# Code Clone Detection based on Event Embedding and Event Dependency

Cheng Huang, Hui Zhou*, Chunyang Ye*, Bingzhuo Li
*School of Computer Science and Technology*
*Hainan University*
Haikou, China
{chenghuang, bzli, huizhou, cyye}@hainanu.edu.cn

***Abstract*—The code clone detection method based on semantic similarity has important value in software engineering tasks (e.g., software evolution, software reuse). Traditional code clone detection technologies pay more attention to the similarity of code at the syntax level, and less attention to the semantic similarity of the code. As a result, candidate codes similar in semantics are ignored. To address this issue, we propose a code clone detection method based on semantic similarity. By treating code as a series of interdependent events that occur continuously, we design a model namely EDAM to encode code semantic information based on event embedding and event dependency. The EDAM model uses the event embedding method to model the execution characteristics of program statements and the data dependence information between all statements. In this way, we can embed the program semantic information into a vector and use the vector to detect codes similar in semantics. Experimental results show that the performance of our EDAM model is superior to state-of-the-art open source models for code clone detection.**

***Index Terms*—code search, code clone detection, event embedding, event dependency**

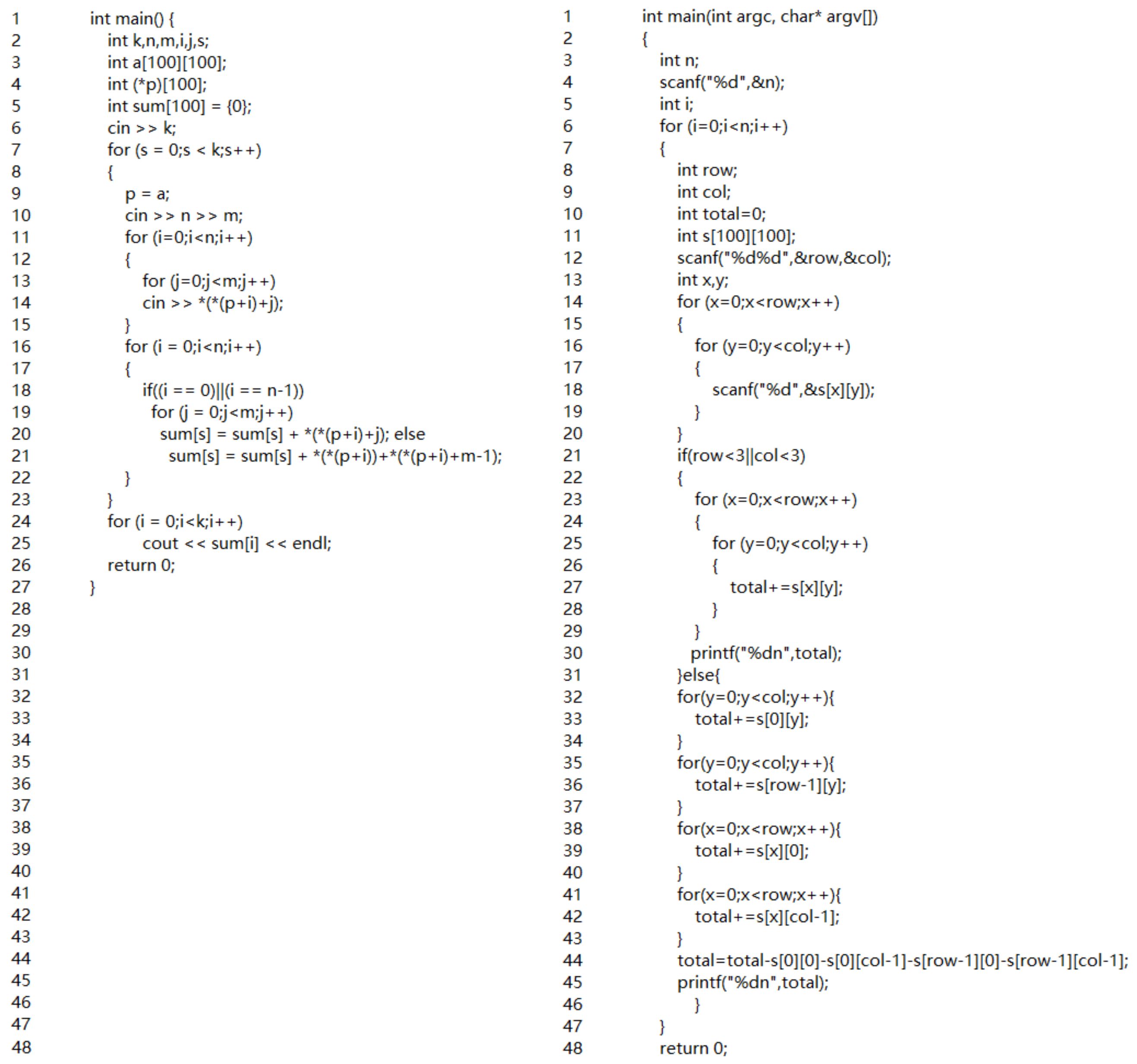
## I. Introduction

Code clone detection technology is important for many software engineering tasks (e.g., software evolution, software reuse). Existing code clone detection methods are mainly divided into three categories, namely text-based, syntax-based and semantic-based clone detection methods [1]. These code clone dection methods play an important role in program understanding, plagiarism detection, copyright protection, code compression, software evolution analysis, code quality analysis, bug detection, and anti-virus. The core functionality of the code detection model is to calculate the similarity of the code. Given a certain target code fragment, the system will first calculates the similarity between the target code fragment and all the code fragments in the database, and then returns the result according to the similarity between the codes. The similarity between codes can be classified into four levels [2]. Type-1 similarity means that the two pieces of code are identical except for the differences in spaces, layout and comments. Type-2 similarity means the code pairs are identical except for the variable name, type name, and function name. Type-3 similarity means that there are several additions and deletions of statements, and the use of different identifiers, text, types, spaces, layout and comments in the code pairs, but they are still similar. Type-4 similarity detects code pairs that are functionally similar, but they are different in text or syntax. Type-3 and Type-4 similarity can also be further divided into three categories based on their syntactical similarity values [3]: Strongly Type-3, similarity in range between 0.7 and 1.0, Moderately Type-3, similarity in range between 0.5 and 0.7, and Weakly Type-3 which similarity in range between 0 and 0.5. Weakly Type-3 clone codes are also regarded as a Type-4 clone codes.

*Corresponding author

In practice, it is easy to detect Type-1 clone samples, but it is the most difficult to detect clone samples of Type-4. The model proposed in this paper mainly focuses on the detection of Type-3/4 clone samples. The key idea of code clone detection methods is to extract some information from code fragment, and then use this type of information to identify the semantically similar code fragments. According to the type of information used, these approaches can be classified into Text-based approaches, Token-based approaches, Tree-based approaches, Metric-based approaches, and Graph-based approaches. Existing methods mainly focus on the static characteristics of the source code, and seldom consider the dynamic characteristics of the code.

