Article
Discovery of Business Process Models from Incomplete Logs
Lili Wang 1,2, Xianwen Fang 1,2,* and Chifeng Shao 1

1 College of Mathematics and Big Data, Anhui University of Science and Technology, Huainan 232001, China
2 Anhui Province Engineering Laboratory for Big Data Analysis and Early Warning Technology of Coal Mine Safety, Huainan 232001, China
* Correspondence: xwfang@aust.edu.cn

Abstract: The completeness of event logs and long-distance dependencies are two major challenges for process mining. Until now, most process mining methods have not been able to discover long-distance dependency and assume that the directly-follows relationship in the log is complete. However, due to the existence of high concurrency and the cycle, it is difficult to guarantee that the real-life log is complete regarding the directly-follows relationship. Therefore, process mining needs to be able to deal with incompleteness. In this paper, we propose a method for discovering process models including sequential, exclusive, concurrent, and cyclic structures from incomplete event logs. The method analyzes the co-occurrence class of the log and the model and then uses the technology of combining the behavior profile and co-occurrence class to obtain the communication behavior profile of the co-occurrence class. Furthermore, a method of constructing a substructure from the event log using the co-occurrence class is presented. Finally, the whole process model is built by combining those substructures. The experimental results show that the proposed method can discover process models with complex structures involving cycles from incomplete event logs and also can deal with long-distance dependency in the event log. Meanwhile, the discovered process model has a good degree of consistency with the original model.

Keywords: Petri net; incomplete event log; long-distance dependency; behavior profile; degree of consistency

1. Introduction
The operation of an information system will generate a large number of event logs. These event logs record what activities will occur and in what order they occur in the business process. In addition, there is much other related information, such as executors and execution resources [1]. Process mining extracts valuable insights about processes from these event logs to analyze, monitor, and improve actual business processes. According to van der Aalst [2], process mining techniques can be divided into three directions: (1) process discovery, which uses event logs to generate process models; (2) conformance checking, which compares the behaviors generated by the model with the behaviors in the event log. The aim is to detect, locate, and explain deviations and to measure the severity of these deviations; and (3) process enhancement, which aims to improve or extend an existing process model using information about the actual process recorded in some event log. To avoid “spaghetti-like” process models, process discovery usually consists of three steps: first, preprocessing the noise behavior in the event log, then discovering the process model from the preprocessed event log, and finally evaluating the quality of the process model [3]. This paper does not consider log preprocessing steps and only focuses on how to discover process models from event logs.

In addition to noise, the real-life event log is often incomplete, that is, it is impossible to show all possible behaviors [4]. Most of the existing process mining algorithms assume that the log is weakly complete, i.e., they assume that the direct successor relationship is complete, the causal dependency is complete, or that the log is complete [5]. Obviously,
the assumption that the log is complete is impossible in reality. One of the factors leading to incomplete logs is high concurrency or a cycle. In [6], the authors proposed the concept of a conjoint occurrence class, which solved the incompleteness caused by high concurrency and illustrated how to discover process models from incomplete logs. However, this method has difficulty coping with cycles and does not consider the relationship between the model-based conjoint occurrence class and the log-based conjoint occurrence class.

Based on [6], we further provide a novel concept of a model-based co-occurrence class to capture activities that are always occur together in any executable trace of the model. Meanwhile, the relationship between the model-based co-occurrence class and log-based co-occurrence class is analyzed. Then, how to construct a cyclic structure containing multiple types of substructures is elaborated besides sequential, exclusive, and concurrent structures. Finally, a method of discovering process models from incomplete event logs using the technology of combining the co-occurrence class and behavior profile is presented. The proposed method in this paper solves the problem of the long-distance dependencies and incompleteness caused by the cycle or high concurrency. The experimental results show that when the log is incomplete, the proposed method can still build a process model that has a high degree of consistency with the original model.

In the following, Section 2 first discusses related work. The problem statement is given in Section 3. Section 4 explains some basic knowledge about the co-occurrence class of the log and model. In Section 5, the method of constructing a substructure using the co-occurrence is described. In Section 6, we report on experimental results. Conclusions and future work are presented in Section 7.

2. Related Work

Until now, a series of process mining methods have been proposed, mainly including a directly-follows relationship between activities [7], domain-based theory [8–11], genetic-based algorithm [12,13], frequency-based technology [14–16], cut-based technology [17], abstraction-based technology [18], and so on. Of these, the method based on the directly-follows relationship between activities can only find some subclasses of Petri nets and cannot guarantee soundness and fitness, such as the \( \alpha \) algorithm [19] and its derivatives [20–22]. The technique of domain-based theory mainly includes language-based domain miners [8], state-based domain miners [9], and integer linear programming (ILP) miners [10,11]. Although these methods guarantee the fitness of the model, they cannot guarantee the soundness of the model. The mining method based on the genetic algorithm iterates the process model until the acceptable quality is obtained. It selects the fittest individuals and generates new individuals using two genetic operators, such as crossover and mutation [12,13]. Although the process model obtained by this method has good fitness, it consumes a great deal of running time and cannot guarantee the soundness of the model.

The frequency-based method takes frequencies of events and sequences into account when constructing a process model, such as the Heuristic Mining (HM) algorithm [14], flexible heuristics miner (FHM) [15], or Fodina [16]. It calculates the dependency measure between activities and filters noise behavior by removing causal dependencies below a given threshold. This technique sometimes generates a process model with incorrect behaviors; meanwhile, soundness and fitness are not guaranteed. Cut-based technology, such as the Inductive Miner (IM) [17] and Inductive Miner-infrequent (IMi) algorithms [22], adopts the idea of divide and conquer and uses four cuts to recursively split the log into several sub-logs until it cannot find a cut. Subsequently, it constructs the sub-model from each sub-log. Finally, a block-structured process model can be built by merging all sub-models. Although the soundness of the process model is guaranteed, its precision is very low, that is, the discovered process model is over-generalized. Process discovery algorithms based on abstract techniques, such as fuzzy miner [18], produce models without executable semantics, so neither soundness nor fitness is guaranteed. In [23], state-of-the-art process discovery methods based on the split gateway discovery were presented. Split Miner
combines a novel method to filter directly-follows graphs induced by event logs, with a method for identifying combinations of split gateways that accurately capture concurrency, conflict, and causal relations between neighbors in directly-follows graphs. Split Miner is the first method to guarantee that the discovered BPMN model is deadlock-free and is not restricted to producing a block-structured model. However, when the number of activities in the log is large, the time complexity of the algorithm is high.

At present, most process discovery methods can deal with noises [24–28]. Apart from noise, another problem of the real event log is incompleteness [29]. Since the event log only records the observed behaviors of the business process during its past, it cannot capture the behaviors that will occur in the future. Hence, the event log is usually incomplete. However, few algorithms have the capability of handling the incompleteness of event logs. In [17], based on the IM algorithm, a process discovery algorithm handling incomplete event logs was proposed, which introduced probabilistic behavioral relations that are less sensitive to incompleteness. It adopted the divide-and-conquer approach in the same way as IM and searched for an optimal partition of activities by estimating probabilities of the relations between activities. However, the method proposed in [17] has difficulty in handling long-distance dependency and cannot handle incompleteness caused by high concurrency well. In [6], the authors presented conjoint occurrence classes to handle the incompleteness caused by the concurrency and cycles. This method can infer the causal and concurrent behavior not exhibited in the log. However, the method in [6] cannot handle loop structures and the long-distance dependency hidden in the log. In this work, a co-occurrence class based-log and based-model are provided, respectively, to solve the incompleteness and long-distance dependency; then, we propose the method of discovering a process model with multiple different structures including cycles.

3. Problem Statement

At present, most process discovery methods cannot discover indirect dependencies between activities, such as long-distance dependencies [15] (such as S1 and S2 in Figure 1). The behavior profile uses the weak order relationship between activities to describe the behavior dependency relationship among activities, involving strict order relation, exclusiveness relation, and interleaving order relation [30]. Among them, the exclusiveness relation is the strictest relationship, indicating that two activities cannot appear in any trace at the same time. However, the behavior profile only describes the behavior dependency between activities and cannot capture the actual occurrence dependency and causal dependency of activities in the actual business process, i.e., whether the occurrence of one activity will affect the occurrence of another activity and vice versa.

![Figure 1. A Petri net with long-distance dependencies.](image)

The strict order relationship only indicates that there is a sequence constraint between activities, but it cannot illustrate whether an occurrence dependency relationship between activities exists. As shown in Figure 2, activity A and activity B are both in a strict order
relationship. From Figure 2 and Table 1, we can know the strict order relationship only indicates that if activities A and B occur at the same time, then activity A must occur before activity B. Hence, the sequence constraint of activity A and activity B is captured, while the actual occurrence dependency between activity A and activity B cannot be captured.

