tree (AST) and CFGs through empirical studies and multimodal attention mechanisms. These studies indicate that diverse representational methods can provide a more comprehensive perspective, aiding in a more precise understanding and handling of code.

III. Methodology

In this section, we first introduce the two key tasks in this paper. Then the Co-REX framework is presented in detail. As depicted in Figure 2, the overall model architecture of Co-REX consists of three parts: the word-sentence encoder, the collaborative transformer and the prediction decoder.

A. Problem Formulation

TLD and CEP are two key tasks in runtime exception handling. TLD is determining which statements may throw

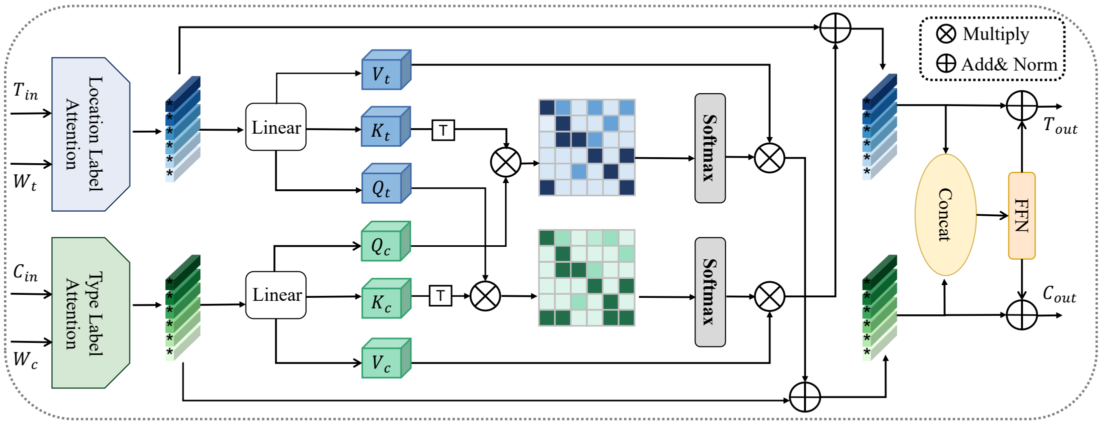

Fig. 3. The Collaborative Transformer.

exceptions and should be enclosed by a try block, while CEP seeks to predict the type of exceptions.

Given the code fragments $C = \{s_1, s_2, \ldots, s_N\}$, where N is the number of statements. TLD is to determine one sequence $\mathbb{Y} = \{y_1, y_2, \ldots, y_N\}$, where $y_i = 0$ represents that statement s_i has no exception and $y_i = 1$ means that statement s_i exists exception. CEP aims to identify a set of exception types $\mathbb{P} = \{p_1, p_2, \ldots, p_M\}$, where M is the number of exception types and p_i means the probability of occurring exception type i. If all statements are labelled by 0, it implies that the code fragment will not generate exceptions at runtime. In this study, nested try-catch blocks are not considered.

B. Word-Sentence Encoder

The initial stage in our model involves the encoding of code fragments to generate context-sensitive embeddings for each statement. Given a code fragment $C = \{s_1, s_2, \ldots, s_N\}$, where each statement s_i is composed of a sequence of tokens $T_i = \{t_{i1}, t_{i2}, \ldots, t_{iK}\}$ and each token t_{ij} in the sequence is first initialized into a corresponding vector $\boldsymbol{x}_{ij} \in \mathbb{R}^d$. To encode the token vectors into a statement-level representation, we employ a BiLSTM network, which can capture dependencies across the entire sequence by processing the sequence in both forward and reverse directions. The output of BiLSTM for each token t_{ij} is a hidden state $\boldsymbol{h}_{ij} \in \mathbb{R}^d$. Then, we employ a attention mechanism over the hidden states to filter out irrelevant tokens. The statement embedding $\boldsymbol{S}_i \in \mathbb{R}^d$ can be calculated as follow:

$$\boldsymbol{S}_i = \sum_{j=1}^{N} \alpha_{ij} \boldsymbol{h}_{ij}, \tag{1}$$

$$\alpha_{ij} = \frac{exp(\boldsymbol{u}_{ij}^T \boldsymbol{u}_w)}{\sum_{j=1}^{N} exp(\boldsymbol{u}_{ij}^T \boldsymbol{u}_w)}, \tag{2}$$

$$\boldsymbol{u}_{ij} = tanh(\boldsymbol{W}_w \boldsymbol{h}_{ij} + \boldsymbol{b}_w), \tag{3}$$

where α_{ij} is the attention coefficient, $\boldsymbol{W}_w \in \mathbb{R}^{d \times d}$, $\boldsymbol{u}_w \in \mathbb{R}^d$ and $\boldsymbol{b}_w \in \mathbb{R}^d$.

After obtaining a sequence of statement vectors, we analyze their sequential dependencies by processing them through another BiLSTM to obtain the hidden states $\boldsymbol{h}'_i = \text{BiLSTM}(\boldsymbol{S}_i)$. We then apply an attention mechanism, assigning weights to these statements to derive the updated embeddings $\boldsymbol{S}'_i \in \mathbb{R}^d$ so that:

$$\boldsymbol{S}'_i = \beta_i \boldsymbol{h}'_i, \tag{4}$$

$$\beta_i = \frac{exp(\boldsymbol{u}_i^T \boldsymbol{u}_s)}{\sum_{j=1}^{K} exp(\boldsymbol{u}_i^T \boldsymbol{u}_s)}, \tag{5}$$

$$\boldsymbol{u}_i = tanh(\boldsymbol{W}_s \boldsymbol{h}'_i + \boldsymbol{b}_s). \tag{6}$$

Then, we can generate the code fragment matrix $\boldsymbol{F} \in \mathbb{R}^{N \times d}$ by concatenating the updated statement embeddings.

C. Collaborative Transformer

The collaborative transformer is a central component of the Co-REX framework, which can make the representation in TLD updated with the guidance of associated CEP task and the representation in CEP updated with the guidance derived from TLD, achieving a bidirectional connection with the two tasks. The detailed architecture of the collaborative transformer is shown in the Figure 3.

To get the explicit exception location and exception type representations for TLD and CEP, we perform label attention over TLD and CEP label. We first treat the code fragment matrix $\boldsymbol{F}$ as the input data $\boldsymbol{T}_{in}$ for TLD and $\boldsymbol{C}_{in}$ for CEP. Secondly, we regard the parameters of the fully-connected TLD decoder layer and CEP decoder layer as exception location embedding matrix $\boldsymbol{W}_t \in \mathbb{R}^{d \times T^{label}}$ and exception type embedding matrix $\boldsymbol{W}_c \in \mathbb{R}^{d \times C^{label}}$ (T^{label} and C^{label} represent the number of TLD and CEP label, respectively), which represent the distribution of labels in a certain sense.

For TLD, we use $\boldsymbol{T}_{in}$ as the query, $\boldsymbol{W}_t$ as the key and value to obtain the explicit exception location representations $\boldsymbol{T}_{label}$ with TLD label attention:

$$\boldsymbol{T}_{label} = \boldsymbol{T}_{in} + softmax(\boldsymbol{T}_{in} \boldsymbol{W}_t) \boldsymbol{W}_t^T, \tag{7}$$

where $\boldsymbol{T}_{label} \in \mathbb{R}^{N\times d}$. Similarly, we treat $\boldsymbol{C}_{in}$ as the query, $\boldsymbol{W}_c$ as the key and value to get the exception type representations $\boldsymbol{C}_{label} \in \mathbb{R}^{N\times d}$, which capture exception type semantic information.