As shown in Fig. 1, this code pair is selected from the data set used in this paper. They are highly similar in function, but low in grammatical similarity. The traditional code clone detection method cannot detect the clone sample shown in Fig. 1.By contrast, the CSEM model we proposed before uses event embedding to model the semantic information of a single statement, and then uses the GAT network to model the control flow information of the program. The reason for modeling control flow is that we believe that control flow of code reflects the semantic dependencies between code statements. Through the above steps, the CSEM model can encode code semantic information. The CSEM model uses the GAT network to model the dependencies of code statements. This approach may have the following problems. First of all, the code statements that are adjacent to each other in the control flow graph may not have interdependence. Since the GAT network can only model the adjacent nodes in the control flow graph, the wrong dependency is captured. For example, as shown in left part of Fig. 1, there is no dependency between the statements 5 and 6, but they are adjacent in the control flow graph. Second, limited by the structure and training cost of the



```
1    int main() {
2      int k,n,m,i,j,s;
3      int a[100][100];
4      int (*p)[100];
5      int sum[100] = {0};
6      cin >> k;
7      for (s = 0;s < k;s++)
8      {
9        p = a;
10       cin >> n >> m;
11       for (i=0;i<n;i++)
12       {
13         for (j=0;j<m;j++)
14         cin >> *(*(p+i)+j);
15       }
16       for (i = 0;i<n;i++)
17       {
18         if((i == 0)||(i == n-1))
19          for (j = 0;j<m;j++)
20           sum[s] = sum[s] + *(*(p+i)+j); else
21            sum[s] = sum[s] + *(*(p+i))+*(*(p+i)+m-1);
22       }
23     }
24     for (i = 0;i<k;i++)
25         cout << sum[i] << endl;
26     return 0;
27   }
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

```
1    int main(int argc, char* argv[])
2    {
3      int n;
4      scanf("%d",&n);
5      int i;
6      for (i=0;i<n;i++)
7      {
8        int row;
9        int col;
10       int total=0;
11       int s[100][100];
12       scanf("%d%d",&row,&col);
13       int x,y;
14       for (x=0;x<row;x++)
15       {
16         for (y=0;y<col;y++)
17         {
18           scanf("%d",&s[x][y]);
19         }
20       }
21       if(row<3||col<3)
22       {
23         for (x=0;x<row;x++)
24         {
25           for (y=0;y<col;y++)
26           {
27             total+=s[x][y];
28           }
29         }
30          printf("%dn",total);
31       }else{
32       for(y=0;y<col;y++){
33         total+=s[0][y];
34       }
35       for(y=0;y<col;y++){
36         total+=s[row-1][y];
37       }
38       for(x=0;x<row;x++){
39         total+=s[x][0];
40       }
41       for(x=0;x<row;x++){
42         total+=s[x][col-1];
43       }
44       total=total-s[0][0]-s[0][col-1]-s[row-1][0]-s[row-1][col-1];
45       printf("%dn",total);
46         }
47     }
48     return 0;
```

Fig. 1. A clone code pair

GAT network, we can only model the statement dependency on the first-order adjacent nodes in the control flow graph. Therefore, it is difficult to capture the relationships between program statements that have semantic dependencies, but are far apart in the source code (usually not adjacent in the control flow graph). For example, as shown in right part of Fig. 1, statement 11 and statement 27 have a dependency relationship of variable s, but the GAT network usually cannot model this type of dependency relationship. In order to solve the above-mentioned limitations of the CSEM model, we propose an EDAM model based on event-dependent graphs. Compared with the CSEM model, the EDAM model directly focuses on the semantic dependencies between statements.Therefore, the EDAM model solves the limitations of the CSEM model we mentioned above. In our experiments, the EDAM model also has better prediction accuracy.

The advantage of our method is that it can model the dynamic semantics of the code. The key-idea of this paper is that a program can be regarded as a series of interdependent events that occur continuously. Therefore, by modeling these events, we can extract the dynamic semantic information of the program. The challenges of this method are how to extract the dynamic semantic information of the code from the source code file, and how to enable the model to use this information to calculate the semantic similarity of two pieces of code. In order to address the above challenges, we propose a model

EDAM that uses event embedding to model the execution characteristics of the program's statements and the data dependence (event dependency) between different statements. We perform event dependency analysis on the source code and transform it into an event dependency graph. The event dependency graph describes the statement execution characteristics of the code and the event dependency relationship between different statements. After that, we input the event dependency graph into the event dependency execution engine, which will analyzes and calculate the event dependency graph, and finally output the vector representation of the program. This vector contains the event-dependent semantic information of the program, so it can be used for semantic-based code clone detection.

The main contributions of this paper are as follows:

- We propose a code clone detection method, which regards the program as a series of continuous interdependent events. We use this key-idea to model the dynamic semantic information of the code. Then we use the embedded vector to measure the semantic similarity of the code. Experimental results show that our method has certain advantages over the comparison models in Type-3/4 code clone detection.
- This article develops a tool chain for the model. The tools in the tool chain can generate the event dependency graph of the program through the event dependency analyzer. Then event embedding execution engine will calculates code semantic vector by using event dependency graph. The code semantic vector can be used to perform code clone detection tasks.

The rest of this paper is organized as follows: Section. II reviews the works in code clone detection. Section. III introduces some preliminary about clone detection. Section. IV describes the EDAM model in detail. Section. V evaluates our EDAM model through experiments. Section. VI summarizes the work of this paper and highlights some future work directions.

## II. RELATED WORK

The code clone detection model has important value in software engineering tasks such as bug detection, copyright protection, software evolution analysis, and anti-virus. Therefore, the study of code clone detection has attracted more attention in recent years. For example, Kim et al. present a model called VUDDY to find out the code pairs that contain the similar risk of error [4], [5]. The core function of the code clone detection model is to convert the code into its vector representation, and then calculate the similarity based on the code embedding vector, thereby selecting the most suitable one or more pieces of code from the candidate code segments as the result. According to the different types of code information used in the code embedding model, existing methods can be divided into the following categories: text-based methods [6]–[9], token-based methods [10]–[12], metric based methods [13], [14], tree-based methods [15], [16], and graph-based methods [17]–[19].

According to the difficulty of code clone detection, it can be divided into Type-1, Type-2, Type-3 and Type-4 code clone detection [2]. The Type-1/2 code clone detection is simple, while the Type-3/4 code clone detection is more difficult. SourcerCC [20], CloneWork [14], Nicad [21] and CCLearner [22] are the most popular models in Type-3 code clone detection. SourcerCC and CloneWork are both hybrid models based on token and index. Nicad is a model that embeds programs based on textual information. Nicard filters and normalizes the code fragments to eliminate the interference of irrelevant factors on the prediction results of the model. CCLearner works on the lexical level, divides program tokens into eight categories and then represents programs as token-frequency list vectors. Then, for each code pair, CCLearner computes a similarity score between the token-frequency lists to identify the similar code pairs. Deckard utlizes abstract syntax tree to embed the program, and then it clusters the embedding vectors of the program to identify similar code segments.

For type-4 detection, Gabel et al. detect semantic similarity by augmenting Deckard [23] with a step to generate vectors for semantic similarity codes [24]. Jiang et al. propose a method to detect semantic similarity codes by executing code fragments against random inputs [25]. Wei and Li use LSTM to generate the vector expression of the code segment, and then calculate the hamming distance of them with the hash function to determine whether they are similiar pairs [26]. The pairs with the smaller hamming distance are classified as similiar pairs. DeepSim encodes the control flow and data flow of a code fragment into a semantic matrix, based on which a deep neural network is designed to measure code functional similarity [27]. Oreo represents the code characteristics using 24 metrics, including the number of variables declared [28]. Then, they train a binary classifier using a vector of 48 dimensions, which corresponds to a pair of code fragments as an input into a symmetrically structured Siamese network. Li et al. propose the CSEM model [19], which uses the GAT network to model the program control flow information, so that the semantics of the program can be embedded.

Different from SourcerCC, Nicad and CloneWork, EDAM is a code clone detection model based on the event dependency graph (a data structure proposed in this paper, C.f. Section. IV), which has a better ability to model code semantics. Also different from the CCLearner and Oreo models, our EDAM model does not need to manually select the mertrics that need to be collected from the code. Different from the DeepSim model, the EDAM model uses an event dependency graph instead of a control flow graph to capture the execution semantics of the code. Different from the CSEM model, the EDAM model introduces the concept of event dependency, which can model the data flow information of the program. Compared with the CDLH model that uses the LSTM network to model the program abstract syntax tree, the EventTransformer used in the EDAM model has a stronger ability to model program semantics. In summary, the EDAM model proposed in this paper can more effectively model the semantic information of the code.