![Figure 2](image)

**Figure 2.** An example of the strict order relation.

<table>
<thead>
<tr>
<th>Behavior Profile</th>
<th>Figure No.</th>
<th>Executable Sequence</th>
<th>Order of Occurrence</th>
<th>Occurrence Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>strict order</td>
<td>Figure 2a</td>
<td>{A,B,AB}</td>
<td>A occurs before B</td>
<td>none</td>
</tr>
<tr>
<td>A → B</td>
<td>Figure 2b</td>
<td>{AC,AB}</td>
<td>A occurs before B</td>
<td>Once activity B occurs, activity A must occur, but not vice versa</td>
</tr>
<tr>
<td></td>
<td>Figure 2c</td>
<td>{AB,CB}</td>
<td>A occurs before B</td>
<td>Once activity A occurs, activity B must occur, but not vice versa</td>
</tr>
</tbody>
</table>

The interleaving order relationship only indicates that there are no occurrence order constraints between activities, but it does not mandate that activities must be enabled at the same time. For example, activity A and activity B are in an interleaving order relationship in Figure 3. However, only activity A and activity B in Figure 3a are concurrent, which is also the most typical structural feature corresponding to the interleaving order relationship. Table 2 shows the occurrence order and occurrence dependency of activities A and B as shown in Figure 3. Table 2 indicates that the occurrence dependency constraint between activities is difficult to capture only according to the interleaving order relationship.

![Figure 3](image)

**Figure 3.** An example of the interleaving order.
Table 2. The relationship between activity A and activity B shown in Figure 3.

<table>
<thead>
<tr>
<th>Behavior Profile</th>
<th>Figure No.</th>
<th>Executable Sequence</th>
<th>Order of Occurrence</th>
<th>Occurrence Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interleaving order</td>
<td>Figure 3a</td>
<td>{CABD,CBAD}</td>
<td>No ordering constraint</td>
<td>A and B occur at most once and are in a co-occurrence relationship</td>
</tr>
<tr>
<td></td>
<td>Figure 3b</td>
<td>{CAE,DBE,CAE,DBE,DBAE}</td>
<td>No ordering constraint</td>
<td>A and B occur at most once, but do not have any occurrence dependencies</td>
</tr>
<tr>
<td></td>
<td>Figure 3c</td>
<td>{CAD,CBD,CAAD,CBBD,CABD,CBACABCD,}</td>
<td>No ordering constraint</td>
<td>A and B occur infinite times and do not have any occurrence dependencies</td>
</tr>
<tr>
<td></td>
<td>Figure 3d</td>
<td>{ABC,ABABC,ABABABC,}</td>
<td>No ordering constraint</td>
<td>A and B occur infinite times, but there is a certain occurrence dependency, such as A and B occur the same number of times, and activity A must occur before B</td>
</tr>
</tbody>
</table>

Another problem is that *the event log represents the behavior that has been observed during the actual operation of the system and cannot contain all possible behaviors of the system. Thus, the log is often incomplete.* One of the problems faced by process discovery is that the log cannot contain enough information to discover a process model, i.e., the log might be incomplete [29]. This might cause the discovered process model to exclude the actual behaviors or contain some additional behaviors that are not in the real process. The existing process mining usually assumes that either the activities are complete or the directly-follows relationships between activities are complete. However, when there exist high concurrency behaviors, the second assumption is often unrealistic. For example, a process consists of 10 concurrent activities, and the log consists of about 10,000 cases. There are \(10! = 3,628,800\) possible sequences caused by concurrent behavior. Obviously, the number of cases in the log is far less than the actual possible sequence, so it is impossible that the log samples all possible directly-follows relationships between activities. There are many factors leading to logs being possibly incomplete. However, cycles and high concurrency are important factors. *When the number of activities on concurrent branches is large, it is impossible for the event log to sample all possible sequences of activities.* Figure 4a shows an initial process model \(M_0\) of each concurrent branch containing two activities. An incomplete event log \(L_1\) is generated by the model \(M_0\), where \(L_1 = \{<\mathrm{abcd}\>, <\mathrm{acbd}\>, <\mathrm{xcabd}\>, <\mathrm{xcab}\>\}.\) The process model obtained by applying the IM algorithm and the HM algorithm to the event log is shown in Figure 4b,c. The precision of the model \(M_1\) is 1, and the fitness is low. The fitness and precision of the model \(M_2\) are both 1, but this model is overfitting. Therefore, incomplete logs caused by concurrency will lead to low-quality process models or even erroneous results.

Figure 4. Process mining results of incomplete logs caused by high concurrency.
To solve the above problems, a method for discovering the process model from incomplete logs is presented. We adopt the technology of combining behavioral profiles and co-occurrence classes to analyze the structural characteristics and communication behavior relationships of co-occurrence classes, then merge several co-occurrence classes into a larger substructure to find the process model. The research framework of the proposed method is shown in Figure 5. The proposed method solves the problem that behavior profiles cannot capture the occurrence dependency between activities and the incompleteness caused by high concurrency. The experimental results show that the proposed method can find long-distance dependency, and the discovered process model has a good consistency with the original model.

![Figure 5. Framework of the proposed method.](image)

4. Construction Method of Co-Occurrence Class

In the actual business process, some activities may be mandatory nodes of the whole process, and the occurrence of some activities will lead to the subsequent occurrence of other activities, or the occurrence of some activities can be accompanied by the occurrence of some activities that precede it. Obviously, in addition to the constraints on the order of occurrence, there also exist occurrence dependencies between activities. Therefore, this section analyzes the invariant predecessor set and invariant successor set of activities to capture the co-occurrence invariant set of activities in the log or model. Activities are divided into several subsets with co-occurrence dependency in the log or model through the co-occurrence classes. Section 4.1 presents the method of discovering co-occurrence classes from the event log and analyzes the characteristics of co-occurrence classes. Section 4.2 provides the method of discovering co-occurrence classes from the model.

4.1. Discovery of Co-Occurrence Classes from Logs

The event log shows the actual execution of activities in the business process. By analyzing the occurrence dependencies between activities in each trace, the occurrence dependency between activities in the log can be found. In the following, the relevant definitions of trace handling operations are given.

**Definition 1** (Trace handling operations [6]). Let $L$ be an event log, $A_L$ is an activity set over $L$. For $\forall a \in A_L$, if there exists a trace $\sigma \in L$, such that $a \in \sigma$, where $|\sigma| = n$:

1. $\sigma[k] = a$ indicates that the $k$-th element in the trace $\sigma$ is an activity $a$;
2. $\text{Pos}(a, \sigma) = \{p, q, r, \cdots, s\}$ such that $\sigma[p] = a, \cdots, \sigma[s] = a$;
3. The direct predecessor activity of the $i$-th activity in the trace $\sigma$ is defined as $\text{Pred}(i, \sigma) = \sigma[i - 1], i \in \{2, 3, \cdots, n\}$;
4. The direct successor activity of the $i$-th activity in the trace $\sigma$ is defined as $\text{Succ}(i, \sigma) = \sigma[i + 1], i \in \{1, 2, \cdots, n - 1\}$;
(5) Invariant set of activities that appear before activity \(a\) in trace \(\sigma\) is denoted as \(\text{Predset}(a, \sigma)\),

\[
\text{Predset}(a, \sigma) = \begin{cases} 
\phi \text{ if } \text{Pos}(a, \sigma) = 1 \\
\bigcup_{i=q+2}^{p} \text{Pred}(i, \sigma) \cap \bigcup_{i=q+2}^{r} \text{Pred}(i, \sigma) \cap \bigcup_{i=r+2}^{s} \text{Pred}(i, \sigma) & \text{if } \text{Pos}(a, \sigma) = \{p, \ldots, q, r, s\}, \text{where } p < \cdots < q < r < s
\end{cases}
\] (1)

(6) Invariant set of activities that appear after activity \(a\) in trace \(\sigma\) is denoted as \(\text{Succset}(a, \sigma)\),

\[
\text{Succset}(a, \sigma) = \begin{cases} 
\phi \text{ if } \text{Pos}(a, \sigma) = n \\
\bigcup_{i=q}^{n-2} \text{Succ}(i, \sigma) \cap \bigcup_{i=q}^{r-2} \text{Succ}(i, \sigma) \cap \bigcup_{i=s}^{r-1} \text{Succ}(i, \sigma) & \text{if } \text{Pos}(a, \sigma) = \{p, q, r, \ldots, s\}, \text{where } p < q < r < \cdots < s
\end{cases}
\] (2)

For example, considering a trace \(\sigma = \{a, b, c, d, c, f\}\), using Definition 1, we obtain \(\sigma[3] = c, \text{Pos}(b, \sigma) = \{2, 5\}, \text{Pred}(4, \sigma) = \sigma[3] = c, \text{Succ}(4, \sigma) = \sigma[5] = b, \text{Predset}(a, \sigma) = \emptyset, \text{Predset}(b, \sigma) = \{a\} \cap \{c, d\} = \emptyset, \text{Succset}(a, \sigma) = \{b, c, d, e, f\}, \text{Succset}(b, \sigma) = \{c, d, e, f\} \cap \{c, d\} = \{c, d\} \).