Then, we design different linear projections to map the matrix $\boldsymbol{T}_{label}$ and $\boldsymbol{C}_{label}$ to queries $(\boldsymbol{Q}_t, \boldsymbol{Q}_c)$, keys $(\boldsymbol{K}_t, \boldsymbol{K}_c)$ and values $(\boldsymbol{V}_t, \boldsymbol{V}_c)$. To obtain the exception location representations to incorporate the corresponding exception type information, we utilize the typical self-attention mechanism to model the input data. Specifically, we treat $\boldsymbol{Q}_t$ as queries, $\boldsymbol{K}_c$ as keys, $\boldsymbol{V}_c$ as values and employ layer normalization function to generate the type-aware location representation $\boldsymbol{T}' \in \mathbb{R}^{N\times d}$. The process can be expressed as the following equations:

$$\boldsymbol{T}' = LayerNorm(\boldsymbol{T}_{label} + softmax(\frac{\boldsymbol{Q}_t\boldsymbol{K}_c^T}{\sqrt{d_k}})\boldsymbol{V}_c). \quad (8)$$

Similarly, we treat $\boldsymbol{Q}_t$ as queries, $\boldsymbol{K}_t$ as keys and $\boldsymbol{V}_t$ as values to obtain the location-aware type representation $\boldsymbol{C}' \in \mathbb{R}^{N\times d}$.

Furthermore, we apply a feed-forward network (FFN) to integrate the information flows of exception location and exception type. We first concatenate $\boldsymbol{T}'$ and $\boldsymbol{C}'$ to get $\boldsymbol{H} \in \mathbb{R}^{N\times 2d}$ and then feed it into FFN as follow:

$$\boldsymbol{T}_{out} = LayerNorm(\boldsymbol{T}' + FFN(\boldsymbol{H})), \quad (9)$$

$$\boldsymbol{C}_{out} = LayerNorm(\boldsymbol{C}' + FFN(\boldsymbol{H})), \quad (10)$$

$$FFN(\boldsymbol{H}) = max(0, \boldsymbol{H}\boldsymbol{W}_1 + \boldsymbol{b}_1)\boldsymbol{W}_2 + \boldsymbol{b}_2, \quad (11)$$

where $\boldsymbol{T}_{out} \in \mathbb{R}^{N\times d}$ and $\boldsymbol{C}_{out} \in \mathbb{R}^{N\times d}$ are the obtained representations of exception location and exception type, respectively.

D. Prediction Decoder

In order to conduct sufficient interaction between the two tasks, we apply a stacked collaborative transformer with multiple layers. After stacking L layer, we obtain the final exception location representations $\boldsymbol{T}_{final} \in \mathbb{R}^{N\times d}$ and final exception type representations $\boldsymbol{C}_{final} \in \mathbb{R}^{N\times d}$.

For TLD decoder layer, we predict the result of whether each statement of code fragment has an exception as follow:

$$\boldsymbol{S}_t = \sigma(\boldsymbol{T}_{final}\boldsymbol{W}_t), \quad (12)$$

where $\boldsymbol{S}_t \in \mathbb{R}^{N\times T^{label}}$ is the score vector of label in TLD and $\boldsymbol{W}_t \in \mathbb{R}^{d\times T^{label}}$ is the training parameters. Generally, σ is the sigmoid function, $T^{label} = 1$ and $\boldsymbol{S}_t(i) > 0.5$ indicates statement i has a runtime exception while $\boldsymbol{S}_t(i) <= 0.5$ means there is no exception.

For CEP decoder layer, we use the following formula to obtain the exception score $\boldsymbol{S}_c \in \mathbb{R}^{N\times T^{label}}$:

$$\boldsymbol{S}_c = \sigma(\boldsymbol{C}_{final}\boldsymbol{W}_c), \quad (13)$$

where σ is the ReLU function, $\boldsymbol{W}_c \in \mathbb{R}^{d\times C^{label}}$ is the training parameters and exception label $\hat{y}$ can be calculated by $\hat{y} = argmax(\boldsymbol{S}_c(i))$.

IV. Experimental Results

This section first details the datasets, baselines and related experiments. Then, the results of related experiments are reported to justify our superiority. Finally, we discuss our observations from the results.

A. Data Collection

To evaluate the performance of Co-REX, we create a runtime exception location detection and type prediction dataset. Firstly, we develop automated scripts to clone Java projects from Gitee and GitHub. In particular, we target projects publised after 2010 with more than 50 star ratings to make sure that we include only recent and well-regarded code. Then we batch process these projects to extract Java methods that were suitable for analysis by using the JavaProjectBuilder tool. Additionally, we focus on methods that contained only a single try block and whose length ranged between 3 to 50 lines, to maintain consistency and relevance to typical exception handling scenarios.