## III. Preliminary

### A. event embedding

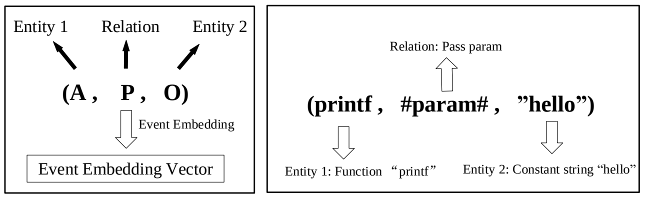


Fig. 2. Event embedding in program

We describe the definition of event embedding as follows: Consider a triple $(A, P, O)$ as shown in Fig. 2, where $A$ is entity 1, $O$ is entity 2, and $P$ is the relationship between two entities. The purpose of event embedding is to convert triples $(A, P, O)$ into a vector $X$.

The program can be regarded as a series of interdependent events that occur continuously, so we can use the event embedding method to model the semantics of the program. As shown in Fig. 2, the function statement printf("hello") can be regarded as an event, in which the function printf is regarded as entity 1, the string constant "hello" as the function parameter is regarded as entity 2, and the relationship "passing parameters" between the two entities is regarded as the relationship $P$. Therefore, the event print ("hello") can be converted into a vector via event embedding. It should be noted that in Fig. 2, "#param#" wrapped by "#" represents the relationship between entities.

### B. event dependency

In programming languages, a statement usually contains more than one event, such as the statement this.printf("hello", p), which is composed of multiple events. As shown in Fig. 3, the event embedding process of the statement this.printf("hello", p) can be regarded as a tree structure. The calculation process of event embedding needs to follow the rule of calculating from the leaf nodes of the tree to the root node in turn. The embedding process of the entire statement can be defined as follows:

$$\begin{aligned} e_1 &= embed(constantStr, \#parammix\#, p) \\ e_2 &= embed(e_1, \#param\#, printf) \\ e_3 &= embed(e_2, \#invoke\#, this) \end{aligned} \tag{1}$$

where $\#parammix\#$ represents the mixing of parameters before passing parameters, $\#param\#$ represents passing parameters, $\#invoke\#$ represents function calls, $e_1$, $e_2$ are intermediate events, and $e_3$ is the final result of the event embedding of the statement. In this case, we say that event $e_2$ depends on event $e_1$, and event $e_3$ depends on event $e_2$.

Note that the event dependency exists not only in a single program statement, but also between multiple statements. As shown in Fig. 4, the embedding process of the statements shown in the bottom of the figure can be transformed into two

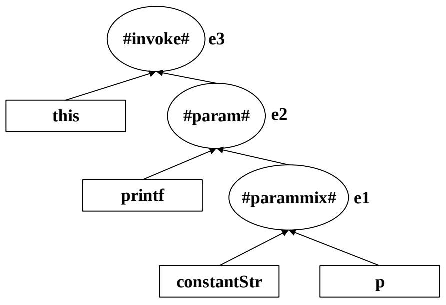
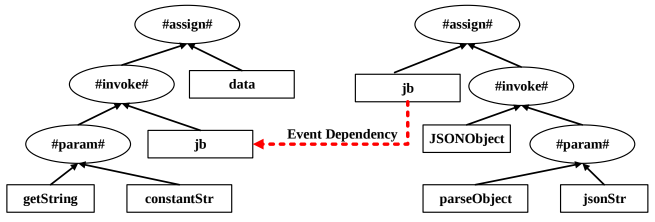


Fig. 3. Event embedding tree



```
JSONObject jb = JSONObject.parseObject(jsonStr);
String data = jb.getString( "data" );
```

Fig. 4. Event dependency in different statements

event embedding trees. In this case, the event embedding of the second statement depends on the embedding result of the first statement. Therefore, an event-dependent relationship arises between the two statements. Our EDAM model can consider both these two types of dependencies. Note that the concept of event embedding tree is to facilitate the explanation of the embedding calculation process of continuous events. The dependency of events in the event embedding trees form an event dependency graph, which is implemented in our EDAM model.

## IV. Model

### A. Overall Structure

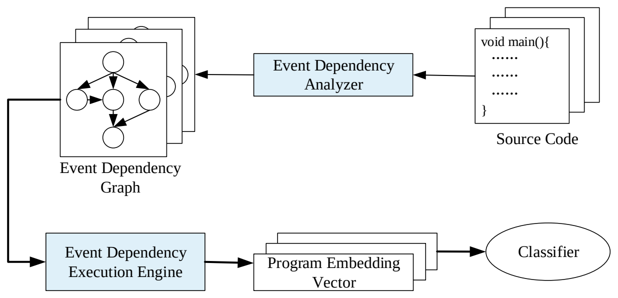


Fig. 5. The structure of our model.

As shown in Fig. 5, the core components of our model are the event dependency analyzer and the event dependency execution engine, where the event dependency analyzer is responsible for converting the source code into an event dependency graph. The event execution engine is responsible for converting the event dependency graph into a vector representation of the program through calculation. The event dependency graph is a directed acyclic graph. It describes not only the event dependency of each statement in the program (as shown in Fig. 3), but also the event dependency between different statements. By analyzing the event information provided in the event-dependent graph, the event-dependent execution engine calculates the event semantics of the program to obtain the final program semantic vector. The classifier is responsible for calculating the similarity of the code pair according to the resulting program vector representation and returning the classification result (i.e., whether it is a similar code pair).

## *B. Preprocess*

In this section, we introduce the preprocessing process of the data set.

**Operator**:

For the C language, we identify 38 common operators (e.g, assign, return, param, invoke, parammix, sizeof, structure access). These operators play the role of P in the event triple $(A, P, O)$, and are used to describe the relationship between two entities. For example, the operator in the event $c\# < \#1$ is $<$, which represents a numerical comparison between the variable entity $c$ and the constant entity 1.

**Analysis of event dependence**

We use an event analyzer to analyze the event dependency in the program according to the defined operator and generate an event dependency graph of the program. There are two main types of event dependency in program code. The first is the event dependency within a single program statement, and the second is the event dependency between multiple program statements (e.g Section 3.2). We use event dependency graphs to describe these two dependencies at the same time.

As shown in Fig. 6, the left part of the figure is a simple C language code fragment, and the right part is its corresponding event dependency graph generated by the event dependency analyzer. In this code, the event dependencies exist not only within a single statement, but also between multiple statements. Each node in the event dependency graph represents an event, and each edge from A to B represents that event B depends on event A. According to the rules of our event dependency analyzer, each statement in the code fragment on the left can be parsed into a subgraph in the code event dependency graph on the right. The color of each statement in the left picture corresponds to the color of the event-dependent subgraph of the corresponding statement in the right picture. For example, the last statement printf("d%", maxs) in the left figure is marked in blue, which corresponds to the nodes 11, 12, and 13 that are also marked in blue in the event dependency graph.

The event dependency graph can not only represent the event dependency within a single statement, but also reflect the event dependency between multiple statements. For example, the printf statement uses the variable maxs, and maxs is assigned in the previous statement. There is an event dependency relationship between these two statements. This dependency relationship is defined in the event dependency graph, that is, there is an edge from node 10 to node 11. The value of Entity1 in node 11 is node 10, which means that the calculation result of node 10 will be represented as a vector of entity 1 in node 11. In the event-dependent execution engine, we first use topological sorting to split the event embedding tasks described by the event-dependent graph to ensure that the event embedding nodes that need to rely on the pre-node must be performed after the calculation of pre-node is completed.