\(\text{Predset}(a, \sigma)\) consists of activities appearing between two activities \(a\) together and before the first occurrence of the activity \(a\). \(\text{Succset}(a, \sigma)\) consists of activities appearing between two activities \(a\) together and after the last occurrence of the activity \(a\). Therefore, \(\text{Predset}(a, \sigma)\) and \(\text{Succset}(a, \sigma)\) denote the set of possible activities that must occur before the activity \(a\) and after the activity \(a\) in any trace \(\sigma\), respectively. To capture the predecessor and successor activities that have co-occurrence dependencies with an activity in the log, Definition 2 gives the invariant predecessor set and the invariant successor set of the activity in the log.

**Definition 2** (Log Invariant predecessor set, Invariant successor set). Let \(L\) be an event log, \(A_L\) is an activity set over \(L\). For every \(a \in A_L\), invariant predecessor set of activity \(a\) is defined as \(\text{InvPredS}(a, L) = \bigcap_{\sigma \in L} \text{Predset}(a, \sigma)\), and invariant successor set of activity \(a\) is defined as \(\text{InvSuccS}(a, L) = \bigcap_{\sigma \in L} \text{Succset}(a, \sigma)\).

The invariant predecessor set in Definition 2 represents the set of activities that always occur before the activity \(a\) in any trace of the log. Similarly, the invariant successor set represents the activity set that always occurs after the activity \(a\) in any trace of the log. The activities in the invariant predecessor set or the invariant successor set not only have a strict order relationship with the activity \(a\) but also have a co-occurrence relationship with the activity \(a\). Definition 3 further gives the concept of a co-occurrence invariant set based on Definition 2.

**Definition 3** (Log Co-occurrence invariant set). Let \(L\) be an event log, \(A_L\) is an activity set over \(L\). For every \(a \in A_L\), co-occurrence invariant set of activities \(a\) is denoted as \(\text{Oi}(a, L) = \text{InvPredS}(a, L) \cup \text{InvSuccS}(a, L) \cup \{a\}\).

In Definition 3, \(\text{Oi}(a, L)\) presents a set of activities that occur simultaneously with the activities \(a\) in any trace of the log \(L\). There is not only a behavior dependency but also a co-occurrence dependency between them.

For example, given an activity log \(L_2\) as shown in Table 3, the process model shown in Figure 6 is the original model of the log \(L_2\). This paper takes the event log \(L_2\) as an example to illustrate the solution process of our proposed method.
Table 3. An event log $L_2$.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Numbers</th>
<th>Sequence of Activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma_1$</td>
<td>5</td>
<td>abcfgijkmq</td>
</tr>
<tr>
<td>$\sigma_2$</td>
<td>4</td>
<td>abcfgijkmq</td>
</tr>
<tr>
<td>$\sigma_3$</td>
<td>5</td>
<td>adfgijkmq</td>
</tr>
<tr>
<td>$\sigma_4$</td>
<td>6</td>
<td>adfgijkmnnor</td>
</tr>
<tr>
<td>$\sigma_5$</td>
<td>7</td>
<td>adfgijkmnopnor</td>
</tr>
<tr>
<td>$\sigma_6$</td>
<td>8</td>
<td>abcfgijkmnopnor</td>
</tr>
<tr>
<td>$\sigma_7$</td>
<td>6</td>
<td>afghjkbcmq</td>
</tr>
<tr>
<td>$\sigma_8$</td>
<td>5</td>
<td>afghjkdemq</td>
</tr>
<tr>
<td>$\sigma_9$</td>
<td>4</td>
<td>afghjkmnopnor</td>
</tr>
<tr>
<td>$\sigma_{10}$</td>
<td>3</td>
<td>afghjkmnopnor</td>
</tr>
</tbody>
</table>

Figure 6. The original model of log $L_2$.

Table 4 presents the invariant predecessor set, invariant successor set, and co-occurrence invariant set for each activity in the event log $L_2$.

Table 4. Invariant set of all activities in the log $L_2$.

<table>
<thead>
<tr>
<th>Activity</th>
<th>InvPredS</th>
<th>InvSuccS</th>
<th>Oi</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$\emptyset$</td>
<td>{f,g,i,h,j,k,m}</td>
<td>[a,f,g,i,h,j,k,m]</td>
</tr>
<tr>
<td>b</td>
<td>[a]</td>
<td>[c,m]</td>
<td>[a,b,c,m]</td>
</tr>
<tr>
<td>c</td>
<td>[a,b]</td>
<td>[m]</td>
<td>[a,b,c,m]</td>
</tr>
<tr>
<td>d</td>
<td>[a]</td>
<td>[e,m]</td>
<td>[a,d,e,m]</td>
</tr>
<tr>
<td>e</td>
<td>[a,d]</td>
<td>[m]</td>
<td>[a,d,e,m]</td>
</tr>
<tr>
<td>f</td>
<td>[a]</td>
<td>[g,l,h,j,k,m]</td>
<td>[a,f,g,i,h,j,k,m]</td>
</tr>
<tr>
<td>g</td>
<td>[a,f]</td>
<td>[i,k,m]</td>
<td>[a,f,g,i,k,m]</td>
</tr>
<tr>
<td>h</td>
<td>[a,f]</td>
<td>[j,k,m]</td>
<td>[a,f,g,i,j,k,m]</td>
</tr>
<tr>
<td>i</td>
<td>[a,f,g]</td>
<td>[k,m]</td>
<td>[a,f,g,i,k,m]</td>
</tr>
<tr>
<td>j</td>
<td>[a,f,h]</td>
<td>[k,m]</td>
<td>[a,f,h,i,j,k,m]</td>
</tr>
<tr>
<td>k</td>
<td>[a,f,g,i,h,j]</td>
<td>$\emptyset$</td>
<td>$\emptyset$ a,f,g,i,h,j,k,m</td>
</tr>
<tr>
<td>m</td>
<td>[a,f,g,i,h,j,k]</td>
<td>$\emptyset$</td>
<td>$\emptyset$ a,f,g,i,h,j,k,m</td>
</tr>
<tr>
<td>n</td>
<td>$\emptyset$</td>
<td>$\emptyset$ o</td>
<td>$\emptyset$ a,f,g,i,h,j,k,m</td>
</tr>
<tr>
<td>o</td>
<td>[n]</td>
<td>$\emptyset$</td>
<td>$\emptyset$ a,f,g,i,h,j,k,m</td>
</tr>
<tr>
<td>p</td>
<td>[n,o]</td>
<td>$\emptyset$</td>
<td>$\emptyset$ a,f,g,i,h,j,k,m</td>
</tr>
<tr>
<td>q</td>
<td>[a,f,g,h,i,j,k,m]</td>
<td>$\emptyset$</td>
<td>$\emptyset$ a,f,g,i,h,j,k,m</td>
</tr>
<tr>
<td>r</td>
<td>[a,f,g,h,i,j,k,m,n,o]</td>
<td>$\emptyset$</td>
<td>$\emptyset$ a,f,g,i,h,j,k,m,n,o</td>
</tr>
</tbody>
</table>

Since the co-occurrence invariant set represents the set of activities that appear simultaneously with the current activity in all traces of the log, based on this, this paper adopts the co-occurrence invariant set to capture the occurrence dependency between activities. To facilitate the discovery of a co-occurrence dependency among several activities, the activities in the log can be divided into several subsets with equivalent behavior by using the co-occurrence invariant set. Therefore, Definition 5 introduces the concept of co-occurrence classes.
**Definition 4** (Co-occurrence matrix). Let \( L \) be an event log, and \( A_L \) is an activity set over \( L \) and \(|A_L| = n\). Matrix \( A_{n \times n} \) is called as the co-occurrence matrix, where \( A_{ij} \) represents the co-occurrence relation between \( a_i, a_j \in A_L \), if \( a_i \in \text{Oi}(a_j, L) \); then, \( A_{ij} = 1 \).