From this initial extraction, we conduct a statistical analysis to refine our dataset further. We select 291,008 Java methods containing runtime exceptions and collect 227,240 no exception Java methods that match the exception methods in frequency and quantity to balance the dataset. The final benchmark dataset comprises 518,248 Java methods, providing a robust foundation for evaluating our model's performance across a wide range of exception handling scenarios and enhancing the reliability of our experimental results.

B. Baselines

To evaluate the effectiveness of Co-REX, we use the following models as baselines:

- Conditional Random Field (CRF) is a sequence modeling framework used to predict program properties. It takes a raw token sequence as input and maximizes the joint probability of the entire label sequence for a given observation.
- BiLSTM [2] is effective for modeling sequence data and access both past and future input features at a given position to enhance performance.
- NexGen [12] employs BiLSTM and attention mechanisms to generate token and statement embeddings, using a binary classifier to predict if a statement throws an exception.
- D-REX [13] accurately predicts exception types and provides interpretable suggestions by tokenizing the sequence and using a position-aware transformer model.

C. Implementation Details

Due to the different characteristics of TLD and CEP, we evaluate the performance of Co-REX on the two tasks by different approaches.

In TLD experiments, we utilize a dataset comprising 253,928 Java methods associated with the top 53 types of exceptions and 272,688 no exception Java methods. Each type of exception uses for experiments occurring more than

350 times, which ensures a diverse and representative sample of common exception handling patterns. To evaluate our model, we employ the Accuracy, Precision, Recall and F1 Score as performance metrics, which are common metrics in other methods. In CEP experiments, we adopt a rigorous evaluation strategy that considers only the Top-1 accuracy for each method. The core of this Top-1 accuracy is that for a given code segment, the model must predict a runtime exception type with the highest probability. Compared to Top-k accuracy methods, this approach is statistically stricter as it does not consider suboptimal predictions. To validate the exception type prediction capability of Co-REX, we consider to compare the performance of CEP across datasets with different numbers of exception types. Specifically, we filter the data to create datasets containing the top 8 most frequent exception types (8 ET), the top 25 exception types (25 ET) and the top 53 exception types (53 ET), resulting in datasets with 191,022 Java methods, 233,083 Java methods and 251,207 Java methods, respectively.

To ensure the reliability and generalizability of the experimental results, we split the datasets into training, validation and test sets in a ratio of 7:1:2. In each experiment, the Adam optimizer with a learning rate of 0.001 and a momentum of 0.9 is used to minimize the loss value. The embedding dimension d, batch size, dropout rate are set to 128, 64 and 0.1 respectively. All models are implemented using Python and PyTorch on a Linux server.

TABLE I
PERFORMANCE(%) COMPARISON OF ALL METHODS FOR TLD.

Methods	Precision	Recall	F1 score	Accuracy
CRF	65.3	27.6	37.3	54.4
BiLSTM	74.6	69.8	74.1	69.6
NexGen	80.9	74.5	77.4	75.5
D-REX	82.4	74.8	78.2	76.0
Co-REX	**84.7**	**80.3**	**82.0**	**78.1**

D. Performance Analysis

The prediction performance of our model and the other baselines for TLD is shown in Table I. Overall, Co-REX achieves better prediction results than the other baselines, which exhibits its superior ability in TLD. In the following, we present thorough observations and analyses of the experimental results:

In the benchmark dataset, it achieves 84.7% Precision, 80.3% Recall, 78.1% Accuracy and an F1 score of 82%. Additionally, Co-REX shows improvements over NexGen in Accuracy, Precision, Recall and F1 score by 3.8%, 5.8%, 4.6% and 3.6%, respectively. Compared to the current state-of-the-art D-REX, Co-REX performs better by 2.3%, 5.5%, 3.8% and 2.1% in Accuracy, Precision, Recall and F1 score. These results indicate that considering only source code tokens is insufficient to capture code semantics and it is also necessary to pay attention to the exception type information in the method, which helps locate the exceptions. Intuitively, Co-REX greatly improves its performance in locating anomalies by integrating anomaly location information with anomaly type information and collaborating with each other through information exchange.