## *C. Event dependent execution engine*

The event-dependent execution engine is responsible for performing calculations based on the event-dependent graph to generate the semantic vector of the code. The execution process of the event-dependent execution engine is shown in Fig. 7. The execution engine receives the event dependency graph as input, and then uses the event Transformer defined in this paper to embed the events described in the event dependency graph. The output of the event transformer is a matrix composed of the event embedding vectors of each node in the event dependency graph. We call this matrix the event embedding matrix. The vector in the i-th row of the event embedding matrix is the event embedding vector corresponded to node i in event dependency graph calculated by Event Transformer. After that, we input the event embedding matrix into the restore layer. The restore layer converts the event dependency matrix into a program embedding matrix where the kth row of the program embedding matrix represents the event embedding result of the kth row statement in the source code. Finally, we use the convolutional layer to extract the semantics of the program embedding matrix and generate the program embedding vector. In this way, the program embedding vector contains the dynamic semantic information of the source code fragment.

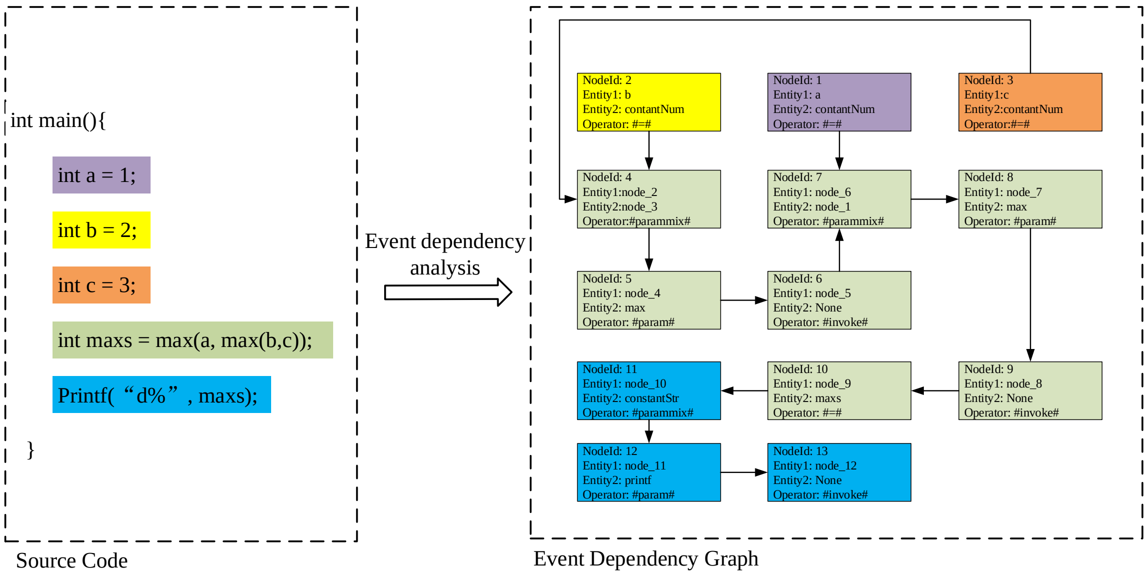
**Event Cell**

Event cell is the basic unit for event embedding calculation. Given a triple $(A, P, O)$, the Event Cell is responsible for converting the event $(A, P, O)$ into the corresponding event embedding vector. The structure of Event Cell is shown in Fig. 8. We define its calculation process as follows:

$$\begin{aligned} e^k &= concat\left(vec\left(A\right) * T_{p1}^k,\ vec\left(O\right) * T_{p2}^k\right) \\ a &= concat(e^1, e^2, \cdots e^k) \\ o &= Dense(a) \end{aligned} \tag{2}$$

where $T_{p1}$ and $T_{p2}$ are two tensors with dimensions $(k, width, height)$ associated with a specific operator $P$. These two tensors are responsible for mapping entity vectors to multiple high-dimensional vector spaces. $T_{p1}^k$ represents the k-th matrix in the tensor, and the size of the matrix is $(width, height)$. $A$ and $O$ are entity 1 and entity 2 respectively. $Vec$ represents a function that maps entities $A$ and $O$ to their corresponding vectors. The choice of the vec function can be diverse. One can use models such as word2vec or BERT, or implement it in a custom way. The $Dense$ function represents a fully connected layer.

In this paper, the $vec$ function we used treats different variable names and function names as different entities. For



```
int main(){
    int a = 1;
    int b = 2;
    int c = 3;
    int maxs = max(a, max(b,c));
    Printf( “d%” , maxs);
}
```

Event dependency analysis
NodeId: 2 Entity1: b Entity2: contantNum Operator: #=#
NodeId: 1 Entity1: a Entity2: contantNum Operator: #=#
NodeId: 3 Entity1:c Entity2:contantNum Operator:#=#
NodeId: 4 Entity1:node_2 Entity2:node_3 Operator:#parammix#
NodeId: 7 Entity1: node_6 Entity2: node_1 Operator: #parammix#
NodeId: 8 Entity1: node_7 Entity2: max Operator: #param#
NodeId: 5 Entity1: node_4 Entity2: max Operator: #param#
NodeId: 6 Entity1: node_5 Entity2: None Operator: #invoke#
NodeId: 11 Entity1: node_10 Entity2: constantStr Operator: #parammix#
NodeId: 10 Entity1: node_9 Entity2: maxs Operator: #=#
NodeId: 9 Entity1: node_8 Entity2: None Operator: #invoke#
NodeId: 12 Entity1: node_11 Entity2: printf Operator: #param#
NodeId: 13 Entity1: node_12 Entity2: None Operator: #invoke#
Source Code
Event Dependency Graph

Fig. 6. Event dependency graph

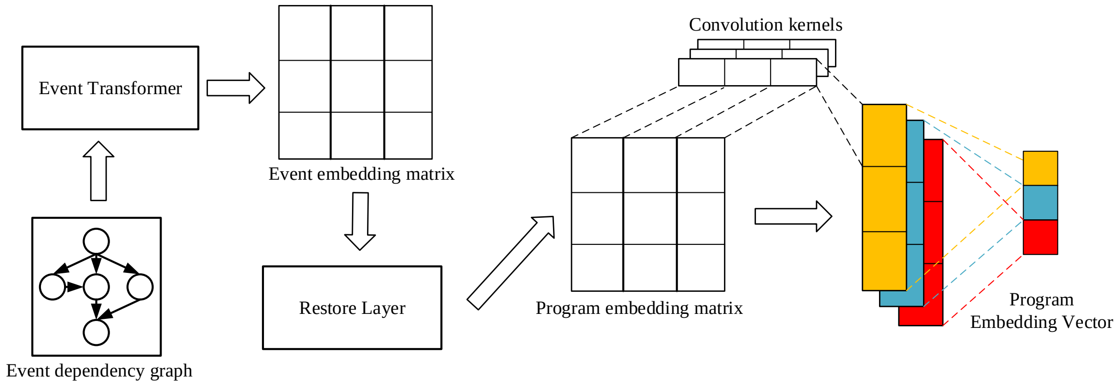


Fig. 7. Execution process of event dependency execution engine.