**Definition 5** (Co-occurrence class). Let \( L \) be an event log, \( A_L \) is an activity set over \( L \). \( A_{n \times n} \) is a co-occurrence matrix of the log \( L \). \( a_p, a_q, a_r, \ldots, a_s \in A_L \), an activity set \( \text{COC} = \{a_p, a_q, a_r, \ldots, a_s\} \) is called a co-occurrence class if and only if \( A_{ij} = 1 \), where \( i = p, q, r, \ldots, s; j = p, q, r, \ldots, s \).

Definition 5 indicates that there exists a co-occurrence dependency between all activities in the co-occurrence class; that is, they always appear in a trace of the log together. Table 5 shows the co-occurrence relationship matrix of the event log.

**Table 5. Co-occurrence matrix for log \( L_2 \).**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>m</th>
<th>n</th>
<th>o</th>
<th>p</th>
<th>q</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>f</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>g</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>h</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>j</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>k</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>l</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>m</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>o</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>p</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>q</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>r</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Using Definition 5, we can get co-occurrence classes from Table 5: \( \text{COC} = \{\text{COC}_1, \text{COC}_2, \text{COC}_3, \text{COC}_4, \text{COC}_5, \text{COC}_6, \text{COC}_7, \text{COC}_8\} \), where \( \text{COC}_1 = \{b, c\} \), \( \text{COC}_2 = \{d, e\} \), \( \text{COC}_3 = \{a, m, f, g, i, k\} \), \( \text{COC}_4 = \{h, j\} \), \( \text{COC}_5 = \{n, o\} \), \( \text{COC}_6 = \{p\} \), \( \text{COC}_7 = \{q\} \), \( \text{COC}_8 = \{r\} \).

The co-occurrence class has some interesting properties. The relevant conclusions and proofs are given below.

**Property 1** (Co-occurrence). Let \( L \) be an event log, \( \text{COC}_k \) is a co-occurrence class got from event log \( L \). For \( \forall a_i \in \text{COC}_k \), if \( \exists \sigma \in L \) such that \( a_i \in \sigma \), then for \( \forall a_j \in \text{COC}_k \backslash a_i, a_j \in \sigma \) holds.

**Proof.** If \( \exists a_i \in \text{COC}_k \backslash a_i \) such that \( a_i \notin \sigma \), then \( a_i \notin \text{Predset}(a_i, \sigma) \) and \( a_i \notin \text{Succset}(a_i, \sigma) \). So \( a_i \notin \text{InvPredS}(a_i, L) \) and \( a_i \notin \text{InvSuccS}(a_i, L) \). Obviously, \( a_i \notin \text{Oi}(a_i, L) \). Therefore, \( a_i, a_j \) cannot both belong to \( \text{COC}_k \). There exists a contradiction.

**Property 2** (Invariant order). Let \( L \) be an event log, \( \text{COC}_k = \{a_p, a_q, a_r, \ldots, a_s\} \) is a co-occurrence class got from the event log \( L \). If \( \exists \sigma_i \in L \), activities \( a_p, a_q, a_r, \ldots, a_s \) satisfy partial order relation \( a_p \prec a_q \prec a_r \prec \cdots \prec a_s \), for \( \forall a_i \in \text{COC}_k \), \( \exists \sigma_i \in L \), if \( a_i \in \sigma_i \), then activities \( a_p, a_q, a_r, \ldots, a_s \) also satisfy partial order relation \( a_p \prec a_q \prec a_r \prec \cdots \prec a_s \) in a trace \( \sigma_i \). i.e., if activities in \( \text{COC}_k \) occurred in a trace, they should occur in the same order.

**Proof.** Assume there exist two activities that appear in two different traces in different orders, and let two activities be \( a_p, a_r, a_p \prec a_q \prec a_r \prec \cdots \prec a_s \) holds in trace \( \sigma_i \), but \( a_p \prec a_r \prec a_q \prec a_r \prec \cdots \prec a_s \) holds in trace \( \sigma_j \). Thus, \( \text{InvPredS}(a_q, L) = \{a_p, \ldots, a_p\} \cap \{\cdots, a_p, a_r\} \cap \cdots, \text{InvSuccS}(a_q, L) = \{a_r, a_p, \ldots, a_r\} \cap \{a_r, \cdots, a_s\} \cap \cdots \). Clearly,
\[ a_r \notin O_i(a_q, L). \] Therefore, it is impossible for \( a_q, a_r \) to belong to the same co-occurrence class at the same time. There exists a contradiction. \( \square \)

Algorithm 1, Algorithm 2, and Algorithm 3, respectively, show how to obtain the invariant predecessor set, invariant successor set, and co-occurrence class of any activity from the log.

---

**Algorithm 1:** An invariant predecessor set of any activity

**Function:** GetInvPredS(a, L)

**Input:** An event log L, a set of activity \( A_L \)

**Output:** An invariant predecessor set of any activity

```plaintext
1 foreach activity a in \( A_L \) do 
2     InvPredS(a, L) = \emptyset; / / initializes the invariant predecessor set; 
3     foreach trace \( \sigma \) in L do 
4         if (Pos(a, \sigma) = 1) then 
5             Predset(a, \sigma) = \emptyset; 
6             else Pos(a, \sigma) = \{p, \cdots, q, r, s\} ( where p < q < r < s); 
7             Predset(a, \sigma) = \bigcup_{i=p}^{q-2} \text{Pred}(i, \sigma) \cap \cdots \cap \bigcup_{i=q+2}^{r-2} \text{Pred}(i, \sigma); 
8         end 
9     end 
10    InvPredS(a, L) = InvPredS(a, L) \cap Predset(a, \sigma); 
11 if InvPredS(a, L) = \emptyset then 
12     return InvPredS(a, L); 
13 break; 
14 end 
15 end 
16 return InvPredS(a, L); 
```

---

**Algorithm 2:** An invariant successor set of any activity

**Function:** GetInvSuccS(a, L)

**Input:** An event log L, a set of activity \( A_L \)

**Output:** An invariant successor set of any activity

```plaintext
1 foreach activity a in \( A_L \) do 
2     InvSuccS(a, L) = \emptyset; / / initializes the invariant successor set; 
3     foreach each trace \( \sigma \) in L do 
4         if (Pos(a, \sigma) = n) then 
5             Succset(a, \sigma) = \emptyset; 
6             else Pos(a, \sigma) = \{p, \cdots, q, r, s\} ( where p < q < r < s); 
7             Succset(a, \sigma) = \bigcup_{i=p}^{q-2} \text{Succ}(i, \sigma) \cap \cdots \cap \bigcup_{i=q+2}^{r-1} \text{Succ}(i, \sigma); 
8         end 
9     end 
10    InvSuccS(a, L) = InvSuccS(a, L) \cap Succset(a, \sigma); 
11 if InvSuccS(a, L) = \emptyset then 
12     return InvSuccS(a, L); 
13 break; 
14 end 
15 end 
16 return InvSuccS(a, L); 
```
Algorithm 3: Getting co-occurrence classes from the event log

<table>
<thead>
<tr>
<th>Function: GetCOCSet(L, A_L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: An event log L, A set of activities A_L</td>
</tr>
<tr>
<td>Output: A set of co-occurrence classes</td>
</tr>
</tbody>
</table>

/* getting the invariant set of any activities */

1. foreach activity a in A_L do
   2. InvPredS(a, L) = GetInvPredS(a, L);
   3. InvSuccS(a, L) = GetInvSuccS(a, L);
   4. Oi(a) = InvPredS(a, L) \cup InvSuccS(a, L) \cup \{a\};
end

/* Constructing co-occurrence matrix M_{n \times n} */

6. N = |A_L|;
7. for i in N do
   8. for j in N do
      9. M[i][j] = 0; //initialize;
   10. end
  11. end
12. for i in N do
   13. for j in N do
      14. a = GetActivityName(i);
      15. b = GetActivityName(j);
      16. if b \in Oi(a) then
          17. M[i][j] = 1;
      18. end
   19. end
20. end

/* Construct Co-occurrence class */

21. COCSet = \phi; //initialize the set of co-occurrence class;
22. for i in N do
   23. COC_i = \{a_i\}; //initialize each activity to a co-occurrence class;
   24. COCSet = COCSet \cup COC_i;
25. end
26. for COC_i and COC_j in COCSet do
   27. foreach activity a in COC_i do
      28. k = GetActivityIndex (a);
      29. foreach activity b in COC_j do
         30. l = GetActivityIndex (b);
         31. if (M[k][l] = 1 && M[l][k] = 1) then
             32. COC_{ij} = COC_i \cup COC_j;
             33. COCSet = COCSet \cup COC_{ij} = COC_i \cup COC_j;
         34. end
      35. end
   36. end
37. end
38. return COCSet;

In Algorithm 1, Step 2 initializes the invariant predecessor set to an empty set, Steps 3–9 solve the invariant predecessor set of the activity in each trace, Steps 10–15 get the invariant predecessor set of the activity in the log, and Steps 11–14 indicate that once the invariant predecessor set becomes an empty set, the algorithm is stopped.