TABLE II
TOP-1 ACCURACY(%) COMPARISON OF ALL METHODS FOR CEP.

Methods	8 ET	25 ET	53 ET
BiLSTM	63.9	59.3	55.4
NexGen	84.0	81.5	79.3
D-REX	83.2	81.1	78.5
Co-REX	**85.9**	**83.4**	**82.4**

The performance in terms of Top-1 accuracy of Co-REX and the other baselines for CEP is shown in Table II. It is evident that the pure BiLSTM model performs the worst among the methods. It achieves only 63.9%, 59.3% and 55.4% accuracy on the 8 ET, 25 ET and 53 ET dataset respectively. The NexGen reaches an accuracy of 84%, 81.5% and 79.3%. D-REX obtains only 83.2%, 81.1% and 78.5%, indicating that merely using token and method location information is insufficient for accurately predicting exception types. It is noteworthy that Co-REX outperforms NexGen by 1.9%, 1.9% and 3.1% accuracy on the 8 ET, 25 ET and 53 ET dataset. The result means that exception type prediction accuracy can be improved by the guidance of the exception location information.

E. Ablation study

To further understand the contributions of individual components within the Co-REX model, we conduct a series of ablation experiments in the benchmark dataset. Figure III show the result of this ablation experiment.

Impact of Explicit Representations. We remove the exception location label attention layer and replace $\boldsymbol{T}_{label}$ with $\boldsymbol{T}_{in}$. This means that we only get the exception type representation explicitly, without the exception location semantic information. We name it as *w/o location label attention*. Similarly, we perform the *w/o type label attention* experiment. The results show that TLD and CEP performance drops, which demonstrates the initial explicit exception location and exception type representations are critical to the collaborative transformer between the two tasks.

Impact of Collaborative Transformer. In this experiment, we use the traditional transformer instead of the collaborative transformer, namely *w/o collaborative transformer*, thereby preventing TLD and CEP from sharing information bidirectionally. The results show a significant decrease in overall metric for TLD and CEP. The reason is that the traditional transformer only model the interaction implicitly while the collaborative transformer can explicitly consider the cross-impact between two tasks.

Impact of Bidirectional Connection. We examine the effect of bidirectional versus unidirectional information flow

TABLE III
PERFORMANCE(%) OF ABLATION EXPERIMENTS FOR TLD AND CEP.

Model	TLD				CEP
	Precision	Recall	F1 score	Accuracy	Accuracy
Co-REX	**84.7**	**80.3**	**82.0**	**78.1**	**82.4**
w/o location label attention	83.8 (↓0.9%)	79.2 (↓1.1%)	80.4 (↓1.6%)	77.2 (↓0.9%)	82.2 (↓0.2%)
w/o type label attention	84.4 (↓0.3%)	79.9 (↓0.4%)	81.1 (↓0.9%)	77.8 (↓0.3%)	81.2 (↓1.2%)
w/o collaborative transformer	82.6 (↓2.1%)	77.3 (↓3.0%)	78.7 (↓3.3%)	76.2 (↓1.9%)	78.8 (↓3.6%)
with location-to-type only	83.4 (↓1.3%)	78.0 (↓2.3%)	79.7 (↓2.3%)	77.5 (↓0.6%)	81.9 (↓0.3%)
with type-to-location only	84.3 (↓0.4%)	78.7 (↓0.6%)	81.0 (↓1.0%)	77.8 (↓0.3%)	80.3 (↓2.1%)

by keeping only one direction: either from location to type or from type to location. This is achieved by using one type of information representation as queries to attend to the other. We refer to these approaches as *with location-to-type only* and *with type-to-location only*. The results show that Co-REX outperforms both *with location-to-type only* and *with type-to-location only*. This suggests that modeling the mutual interaction between TLD and CEP enhances both tasks in a mutual way, whereas unidirectional models only consider interaction from a single direction.