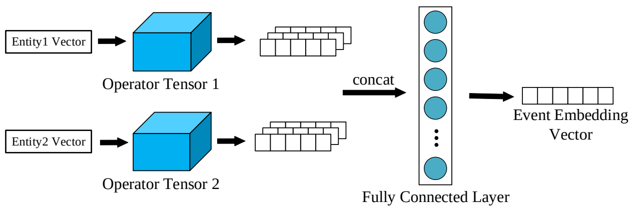


Fig. 8. The structure of Event Cell

example, printf (a, b, c) contains four entities printf, a, b and c. Given a set of entities in a code fragment $N = n_1, n_2, ..., n_d$, the vec function counts the number of times each entity appears in the code fragment, and then the vec function renames the entity to the form of $Top_i$ according to the number of times each entity appears in the source code, where $i$ indicates that the number of occurrences of the entity in the source code ranks i-th. Afterwards, the $vec$ function initializes different $Top_i$ to a random entity vector.

In Event Cell, we use three-dimensional tensors $T_{p1}$, $T_{p2}$ instead of two-dimensional matrices to map entity vectors to high-dimensional spaces. This is mainly because three-dimensional tensors can map entity vectors to multiple high-dimensional spaces, which can improve the semantic expression ability of entity vectors.

**Event Transformer**

Event Cell can perform event embedding calculations on a single triple $(A, P, O)$. The EventCell module can perform the event embedding calculation of the program. However, this method has certain drawbacks. As shown in Fig. 6, the event embedding of node 13 depends on the event embedding results of nodes 1 to 12, and these event embedding calculations constitute an event embedding calculation chain. This chain structure causes us to use Event Cell multiple times to recursively calculate each event on the chain. This brings an obvious disadvantage: when the event embedding calculation chain is very long, the early event information may be forgotten by the model. Especially when a statement is very long and needs to rely on the result of multiple pre-event embeddings, this problem becomes even more important. Therefore, we introduce a gate mechanism to solve this problem [19]. The gate mechanism can enhance the model's ability to remember early events. We introduce the gate mechanism into the Event Cell and call it Event Transformer. The structure of Event Transformer is shown in Fig. 9. We define the forward calculation process of Event Transformer as follows:

$$\begin{aligned} r_t &= \sigma(W_r[A_{t-1},\ O_t]) \\ z_t &= \sigma(W_z[A_{t-1},\ O_t]) \\ \widetilde{A_t} &= Ec\left(r_t * A_{t-1}, P_t, O_t\right) \\ A_t &= (1 - z_t) * A_{t-1} + z_t * \widetilde{A_t} \end{aligned} \tag{3}$$

where $W_r$ and $W_z$ are the weights of reset gate and update gate, respectively. $A_{t-1}$ represents the calculation result of the previous time step in the event embedding calculation chain. $E_c$ represents the Event Cell modeule. $A_t$ is the calculation result of the current time step in the event embedding calculation chain. $P_t$ is the operator corresponding to the current time step.

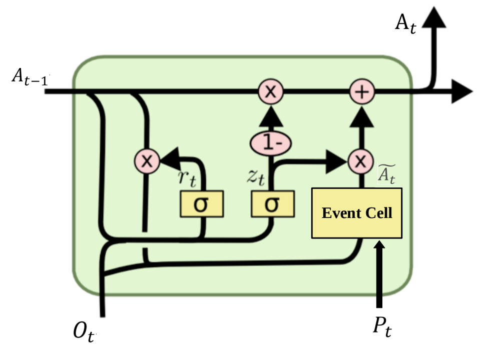


Fig. 9. The structure of Event Transformer.

**Restore Layer**

**Algorithm 1:** Algorithm of restore layer

```
Data: event embedding matrix $E$ of size $w * l$
Result: program embedding matrix $O$
1  $O \leftarrow []$;
2  for $k \leftarrow 0$ to $w$ do
3      currentNode ← node withd id k in event
         dependency graph;
4      if currentNode is the final event in a statement
         then
5          $O$.append($E[k]$);
6      else
7          continue;
8      end
9  end
10 return $O$;
```

Through the above-defined Event Cell and Event Transformer, we can obtain the event embedding matrix composed of the event embedding vector of each node in the event dependency graph. In event embedding matrix, the i-th row vector of matrix is the event embedding result of node with id i in event dependency graph. Since multiple nodes of the event embedding graph usually only correspond to one statement in the original code fragment, we need a restore layer to convert the event embedding matrix into a program embedding matrix. In the program embedding matrix output by the restoration layer, the kth row vector of the matrix is the event embedding result of the kth statement in the source code. The algorithm of the restore layer is defined as in Algorithm. 1.

**Convolutional Layer**

The convolutional layer converts the program embedding matrix into a program embedding vector, which is used for the subsequent code similarity calculation. In the EDAM model, we use multiple convolution kernels and pooling layers to perform this conversion. We define the calculation process of the convolution layer as follows:

$$\begin{aligned} \hat{X} &= Padding(X) \\ Z &= W_{conv} * \hat{X}^T \\ Q &= Pool(Z) \end{aligned} \tag{4}$$

where $X$ represents the program embedding matrix, we let $bz$ represent the number of samples in each batch of training data, $n_o$ represents the number of nodes in each program embedding matrix, $l_n$ represents the length of each node vector of the event dependency graph, then the dimension of $X$ can be expressed as $(bz, n_o, l_n)$. It should be noted that because the number of nodes in different event dependency graphs is inconsistent, we need to padding $X$ before proceeding to the next calculation. $\hat{X}$ represents the program embedding matrix after padding. $W_{conv}$ is a matrix composed of multiple one-dimensional convolution kernels, we let $n_k$ represent the number of one-dimensional convolution kernels, $l_k$ represents the length of each convolution kernel, and the shape of $W_{conv}$ can be expressed as $(n_k, l_k)$. Pool represents the average pooling operation on the last dimension of $Z$. $Q$ is the program embedding vector output by the convolutional layer, and its shape is $(bz, n_k)$. After passing through the convolutional layer, a piece of source code can be expressed as an $n_k$ length program embedding vector.

## D. Code Similarity Evaluation

Given the event embedding vectors of two pieces of code, we use cosine similarity to evaluate the semantic similarity of the two pieces of code. In the code clone detection system, we will calculate the similarity between the target code and all candidate codes, and then determine whether the code pair is a similar code that should be returned by the system according to the similarity threshold $\theta$. When the cosine similarity of two pieces of code is greater than $\theta$, we judge them as a similar code pair, and when the cosine similarity is less than $\theta$, we judge them as a non-similar code pair.

## E. Parameter Learning

We define the loss function as follows:

$$\begin{aligned} g(x_i) =& Conv(Et(x_i)) \\ sim(x_i, x_j) =& \frac{g(x_i) \cdot g(x_j)}{||x_i|| * ||x_j||} \\ Loss =& \sum_{min}^{max} max(0, 1 - sim(x_k, x_m) + sim(x_k, \hat{x}_k)) \end{aligned} \tag{5}$$

Where $Et$ represents the Event Transformer module. $Conv$ represents the convolutional layer, and $g(x_i)$ is the program embedding vector of the sample $x_i$. $similairity$ represents a function for calculating the similarity of two samples. In the Loss function, the code pair $(x_k, x_m)$ is a positive sample, and $(x_k, \hat{x}_k)$ is a negative sample pair. We use random sampling to collect negative samples. The experimental results show that the use of negative samples in the model training process can effectively improve the model's detection ability.