In Algorithm 2, Step 2 initializes the invariant successor set to an empty set, Steps 3–9 solve the invariant successor set of the activity in each trace, Steps 10–15 obtain the invariant
successor set of the activity in the log, and Steps 11–14 indicate that once the invariant successor activity set becomes an empty set, the algorithm is stopped.

Steps 1–5 in Algorithm 3 are used to find the invariant set of any activity, Steps 7–11 initialize each element in the co-occurrence relationship matrix to 0, Steps 12–20 update the value of the co-occurrence relationship matrix according to the invariant activity set, Steps 22–25 set each activity to an initial co-occurrence class, and Steps 21–37 construct all possible co-occurrence classes by merging the initial co-occurrence classes according to the co-occurrence relationship matrix.

### 4.2. Discovering Co-Occurrence Classes from Models

The co-occurrence invariant relationship of activities in the log is closely related to their structural relationship in the model. To facilitate the discovery of their structural relationship in the model according to the co-occurrence invariant relationship of activities in the log, we present some definitions related to the invariance of activity occurring in the model.

**Definition 6** (Execution sequence). Execution sequence $\epsilon_p$ of the process model $N = (A_N, A_o, C, F, T)$ is defined as an executable sequence of the form $\{a_i\} \cdot A^* \cdot \{a_j\}$ from the start activity to the end activity of the process model. All execution sequences of the process model $N$ are denoted as $T(N)$.

**Definition 7** (Model Invariant predecessor set, Invariant successor set). Let $N$ be a process model, $A_N$ be an activity set of model, and $T(N)$ be a set of executable sequences of the model. For $a \in A_N$, the invariant predecessor set of activity $a$ is defined as $InvPredS(a, L) = \bigcap_{\sigma \in T(N)} Predset(a, \sigma)$, and the invariant successor set of activity $a$ is defined as $InvSuccS(a, L) = \bigcap_{\sigma \in T(N)} Succset(a, \sigma)$.

The set of invariant predecessors of activity $a$ in model $N$ represents the set of activities that occur before $a$ in all executable sequences of the model $N$. Similarly, the set of invariant successors of activity $a$ in the model represents the set of activities that occur after $a$ in all executable sequences of the model $N$.

**Definition 8** (Model co-occurrence invariant set). Let $N$ be a process model and $A_N$ be an activity set of models. For $a \in A_N$, the co-occurrence invariant set of activity $a$ is defined as $Oi(a, N) = InvPredS(a, N) \cup InvSuccS(a, N) \cup \{a\}$.

Definition 8 denotes $Oi(a, N)$, which represents the set of activities that always occur simultaneously with the activity $a$ in any execution sequence of the model.

**Definition 9** (Model co-occurrence class). Let $N$ be a process model and $A_N$ be an activity set of the model, where $|A_N| = n$, $A_{n \times n}$ is the co-occurrence relationship matrix of the model $N$. For $a_p, a_q, a_r, \ldots, a_s \in A_N$, a set of activities $COC = \{a_p, a_q, a_r, \ldots, a_s\}$ is called as a co-occurrence class if and only if $A_{ij} = 1$, where $i = p, q, r, \ldots, s, j = p, q, r, \ldots, s$.

Clearly, Definition 9 denotes $\forall a_i, a_j \in COC$, if $a_i \in Oi(a_j, N)$, then $a_j \in Oi(a_i, N)$, i.e., activities in a co-occurrence always occur simultaneously in an execution sequence.

According to Definition 9, Table 6 gives the invariant predecessor set, invariant successor set, and co-occurrence invariant set of each activity in the four basic structures (as shown in Figure 7). Table 7 presents co-occurrence classes of the four basic structures obtained from Table 6.

Table 7 shows that the exclusive structure and the cyclic structure have the same set of co-occurrence classes, the sequential structure has only one co-occurrence class that contains all activities, and the concurrent structure is decomposed into two co-occurrence classes according to the concurrent branch, one of which contains the and-gateway activities.
Table 6. Invariant set of activities in four basic structures.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Sequential Structure</th>
<th>Exclusive Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>InvPredS</td>
<td>InvSuccS</td>
</tr>
<tr>
<td>x</td>
<td>∅</td>
<td>[a,b,c,d,y]</td>
</tr>
<tr>
<td>a</td>
<td>{x}</td>
<td>[b,c,d,y]</td>
</tr>
<tr>
<td>b</td>
<td>{x,a}</td>
<td>[c,d,y]</td>
</tr>
<tr>
<td>c</td>
<td>{x,a,b}</td>
<td>[d,y]</td>
</tr>
<tr>
<td>d</td>
<td>{x,a,b,c}</td>
<td>[y]</td>
</tr>
<tr>
<td>y</td>
<td>{x,a,b,c,d}</td>
<td>∅</td>
</tr>
</tbody>
</table>

Table 7. Co-occurrence classes of the four basic structures.

<table>
<thead>
<tr>
<th>Structure</th>
<th>Sequential</th>
<th>Exclusive</th>
<th>Concurrent</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>co-occurrence class</td>
<td>[x,a,b,c,d,y]</td>
<td>[x,y],[a,b],[c,d]</td>
<td>[x,a,b,y],[c,d]</td>
<td>[x,y],[a,b],[c,d]</td>
</tr>
</tbody>
</table>

5. Building Substructures Using Log Co-Occurrence Classes

Section 4 presents the solution method for the co-occurrence invariant set of any activity in the log or model; then, the co-occurrence class is provided. This section further illustrates how to leverage the co-occurrence classes in the event log to build a model...
substructure that includes the sequence, conflict, concurrency, and cycle. Section 5.1 presents the method of constructing the basic substructure from a single co-occurrence class, and Section 5.2 addresses the communication behavior relationship between the co-occurrence classes and how to combine multiple co-occurrence classes into larger substructures.

5.1. Building the Basic Substructure from a Single Co-Occurrence Class

Properties 1 and 2 indicate that although activities in the co-occurrence class always occur simultaneously in the same order, the structural features of activities in the co-occurrence class are unknown. Theorems 1, 2, and 3, respectively, give their corresponding substructures according to the behavioral relationship of activities in the co-occurrence class in the log.

**Theorem 1** (A substructure corresponding to the co-occurrence class with a directly-follows relationship). Let \( L \) be an event log and \( \text{COC}_k = \{a_{k_1}, a_{k_2}, \ldots, a_{k_p}\} \), \( \forall \sigma \in L, \) if \( \sigma[i] = a_{k_1}, \sigma[i+1] = a_{k_2}, \ldots, \sigma[i+p] = a_{k_p} \) holds, then \( a_{k_1}, a_{k_2}, \ldots, a_{k_p} \) form a sequential substructure.

**Proof.** We assume activities \( a_{k_1}, a_{k_2}, \ldots, a_{k_p} \) constitute non-sequential substructures, i.e., these activities can constitute the substructure of a cycle, concurrency, or conflict except sequence. From the characteristics of a cycle, concurrency, and conflict substructure, it is known that activities \( a_{k_1}, a_{k_2}, \ldots, a_{k_p} \) cannot belong to the same co-occurrence class. Therefore, there exists a contradiction with the assumption. \( \square \)

If activities in the co-occurrence class occur continuously in any trace, it can be known from Theorem 1 that there exists a causal dependence between these activities.

For the concurrent structure, according to the behavioral characteristics of the concurrent structure, different concurrent branches will produce different co-occurrence classes. For instance, the concurrent structure shown in Figure 7c produces two co-occurrence classes \( \{x, a, b, y\} \) and \( \{c, d\} \). Since the order of activities on different concurrent branches does not have restrictions, the activities in a co-occurrence class may not necessarily occur consecutively in an executable trace. In Figure 7c, the activities in the co-occurrence class \( \{x, a, b, y\} \) do not occur consecutively in the executable sequence \( [x, a, b, y, d] \). However, causal occurrence dependencies still exist between the activities from each concurrent branch. In the following, Theorem 2 describes the substructure characteristics corresponding to co-occurrence classes with concurrent relationships.

**Theorem 2** (A substructure corresponding to the co-occurrence class with a concurrent relationship). Let \( L \) be an event log, and \( \text{COC}_i = \{a_{i_1}, a_{i_2}, \ldots, a_{i_p}\}, \text{COC}_j = \{a_{j_1}, a_{j_2}, \ldots, a_{j_q}\} \). If \( \exists a_{i_k} \in \text{COC}_i, \exists a_{j_l} \in \text{COC}_j, 1 \leq k \leq p, 1 \leq l \leq q, a_{i_k} \| a_{j_l} \) holds, then the activities from \( \text{COC}_i \) and \( \text{COC}_j \) form a sequential substructure, respectively.