V. CONCLUSION

In this study, we introduce Co-REX, a novel collaborative transformer model designed to address two interrelated tasks in runtime exception handling: TLD and CEP. Our approach innovatively establishes a bidirectional information flow between TLD and CEP, allowing for mutual enhancement and more robust model performance. The experimental results on the benchmark dataset clearly demonstrate that Co-REX outperforms existing state-of-the-art methods, providing great improvements in Precision, Recall, F1 score and Accuracy. The ablation study further underscores the critical role of the collaborative transformer module in enhancing task interconnectivity and performance. By enabling dynamic information sharing, Co-REX effectively leverages the interdependencies of TLD and CEP, illustrating a promising direction for future research in automated exception handling.

VI. ACKNOWLEDGEMENT

This paper is supported by the National Natural Science Foundation of China No. 62177015.

REFERENCES

[1] E. A. Barbosa, and A. Garcia, "Global-aware recommendations for repairing violations in exception handling," In Proceedings of the 40th International Conference on Software Engineering, pp. 858-858, 2018.

[2] M. Schuster, and K. K. Paliwal, "Bidirectional recurrent neural networks," IEEE transactions on Signal Processing, vol. 45, no. 11, pp. 2673-2681, 1997.

[3] B. Cabral, and P. Marques, "Exception handling: A field study in java and. net," In ECOOP 2007–Object-Oriented Programming: 21st European Conference, pp. 151-175, 2007.

[4] H. Shah, C. Gorg, and M. J. Harrold, "Understanding exception handling: Viewpoints of novices and experts," IEEE Transactions on Software Engineering, vol. 36, no. 2, pp. 150-161, 2010.

[5] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred api knowledge," 2018.

[6] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan, "Building useful program analysis tools using an extensible java compiler," In 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation, pp. 14-23, 2012.

[7] P. Bian, B. Liang, W. Shi, J. Huang, and Y. Cai, "Nar-miner: discovering negative association rules from code for bug detection," In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 411-422, 2018.

[8] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," ACM SIGOPS Operating Systems Review, vol. 35, no. 5, pp. 57-72, 2001.

[9] J. Wu, S. Liu, S. Ji, M. Yang, T. Luo, Y. Wu, and Y. Wang, "Exception beyond Exception: Crashing Android System by Trapping in' Uncaught Exception'," In 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track, pp. 283-292, 2017.

[10] E. A. Barbosa, A. Garcia, and M. Mezini, "Heuristic strategies for recommendation of exception handling code," In 2012 26th Brazilian Symposium on Software Engineering, pp. 171-180, 2012.

[11] E. A. Barbosa, A. Garcia, and M. Mezini, "A recommendation system for exception handling code," In 2012 5th International Workshop on Exception Handling, pp. 52-54, 2012.

[12] M. M. Rahman, and C. K. Roy, "On the use of context in recommending exception handling code examples," In 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, pp. 285-294, 2014.

[13] J. Zhang, X. Wang, H. Zhang, H. Sun, Y. Pu, and X. Liu, "Learning to handle exceptions," In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, pp. 29-41, 2020.

[14] F. Farmahinifarahani, Y. Lu, V. Saini, P. Baldi, and C. Lopes, "D-REX: Static Detection of Relevant Runtime Exceptions with Location Aware Transformer," In 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation, pp. 198-208, 2021.

[15] Y. Li, S. Ying, X. Jia, Y. Xu, L. Zhao, G. Cheng, B. Wang, and J. Xuan, "Eh-recommender: Recommending exception handling strategies based on program context," In 2018 23rd International Conference on Engineering of Complex Computer Systems, pp. 104-114, 2018.

[16] S. Mahajan, N. Abolhassani, and M. R. Prasad, "Recommending stack overflow posts for fixing runtime exceptions using failure scenario matching," In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1052-1064, 2020.

[17] R. Li, B. Chen, F. Zhang, C. Sun, and X. Peng, "Detecting runtime exceptions by deep code representation learning with attention-based graph neural networks," In 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering, pp. 373-384, 2022.

[18] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," In Proceedings of the 15th international conference on mining software repositories, pp. 542-553, 2018.

[19] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. Yu, "Multi-modal attention network learning for semantic source code retrieval," In 2019 34th IEEE/ACM International Conference on Automated Software Engineering, pp. 13-25, 2019.