## F. Back Propagation Through Event(BPTE)

BPTT algorithm is used in the traditional GRU unit training process because the weights of reset gate, update gate and hidden layer can be shared through each time step. In the calculation process of Event Transformer, $T_{p1}$ and $T_{p2}$ in the Event Cell are determined by the operators in each embedded step. Therefore, we propose the BPTE algorithm to train Event Transformer. The back propagation process of BPTE algorithm is defined as follows:

$$\frac{\partial E_t}{\partial T_p} = \sum_{k \in N_p} \frac{\partial E_t}{\partial A_t} * \frac{\partial A_t}{\widetilde{A_t}} \left( \prod_{j=k+1}^{t} \frac{\partial \widetilde{A_j}}{\partial \widetilde{A_{j-1}}} \right) * \frac{\partial A_k}{\partial T_p} \tag{6}$$

where $E_t$ is the training error at step t, and $N_p$ is the set of positions where operator $p$ appears in the embedding tree. For example, for embedding $a$–$b$+$c$–$d$, operator '$-$' appears in the embedding tree at positions 0 and 2, So $N_p = \{0, 2\}$. Through the BPTE algorithm, we can train the Event Transformer. Since we have introduced a gate mechanism in the Event Transformer, the Event Transformer can better fit the historical data in event embedding calculation chain. Also, the exploding gradient and vanishing gradient during the backpropagation can be addressed.

## V. Evaluation

In this section, we evaluate our proposed model through experiments.[1]. We mainly hope to answer the following questions through experiments:

- **RQ1:** What are the advantages and disadvantages of our EDAM model compared to other commonly used models?
- **RQ2:** How does the similarity threshold affect the model prediction results?
- **RQ3:** Is it possible to improve the prediction accuracy of the model by simultaneously using multiple EDAM models for classification? Are these models complementary?
- **RQ4:** Can the program embedding vector express the semantic relevance of the code?

[1]The dataset and the implementation of our model will be released in github.com after the paper is accepted.

## A. Experiment Setup

In this section, we introduce the details of setup in our expirement.

**Data Set**

In the experiment, we use the OJClone dataset. This open source dataset contains 104 OJ questions, each of which contains 500 user-submitted programming codes. Since each code fragment in the data set belongs to an independent problem, we regard the code fragments belonging to the same problem as functionally similar code pairs, and the code fragment belonging to different problems as non-similar code pairs. Also, the code fragments belonging the same question are submitted by different users, therefore they can be regarded as semantic similarity code clone pairs of Type-3/4. Without loss of generality, we select the 10 questions to evaluate our model, and for each question we randomly select 100 code fragments. In total, there are 1000 code fragments. We select 70% of the samples in each problem to generate the training set, and the remaining 30% of the samples to generate the test set. The training set contains 49,000 positive samples and 49,000 randomly sampled negative samples (96,000 in total). The test set contains 44850 samples, of which 4350 are positive samples and 40500 are negative samples.

**Metrics**

Three metrics are used to evaluate the performance of the models in the experiments, i.e., Precision, Recall and F1 Score. The metrics are defined as follows:

$$\begin{aligned} Precision =& \frac{TP}{TP + FP} \\ Recall =& \frac{TP}{TP + FN} \\ F1 =& \frac{2 * Precision * Recall}{Precision + Recall} \end{aligned} \tag{7}$$

where $TP$ represents the number of true positive samples, $FP$ represents the number of false positive samples. $FN$ represents the number of dalse negative samples. Precision mainly measures the accuracy of the model to classify positive samples. Recall mainly reflects the model's ability to cover the similar samples. The above two indicators are difficult to evaluate the true ability of the model when used alone, so in order to integrate the two evaluation indicators of Precision and Recall, the $F1$ Score is also introduced. This metric combines Precision and Recall, which can better evaluate the detection ability of the model.

## B. Comparison of Different Models (RQ1)

In this section, we compare the EDAM model with five state-of-the-art open source code clone detection models, which are CCLearner, Deckard, CloneWork, SourcerCC and CSEM. Among these models, Deckard, CloneWork, CSEM

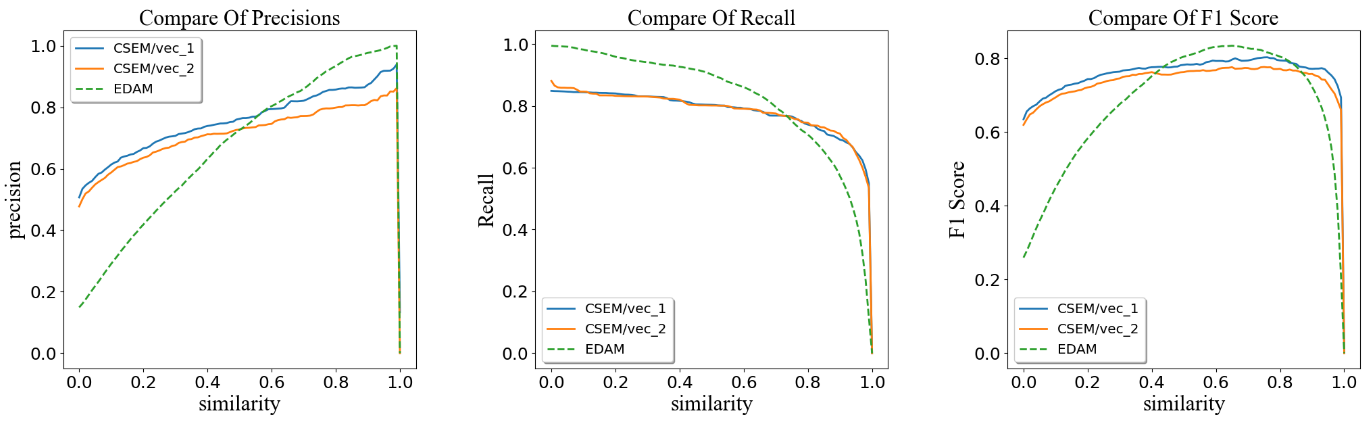


Fig. 10. Impact of Similarity Threshold.

TABLE I
Comparison of Different Models.

| Model | Precision | Recall | F1 Score | Tool Configuration |
|---|---|---|---|---|
| CCLearner | 0.0651 | 0.8324 | 0.1207 | $\theta = 98\%, lr = 0.001, epoch = 300$ |
| Deckard | 0.4201 | 0.3411 | 0.3765 | $\theta = 95\%, MIT = 30, Stride = 2$ |
| CloneWork | 0.4604 | 0.3367 | 0.3890 | $\theta = 40\%, MIT = 1, Mode = type3pattern$ |
| SourcerCC | 0.5124 | 0.3210 | 0.3947 | $\theta = 50\%, MIT = 1$ |
| CSEM/$vec_1$ | 0.8036 | 0.7192 | 0.7583 | $\theta = 80\%, K = 2, tk = 10, GA = [8, 1]$ |
| CSEM/$vec_2$ | 0.7554 | 0.7293 | 0.7416 | $\theta = 70\%, K = 2, GA = [8, 1]$ |
| EDAM | **0.8029** | **0.8606** | **0.8307** | $\theta = 70\%, K = 2, GA = [8, 1]$ |

and SourcerCC can support the detection of C language code. The CCLearner model is a deep learning model for code feature extraction based on token. Its open source version only supports the detection of Java code. We have extended its code to support the processing of C language code. Since Type1/2 similarity codes are very easy to be detected, and almost all detection models can get good results, our experiment mainly focuses on the detection ability of Type3/4 code fragments.

The hyperparameter settings of these models are shown in Table. I. When setting the hyperparameters of the comparison model and our EDAM model, we follow the principle of making the model's F1 Score perform best. In the table, $\theta$ represents the similarity threshold of the model, and lr represents the training learning rate. MIT is the shortest sequence length considered by the model. K is the length of the first dimension of the Operator Tensor in the Event Cell. GA indicates the number of multi-head attention in GAT layer (GA = [8, 1] indicates that there are two sub-layers in GAT layer, and the number of multi-head attention is setting to 8 and 1, respectively). For Deckard, we set its Stride to 2. For CloneWork, we set its detection mode to the highest type3pattern.