**Proof.** As \( \exists a_{i_k} \in \text{COC}_i, \exists a_{j_l} \in \text{COC}_j, \) and \( a_{i_k} \| a_{j_l} \) holds, then \( \text{COC}_i \) and \( \text{COC}_j \) contain activities from different concurrent branches, respectively, and one of the branches should include and-split and and-join activities. From the interleaving behavior relationship, we can know that there is no ordering constraint between activities from two concurrent branches in a trace. According to Property 2, \( a_{i_1} \prec a_{i_2} \prec \cdots \prec a_{i_p}, a_{j_1} \prec a_{j_2} \prec \cdots \prec a_{j_q} \) holds. Therefore, even if the activities in \( \text{COC}_i \) or \( \text{COC}_j \) do not occur consecutively in the trace, there still exists a causal dependency between the activities from each co-occurrence class. As a result, the activities from each concurrent branch constitute a sequential substructure respectively. \( \square \)

Theorem 2 shows that activities in concurrent branches constitute sequential substructures even if they do not occur consecutively in the trace. Theorem 3 addresses the substructure characteristics formed by activities in other co-occurrence classes.
Theorem 3 (A substructure corresponding to the co-occurrence class with strict order relationship). Let \( L \) be an event log and \( A_k \) be an activity set over \( L \), \( \text{COC}_k = \{a_k, \cdots, a_{k+1}, \cdots, a_l\} \), if \( \exists \sigma \in L, \sigma[k] = a_k, \sigma[l] = a_{k+1} \) (where \( l > k + 1 \) ) and for \( \forall a_j \in A_k/\text{COC}_k, a_i \) and \( a_j \) are not in a concurrent relationship, then \( a_k, \cdots, a_l \) form a sequence substructure, along with \( a_{k+1}, \cdots, a_l \) form a sequence substructure. Meanwhile, there exists a non-sequential substructure containing at least one visible activity between \( a_i \) and \( a_{k+1} \).

Proof. As \( \sigma[k] = a_k, \sigma[l] = a_{k+1} \), there exists a trace \( \sigma \) such that \( a_k \) and \( a_{k+1} \) do not occur consecutively. For \( \forall a_j \in A_k/\text{COC}_k, a_k \) and \( a_j \) are not in a concurrent relationship, it can be seen that the activities in \( \text{COC}_k \) do not belong to any concurrent branch. Thus, when activities \( a_k, a_{k+1} \) do not occur consecutively, this indicates that there exists at least one activity on the path from \( a_k \) to \( a_{k+1} \), meanwhile, they belong to the other co-occurrence class without \( \text{COC}_k \). Obviously, the activity is in a sequential relationship with \( a_k \) and \( a_{k+1} \); otherwise, the activity must belong to the same co-occurrence class with \( a_k \) and \( a_{k+1} \). Therefore, there is a non-sequential substructure containing at least one activity between \( a_k \) and \( a_{k+1} \). According to Theorems 1 and 2, the remaining activities of \( \text{COC}_k \) form sequential substructures, respectively. The activities in \( \text{COC}_k \) form the substructure shown in Figure 8. □

Figure 8. A substructure corresponding to the co-occurrence class with a strict order relationship.

5.2. Merging Structures of Co-Occurrence Classes

Section 5.1 illustrates a method for discovering a substructure from a single co-occurrence class. This section presents the method for merging the structure of co-occurrence classes according to the communication behavior relationship between them to construct more complex substructures. Section 5.2.1 addresses the communication behavior relationship between co-occurrence classes, and Section 5.2.2 explains how to merge multiple structures of co-occurrence classes to discover more complex substructures.

5.2.1. Communication Behavior Relationship between Co-Occurrence Classes

According to the behavior dependencies of the activities from different co-occurrence classes in the log, the communication behavior dependencies between the co-occurrence classes can be obtained. These communication behavior relationships include a sequential communication relationship, exclusive communication relationship, concurrent communication relationship, and cycle communication relationship. The analysis of the communication behavior relationship between the co-occurrence classes provides a theoretical basis for constructing a larger substructure by combining the co-occurrence classes.

Property 3 (Sequential communication relationship). Let \( L \) be an event log, and \( \text{COCSets} = \{\text{COC}_1, \text{COC}_2, \cdots, \text{COC}_n\} \) is a set of co-occurrence classes from event log \( L \). If \( \forall a_i \in \text{COC}_i, \forall b \in \text{COC}_j, a_i \rightarrow L b, b \rightarrow L a_i \) holds, then \( \text{COC}_i \rightarrow \text{COC}_j \).

Proof. Assume \( \text{COC}_i = \{a_{i_1}, a_{i_2}, \cdots, a_{i_p}\} \) and \( \text{COC}_j = \{a_{j_1}, a_{j_2}, \cdots, a_{j_q}\} \), \( a_{i_1} \rightarrow a_{i_2} \rightarrow \cdots \rightarrow a_{i_p} \) and \( a_{j_1} \rightarrow a_{j_2} \rightarrow \cdots \rightarrow a_{j_q} \) holds; meanwhile, \( a_{i_1} \rightarrow L a_{j_1} a_{j_2} \cdots a_{j_q} \rightarrow L a_{i_p} \) also holds. Therefore, \( \text{COC}_i \) and \( \text{COC}_j \) are in a sequential relationship. □
Property 4 (Concurrent communication relationship). Let \( L \) be an event log, \( \text{COCSets} = \{\text{COC}_1, \text{COC}_2, \ldots, \text{COC}_n\} \) is a set of co-occurrence classes from the event log \( L \). \( \exists a \in \text{COC}_i, \exists b \in \text{COC}_j, \) if \( a || b \) holds, then \( \text{COC}_i \parallel \text{COC}_j \).

Proof. Assume \( \text{COC}_i = \{a_{i_1}, a_{i_2}, \ldots, a_{i_p}\}, \text{COC}_j = \{a_{j_1}, a_{j_2}, \ldots, a_{j_q}\} \), and \( a_{i_1} \succ a_{i_2} \succ \cdots \succ a_{i_p} \), \( a_{j_1} \succ a_{j_2} \succ \cdots \succ a_{j_q} \) holds. \( \exists a \in \text{COC}_i, \exists b \in \text{COC}_j, \) if \( a || b \) holds, then there must exist an activity (called and-split activity) where by firing this activity, each branch where the activity \( a \) or activity \( b \) is included will receive a token. If and-split and and-join are visible activities, according to the characteristics of concurrent structure, then and-split and and-join activities must appear in the co-occurrence invariant set of activity \( a \) or \( b \). Therefore, they must belong to the co-occurrence class of activity \( a \) or \( b \). If and-split and and-join are invisible activities, then they do not appear in the co-occurrence class of activity \( a \) or \( b \). Assume \( a_{i_1}, a_{i_2} \ldots \) are the visible and-split and and-join activities, respectively. As \( a || b \) and \( a_{i_1} \succ \cdots \succ a \succ \cdots \succ a_{i_{p-1}} \), \( a_{i_1} \succ \cdots \succ b \succ \cdots \succ a_{i_q} \) holds, \( \text{COC}_i \setminus \{a_{i_1}, a_{i_2}, \ldots, a_{i_p}\} \parallel \text{COC}_j \), we denote as \( \text{COC}_i \parallel \text{COC}_j \). \( \square \)

Property 5 (Exclusive communication relationship). Let \( L \) be an event log, \( \text{COCSets} = \{\text{COC}_1, \text{COC}_2, \ldots, \text{COC}_n\} \) is a set of co-occurrence classes from the event log \( L \). If \( \exists a \in \text{COC}_i, \exists b \in \text{COC}_j, a + b \) holds, then \( \text{COC}_i + \text{COC}_j \).

Proof. Obviously it is true. \( \square \)

Property 6 (Cycle communication relationship). Let \( L \) be an event log, \( \text{COCSets} = \{\text{COC}_1, \text{COC}_2, \ldots, \text{COC}_n\} \) is a set of co-occurrence classes from the event log \( L \). If \( \exists a \in \text{COC}_i, \exists b \in \text{COC}_j, a || b \) holds, then \( \text{COC}_i \triangleleft \text{COC}_j \).

Proof. As \( a || b, a \succ b \) and \( b \succ a \) holds. Meanwhile, as \( a || b, a \succ a, b \succ b \) holds. Therefore, the activity from \( \text{COC}_j \) and \( \text{COC}_i \) are in cycle relationship. \( \square \)

5.2.2. Combining Multiple Co-Occurrence Classes to Find More Complex Substructures

According to the communication behavior relationship between co-occurrence classes, the method of constructing multiple substructures from co-occurrence classes is provided below. These substructures include concurrency, conflict, sequence, and loop.

If \( \exists a \in A_L, \exists \text{COC}_i \in \text{COCSet}, \forall a \in \text{COC}_i, b \rightarrow a \) holds, then it is abbreviated as: \( b \rightarrow \text{COC}_i \). If \( \forall a \in \text{COC}_i, a \rightarrow b \) holds, then it is abbreviated as: \( \text{COC}_i \rightarrow b \). Similarly, for || +, it is abbreviated as: \( b || \text{COC}_i, b + \text{COC}_i \).

1. Concurrent substructure: It is known that \( \text{COC}_i \parallel \text{COC}_j \), if exists activities \( x_1, x_2, \ldots, x_t, \ldots, y_1, \ldots, y_k \in \text{COC}_j \), where \( x_1 > x_2 > \cdots > x_t, y_1 > y_2 > \cdots > y_k \), \( x_1 \succ y_1 \), and \( x_i \rightarrow \text{COC}_j, \text{COC}_j \rightarrow y_1 \) holds, then \( x_i, y_1 \) corresponding to and-split and and-join activity respectively. If not exists \( x_i \) or \( y_1 \) such that \( x_i \rightarrow \text{COC}_j \) or \( \text{COC}_j \rightarrow y_1 \), then and-split or and-join are hidden transitions. Meanwhile, \( \forall a \in \text{COC}_i \setminus \{x_1, x_2, \ldots, x_t, y_1, \ldots, y_k\}, \forall b \in \text{COC}_j, a || b \) holds, then the typical substructure of \( \text{COC}_i \) and \( \text{COC}_j \) is shown in Figure 9.

![Figure 9. Typical concurrent substructure composed of \( \text{COC}_i \) and \( \text{COC}_j \).](image-url)

2. Conflict substructure: If \( \text{COC}_i + \text{COC}_j \), then \( \text{COC}_i \) and \( \text{COC}_j \) constitute the branch of the conflict substructure, respectively.

3. Loop substructure: For loop structures, the key is to distinguish the do and redo part in a loop structure. Firstly, the characteristics of three different loop substructures
are analyzed, and the invariant predecessor set, invariant successor set, co-occurrence invariant set, and co-occurrence class of each activity in different loop substructures are obtained. Then, different loop substructures are obtained according to the characteristics of the invariant predecessor set, the invariant successor set, the co-occurrence invariant set, and the co-occurrence class of the activities in the log.

When the loop body is a sequential structure, a typical structure is shown in Figure 10, and Table 8 shows the invariant predecessor (or successor) set and co-occurrence invariant set of each activity in the loop substructure.

![Diagram](image_url)

**Figure 10.** Loop substructure with a sequential structure.

**Table 8.** Invariant set of activities in Figure 9.

<table>
<thead>
<tr>
<th>Activity</th>
<th>InvPreds</th>
<th>InvSuccS</th>
<th>Oi</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>∅</td>
<td>[a,b,y]</td>
<td>[x,a,b,y]</td>
</tr>
<tr>
<td>a</td>
<td>∅</td>
<td>[b]</td>
<td>[a,b]</td>
</tr>
<tr>
<td>b</td>
<td>[a]</td>
<td>∅</td>
<td>[a,b]</td>
</tr>
<tr>
<td>c</td>
<td>[a,b]</td>
<td>[d,a,b]</td>
<td>[a,b,c,d]</td>
</tr>
<tr>
<td>d</td>
<td>[c,a,b]</td>
<td>[a,b]</td>
<td>[a,b,c,d]</td>
</tr>
<tr>
<td>e</td>
<td>∅</td>
<td>[f]</td>
<td>[e,f]</td>
</tr>
<tr>
<td>f</td>
<td>[e]</td>
<td>∅</td>
<td>[e,f]</td>
</tr>
<tr>
<td>y</td>
<td>[x,a,b]</td>
<td>∅</td>
<td>[x,a,b,y]</td>
</tr>
</tbody>
</table>

Co-occurrence classes obtained from Table 8 are COC$\text{Sets} = \{ \{x,y\}, \{a,b\}, \{c,d\} \}$

**Corollary 1.** Let $L$ be an event log, COC$\text{i}$, COC$\text{j}$ are co-occurrence classes from the event log $L$, and COC$\text{i} \cap$ COC$\text{j}$. If $\forall a \in$ COC$\text{i}$, InvPreds$\text{(a)} \cap$ InvSuccS$\text{(a)} = $ COC$\text{j}$ holds, then COC$\text{i}$ is the redo substructure of the loop structure, while COC$\text{j}$ is the do substructure of the loop structure.

When the loop body is an exclusive structure, a typical structure is shown in Figure 11, and Table 9 shows the invariant predecessor (or successor) set and co-occurrence invariant set of each activity in the loop substructure.

**Table 9.** Invariant set of activities in Figure 11.

<table>
<thead>
<tr>
<th>Activity</th>
<th>InvPreds</th>
<th>InvSuccS</th>
<th>Oi</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>∅</td>
<td>[y]</td>
<td>[x,y]</td>
</tr>
<tr>
<td>a</td>
<td>∅</td>
<td>[b]</td>
<td>[a,b]</td>
</tr>
<tr>
<td>b</td>
<td>[a]</td>
<td>∅</td>
<td>[a,b]</td>
</tr>
<tr>
<td>c</td>
<td>∅</td>
<td>[d]</td>
<td>[c,d]</td>
</tr>
<tr>
<td>d</td>
<td>[c]</td>
<td>∅</td>
<td>[c,d]</td>
</tr>
<tr>
<td>e</td>
<td>∅</td>
<td>[f]</td>
<td>[e,f]</td>
</tr>
<tr>
<td>f</td>
<td>[e]</td>
<td>∅</td>
<td>[e,f]</td>
</tr>
<tr>
<td>y</td>
<td>[x]</td>
<td>∅</td>
<td>[x,y]</td>
</tr>
</tbody>
</table>
Co-occurrence classes obtained from the Table 9 are as follows: 
\[ \text{COCSets} = \{\{x, y\}, \{a, b\}, \{c, d\}, \{e, f\}\}. \]

**Corollary 2.** Let \( L \) be an event log, \( \text{COC}_i, \text{COC}_j, \text{COC}_k \) are co-occurrence classes from the event log \( L \), and \( \text{COC}_i \cap \text{COC}_j \cap \text{COC}_k \), if the following two conditions hold:

1. For \( \forall \sigma \in L, \text{Occur}(\text{COC}_i, \sigma) + \text{Occur}(\text{COC}_j, \sigma) = \text{Occur}(\text{COC}_k, \sigma) + 1; \)
2. \( \exists \sigma \in L, \text{Occur}(\text{COC}_i, \sigma) = 0, \text{Occur}(\text{COC}_j, \sigma) > 0 \) or \( \text{Occur}(\text{COC}_j, \sigma) = 0, \text{Occur}(\text{COC}_i, \sigma) = 0 \)

where \( \text{Occur}(\text{COC}_k, \sigma) \) represents the number of times the sequence formed by \( \text{COC}_k \) appears in the trace \( \sigma \), then \( \text{COC}_i, \text{COC}_j \) is the do part of the loop structure with \( \text{COC}_i \) and \( \text{COC}_j \) in an exclusive substructure, and \( \text{COC}_k \) is the redo part of the loop structure.

When the loop body is a concurrent structure, a typical structure is shown in Figure 12, and Table 10 shows the invariant predecessor (or successor) set and co-occurrence invariant set of each activity in the loop substructure.