The experimental results are shown in Table. I. Our EDAM model is significantly better than other models in Precision, Recall, and F1 Score.We think this is because EDAM can better model the semantic information of the code. Compared with the CSEM model, which is also the code embedding model based on event embedding, our EDAM model still has a big improvement. We think this is mainly because we have introduced event-dependent information into the model. In our experiments, the performance of the CClearner model was poor. This is because the samples in our data set are homework codes submitted by students. The variable names in these code file are relatively simple, and there are a large number of simple variable names and function names such as a, b, and c.The CCLearner model extracts code semantics based on tokens such as variable names and function names. These simple token names in the data set interfere with CCLearner's ability to correctly extract code semantic similarity, causing the CCLearner model to think that almost all code fragments are similar.

## C. *Similarity Threshhold(RQ2)*

In this section, we mainly analyze the impact of different similarity thresholds on the model. We chose three models for comparison. The experimental results are shown in Fig. 10. The F1 Score of the EDAM model performs best when the similarity threshold is set to 60%. The performance of the two sub-models of CSEM, CSEM/$vec_1$ and CSEM/$vec_2$ are relatively close, and CSEM/$vec_1$ is slightly better than CSEM/$vec_2$ in terms of precision and F1 Score. At the same time, we can observe an interesting phenomenon, that is, the slope of the first half of the curve of the CSEM model is relatively flat, and after reaching a certain threshold, the slope of the curve changes dramatically. This means that the program vector generated by the CSEM model is likely to appear in a small vector space, which is not convenient for the model to make further distinctions between positive and negative samples.The slope of the EDAM curve in the figure is obviously greater than the slope of the CSEM's, which indicates that the EDAM model is more sensitive to the similarity threshold than the CSEM model. We believe that this phenomenon reflects that the program embedding vectors generated by the EDAM model are distributed in a wider vector space. These vectors have higher cohesion when the categories are the same, and higher separation when the

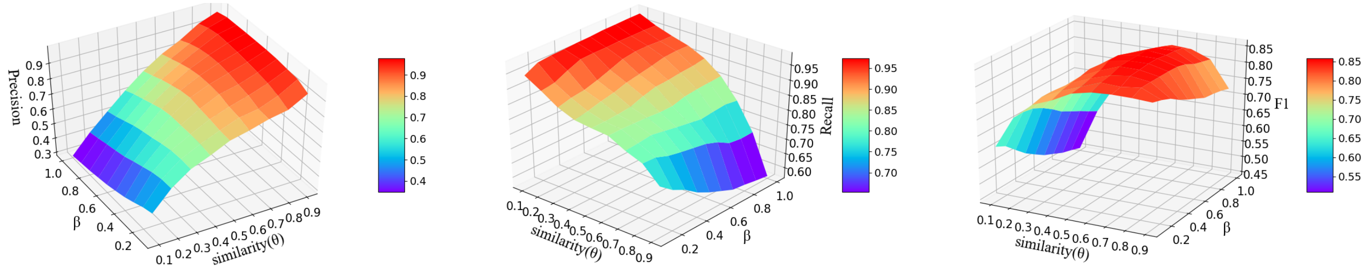


Fig. 11. Impact of fusion parameters

categories are different. In Section. V-E, we visualized the program embedding vectors generated by the EDAM model, and the results partially proved our above conclusions.

## D. Multi-model fusion(RQ3)

TABLE II
COMPARISON OF FUSION MODELS.

| Model | Precision | Recall | F1 Score | Tool Configuration |
|---|---|---|---|---|
| CSEM/$f_1$ | **0.9411** | 0.6914 | **0.7971** | $\theta_{vec_1} = 80\%, \theta_{vec_2} = 70\%$ |
| CSEM/$f_2$ | 0.8856 | 0.7731 | 0.8255 | $\theta = 70\%, \beta = 0.6$ |
| EDAM/$f_1$ | 0.8818 | **0.8486** | **0.8648** | $\theta = 50\%, \beta = 0.6$ |

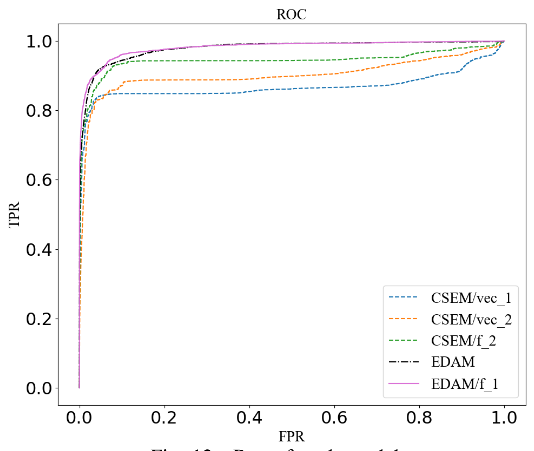


Fig. 12. Roc of each model

In the experimental process of this paper, we found that fusing the calculation results of multiple EDAM models can significantly improve the prediction accuracy of the model. This is because different EDAM models have different classification tendencies for the same samples due to random factors such as different sample input orders during the training process. Therefore, in this section, we propose an EDAM-based fusion model EDAM/$f_1$, and we analyze the difference in performance between the CSEM-based fusion models and the EDAM-based fusion models. The purpose of model fusion is to enhance the classification ability of the model by using the prediction results of multiple models at the same time.

The EDAM/$f_1$ model fuses the calculation results of two independent EDAM models, The formula is defined as $score(x1, x2) = \beta * score1(x_1, x_2) + (1-\beta) * score2(x_1, x_2)$, where $score1$ represents the score of the first EDAM model, $score2$ represents the score of the second EDAM model, and $\beta$ is used as a weight to balance the impact of the two models on the final score.

The experimental results are shown in Table. II, where CSEM/$f_1$ and CSEM/$f_2$ are two CSEM-based fusion models proposed by Li et al. The difference between the two lies in the way of fusion of the model. The CSEM/$f_1$ model performs best on Precisoion, while the EDAM/$f_1$ model performs best on Recall and F1 Score. This phenomenon indicates that the CSEM/$f_1$ model and EDAM/$f_1$ model have their own advantages in different types of tasks. When we need the similar samples predicted by the model to be as accurate as possible, the CSEM/$f_1$ model should be considered, because its Precision can reach 0.94, which means that when the CSEM/$f_1$ model judges a sample as a semantically similar sample, the reliability of the classification results is 94%. Otherwise, when we need the model to detect as many similar samples as possible, we should consider using the EDAM/$f_1$ model, because its Recall reaches 0.84, which means that when the EDAM/$f_1$ model is used, 84% of the similar samples in data set can be detected by the model.

We also compared the ROC of the models, and the experimental results are shown in Fig. 12. The ROC of CSEM/$f_1$ is significantly better than CSEM/$vec_1$ and CSEM/$vec_2$ models. After using multiple EDAM models for fusion, the EDAM/$f_1$ model has a further improvement compared to the EDAM model, so the EDAM-based fusion model EDAM/$f_1$ performs best among all models.

In order to further explore the influence of different fusion parameter settings on the EDAM/$f_1$ model, we analyzed the fusion parameter settings of the EDAM/$f_1$ model. The experimental results are shown in Fig. 11, where similarity($\theta$) and $\beta$ represent the similarity threshold and the balance parameter respectively. The z-axis of the three subgraphs represent the three evaluation metrics ,which are Precision, Recall, and F1. According to experiment result shown in Fig. 11, we can observe that when $\theta$ is set to 0.5 and $\beta$ is set to 0.6, the F1 Score of the model performs best.

## E. Visualization of code semantics(RQ4)

In this section, we use the UMAP algorithm to visualize the program semantic embedding vector generated by the EDAM model. The purpose of this study is to verify whether the code semantic embedding vector generated by the model can show semantic relevance in the high-dimensional vector space.

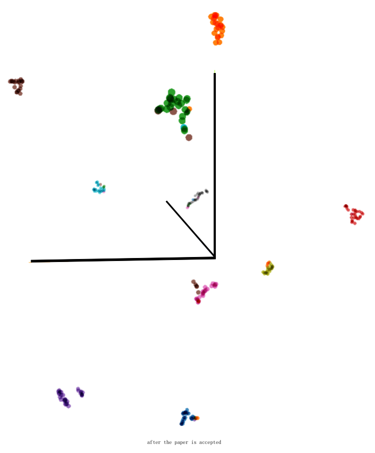


Fig. 13. Visualization of program embedding vectors

The experimental results are shown in Fig. 13. Each point in the figure represents a data point generated by mapping a program embedding vector to a three-dimensional space through the UMAP algorithm, and its color represents the sample category of the point. We used 10 categories of samples in the experiment, so the data points in the figure have 10 colors. We can observe that the data points in the graph are clustered into different clusters according to the different colors, while the data points of different colors are separated from each other. This result shows that our model can effectively learn the semantic information of the code, so that the code embedding vector generated by our model has good semantic expression ability in high-dimensional space.

### *F. Threats to Validity*

The initialization of the network parameters and the negative samples selected during the training process will affect the performance of the model. To alleviate the influence of these confounding factors, we will train multiple models and use their average results to evaluate our model.

Uncertain factors in the model training process will also affect the performance of the model. In order to solve this problem and improve the performance of our model, we use a hybrid model to combine the advantages of the two models at the same time and improve the performance of our model.

In experiments, biased dataset may affect the generalization of the results. Therefore, in our experiment, we use the OJClone dataset to train our model. This dataset is an open source dataset containing many OJ questions, and each code fragment in the dataset is submitted by a different person. This means that the diversity of code semantics in the data set is rich, which is an important feature of real-world projects. Therefore, we choose this data set to solve the problem of bias in the data set.

## VI. Conclusions and Future Work

Detectiong Type-3/4 clone codes is important in many software engineering tasks and is crucial to the quality of software systems. In this paper, we propose an event embedding based method, EDAM, to detect semantic code clone. We treat the program as a series of continuous interdependent events and propose a novel approach to model the execution semantics of statements by using event embedding and event dependency. In this way, both the execution semantics of statements and their dependency infomation are captured to detect code fragments that are similar in semantics but different in syntax. Experimental results show that our model outperforms existing models in terms of Precision, Recall and F1 Score. In addition, we demonstrate that the integration of two implementation options help to improve the performance of our code clone detection model. We also conduct a visualization to investigate the insights of our solution.

Currently, our model supports C programs only. We plan to extend the model to support Java and other languages. Also, we plan to conduct more experiments to compare our model with other models using different dataset.

## References

[1] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, “A systematic review on code clone detection,” *IEEE Access*, vol. 7, pp. 86 121–86 144, 2019.

[2] C. K. Roy and J. R. Cordy, “A survey on software clone detection research,” *SCHOOL OF COMPUTING TR 2007-541, QUEEN’S UNIVERSITY*, vol. 115, 2007.

[3] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, “Towards a big data curated benchmark of inter-project code clones,” in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 476–480.

[4] S. Kim and H. Lee, “Software systems at risk: An empirical study of cloned vulnerabilities in practice,” *Computers & Security*, vol. 77, pp. 720–736, 02 2018.

[5] S. Kim, S. Woo, H. Lee, and H. Oh, “Vuddy: A scalable approach for vulnerable code clone discovery,” in *2017 IEEE Symposium on Security and Privacy (SP)*, 05 2017, pp. 595–614.

[6] S. Ducasse, M. Rieger, and S. Demeyer, “A language independent approach for detecting duplicated code,” *Conference on Software Maintenance*, pp. 109–118, 12 2000.

[7] J. Johnson, “Identifying redundancy in source code using fingerprints,” in *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, 01 1993, pp. 171–183.

[8] Johnson, “Substring matching for clone detection and change tracking,” in *Proceedings 1994 International Conference on Software Maintenance*, 10 1994, pp. 120–126.

[9] C. Ragkhitwetsagul and J. Krinke, “Using compilation/decompilation to enhance clone detection,” in *2017 IEEE 11th International Workshop on Software Clones (IWSC)*, 02 2017, pp. 1–7.

[10] M. Nishi and K. Damevski, “Scalable code clone detection and search based on adaptive prefix filtering,” *Journal of Systems and Software*, vol. 137, pp. 130–142, 11 2017.

[11] B. Baker, “A program for identifying duplicated code,” *Proceedings of Computing Science and Statistics: 24th Symposium on the Interface*, vol. 24, 07 1992.

[12] T. Kamiya, S. Kusumoto, and K. Inoue, “Ccfinder: A multilinguistic token-based code clone detection system for large scale source code,” *Software Engineering, IEEE Transactions on*, vol. 28, pp. 654–670, 08 2002.

[13] M. Tsunoda, Y. Kamei, and A. Sawada, “Assessing the differences of clone detection methods used in the fault-prone module prediction,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 03 2016, pp. 15–16.

[14] J. Svajlenko and C. Roy, “Fast and flexible large-scale clone detection with cloneworks,” in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 05 2017, pp. 27–30.

[15] R. Koschke, R. Falke, and P. Frenzel, “Clone detection using abstract syntax suffix trees,” in *2006 13th Working Conference on Reverse Engineering*, 10 2006, pp. 253–262.

[16] S. Chodarev, E. Pietrikova, and J. Kollar, “Haskell clone detection using pattern comparing algorithm,” in *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*, 07 2015, pp. 1–4.

[17] J. Krinke, “Identifying similar code with program dependence graphs,” in *Proc. of 8th Working Conference on Reverse Engineering, 2001*, 2001, pp. 301–309.

[18] M. Wang, P. Wang, and Y. Xu, “Ccsharp: An efficient three-phase code clone detector using modified pdgs,” in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, 12 2017, pp. 100–109.

[19] B. Li, C. Ye, S. Guan, and H. Zhou, “Semantic code clone detection via event embedding tree and gat network,” in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2020, pp. 382–393.

[20] H. Sajnani, V. Saini, J. Svajlenko, C. Roy, and C. Lopes, “Sourcerercc: scaling code clone detection to big-code,” in *Proceedings of the 38th International Conference on Software Engineering*, 05 2016, pp. 1157–1168.

[21] C. Roy and J. Cordy, “Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization,” in *IEEE International Conference on Program Comprehension*, 07 2008, pp. 172–181.

[22] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, “Cclearner: A deep learning-based clone detection approach,” in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 249–260.

[23] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “Deckard: scalable and accurate tree-based detection of code clones,” in *29th International Conference on Software Engineering (ICSE’07)*, 06 2007, pp. 96–105.

[24] M. Gabel, L. Jiang, and Z. Su, “Scalable detection of semantic clones,” in *Proceedings of the 30th International Conference on Software Engineering*, 01 2008, pp. 321–330.

[25] L. Jiang and Z. Su, “Automatic mining of functionally equivalent code fragments via random testing,” in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, 01 2009, pp. 81–92.

[26] W. Huihui and M. Li, “Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code,” in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 08 2017, pp. 3034–3040.

[27] G. Zhao and J. Huang, “Deepsim: deep learning code functional similarity,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 10 2018, pp. 141–151.

[28] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, “Oreo: Detection of clones in the twilight zone,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, 2018, p. 354–365.