![Figure 12. Loop substructure with a concurrent structure.](image)

**Table 10. Invariant set of activities in Figure 12.**

<table>
<thead>
<tr>
<th>Activity</th>
<th>InvPreds</th>
<th>InvSuccS</th>
<th>Oi</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>( \emptyset )</td>
<td>( {a, b, c, d, y} )</td>
<td>( {x, a, b, c, d, y} )</td>
</tr>
<tr>
<td>a</td>
<td>( {x} )</td>
<td>( {b, y} )</td>
<td>( {x, a, b, y} )</td>
</tr>
<tr>
<td>b</td>
<td>( {x, a} )</td>
<td>( {y} )</td>
<td>( {x, a, b, y} )</td>
</tr>
<tr>
<td>c</td>
<td>( {x} )</td>
<td>( {d, y} )</td>
<td>( {x, c, d, y} )</td>
</tr>
<tr>
<td>d</td>
<td>( {x, c} )</td>
<td>( {y} )</td>
<td>( {x, c, d, y} )</td>
</tr>
<tr>
<td>e</td>
<td>( {x, a, b, c, d, y} )</td>
<td>( {x, a, b, c, d, y} )</td>
<td>( {e, x, a, b, c, d, y} )</td>
</tr>
<tr>
<td>y</td>
<td>( {x, a, b, c, d} )</td>
<td>( \emptyset )</td>
<td>( {x, a, b, c, d, y} )</td>
</tr>
</tbody>
</table>

Co-occurrence classes obtained from Table 10 are \( \text{COCSets} = \{\{x, a, b, y\}, \{c, d\}, \{e\}\}. \)

**Corollary 3.** Let \( L \) be an event log, \( \text{COC}_i, \text{COC}_j, \text{COC}_k \) are co-occurrence classes from the event log \( L \), and \( \text{COC}_j \cap \text{COC}_k \). If the following two conditions hold,

1. \( \forall a \in \text{COC}_k, \text{InvPreds}(a) \cap \text{InvSuccs}(a) = \text{COC}_i \cup \text{COC}_j \),
2. \( \forall \sigma \in L, \text{Occur}(\text{COC}_i, \sigma) = \text{Occur}(\text{COC}_j, \sigma) \),
then \( \text{COC}_i, \text{COC}_j \) is the do part of the loop structure with \( \text{COC}_i \) and \( \text{COC}_j \) in a concurrent substructure, while \( \text{COC}_k \) is the redo part of the loop structure.

**Definition 10.** [Merge activities in relation \( \rightarrow \) [6]] When the left activities in two \( \rightarrow \) relations are the same (\( a \rightarrow b, a \rightarrow c \)), two possible substructures can be created.

(a) The places of \( \rightarrow \) relations are merged into a single one if and only if (\( b \rightarrow c \lor c \rightarrow b \)). This is denoted as \( [a, b+c] \).

(b) The places of \( \rightarrow \) relation are not merged if and only if (\( b \rightarrow c \lor c \rightarrow b \)). This is denoted as \( [a, b|c] \).

Similarly, for \( \rightarrow \) relations when the right activity is common, i.e., (\( b \rightarrow a \lor c \rightarrow a \)), the substructure created will be either \( [b|c, a] \) or \( [b+c, a] \) accordingly. In general, a set of \( \rightarrow \) relations in the form (\( a \rightarrow b, a \rightarrow c, \ldots, a \rightarrow d \)) may produce either \( [a, b+c+d] \) or \( [a, b|c||d] \). Consequently, the merging can be applied to composed relations, i.e., \( [b+c, a] \) and \( [b+c, d] \) leads \( [b+c, a+d] \).

Algorithm 4 presents the construction steps for discovering process models from event logs using the methods in Sections 5.1 and 5.2. Step 1 constructs the co-occurrence class from the event log, Steps 2–4 use the method in Section 5.1 to construct the corresponding substructure for each co-occurrence class, Steps 5–7 use the method in Section 5.2.1 to obtain the communication behavior relationship between any two co-occurrence classes, Steps 8–13 use the method proposed in Section 5.2.2 to merge several co-occurrence classes into a larger sub-module, and Step 14 further merges the remaining sub-modules into a complete process model through the directly-follows relationship between activities.

**Algorithm 4:** Construct Petri net from the event log

**Input:** the event log \( L \)

**Output:** Petri Net

1. \( \text{cocset} = \text{GetCOCSet}(L, A_L) \);
2. \( \text{foreach} \ \text{coc in cocset do} \)
3. \( \quad \text{substructset} = \text{GetSubstruct} (\text{coc}) \)
4. \( \text{end} \)
5. \( \text{for} \ \text{coc}_i, \text{ coc}_j \ \text{in} \ \text{cocset do} \)
6. \( \quad \text{GetbehaviorRelation} (\text{coc}_i, \text{coc}_j) \)
7. \( \text{end} \)
8. \( \text{for} \ \text{coc}_i, \text{ coc}_j \ \text{in} \ \text{cocset do} \)
9. \( \quad \text{if} (\text{coc}_i \parallel \text{coc}_j) \ \text{merge} (\text{coc}_i, \text{coc}_j, \text{parallel}); \)
10. \( \quad \text{if} (\text{coc}_i + \text{coc}_j) \ \text{merge} (\text{coc}_i, \text{coc}_j, \text{choice}); \)
11. \( \quad \text{if} (\text{coc}_i \odot \text{coc}_j) \ \text{merge} (\text{coc}_i, \text{coc}_j, \text{loop}); \)
12. \( \quad \text{if} (\text{coc}_i \rightarrow \text{coc}_j) \ \text{merge} (\text{coc}_i, \text{coc}_j, \text{sequence}); \)
13. \( \text{end} \)
14. \( \text{find the missing} \rightarrow \text{relations}; \)
15. \( \text{build the final Petri Net as stated def 10} \)

6. Experiment Evaluation

**Experiment purpose.** In this section, we experimentally evaluate whether the method proposed in this paper can discover long-distance dependencies from event logs and can rediscover process models with complex cycle structures from small event logs. Several event logs were used to compare our method with other process discovery methods, such as Inductive Miner (IM) [17], the Heuristics Miner (HM) algorithm [14], Inductive Miner-Incomplete-ness (IMin) [17], and Conjoint Occurrence Miner (CoM) [6]. The experimental evaluation in this section mainly answers two questions. The first question is whether the proposed method can discover long-distance dependencies. The second question is whether the proposed method can correctly discover the original model with a complex structure.
from incomplete event logs caused by concurrency. In addition to concurrent, sequential, and exclusive structures, the model also may contain complex loop structures.

**Experiment design.** For problem 1, we generate event logs containing 1000 traces by simulating the model M1 with long-distance dependencies shown in Section 3. Then, we try to use these logs to rediscover the process model using different methods. We report on the experimental comparison results of the quality of the process model and the consistency between the original model and the discovered model [31]. Meanwhile, we report the discovery of long-distance dependencies in the original model. For problem 2, event logs with different completeness levels of directly-follows relationships are generated by simulating model M2 shown in Figure 6. These logs are obtained by removing some of the directly-follows relationships derived from the concurrent structure. Different algorithms are used to reconstruct the original process model M2 from these event logs of different sizes, and the differences between the discovered process model and the original model are reported. The method proposed in [31] is used to measure the Degree of Trace Consistency (DTC) and Degree of Profile Consistency (DPC) of alignment between the models. All miners use default parameters, and h in IMin is set to 0.

**Experiment Result.** From Table 11, it can be seen that the proposed method and the CoM method can discover long-distance dependencies, but compared with the CoM method, the proposed method can better discover the cyclic structure. IMin, IM, and HM cannot rediscover the original process model due to their inability to discover long-distance dependencies. The CoM method also cannot discover the original process model due to its limitations in the cyclic structure. When different methods are applied to the event log generated by M1, the consistency between the discovered process model and the original model is measured from two aspects by using the method given in [31]. The experimental results show that the method proposed in this paper has good results regarding the consistency of the discovered process model and the original model in terms of both the degree of trace consistency and degree of profile consistency. Since long-distance dependencies do not affect the behavior profile relationship between activities, the process model obtained by the IMin, IM, and HM algorithms has a good degree of profile consistency with the original model.

Table 11. Experimental results obtained by different methods.

<table>
<thead>
<tr>
<th>Item</th>
<th>Imin</th>
<th>IM</th>
<th>HM</th>
<th>CoM</th>
<th>The Proposed Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long-distance dependency</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Cycle structure rediscovery</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>DTC</td>
<td>93.66%</td>
<td>93.66%</td>
<td>95.45%</td>
<td>96.71%</td>
<td>100%</td>
</tr>
<tr>
<td>DPC</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>98.70%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Figure 13 shows that the IM algorithm and the HM algorithm require that the event log is complete with a directly-follows relationship, and IMin, COM, and the proposed method can handle an incomplete event log. Although the IMin method can handle the incompleteness of the event log, it cannot discover the long-distance dependency; thus, the value of DTC is lower than that of DPC between the model discovered by IMin and the original model. IMin mainly retains the behavior relationship with high probability based on the directly-follows relationship between activities to determine the behavior between the activities. Hence, the requirement for completeness of IMin is lower than that of the IM and HM algorithms but is higher than the proposed method and the CoM method. The proposed method and CoM method require that the completeness degree of the log is up to 85%. However, because the CoM method has some limitations in discovering the cycle structure, the consistency degree between the discovered model and the original model cannot reach 1, even if the log completeness degree is more than 85%.
IMin requires the completeness ratio of the log to be about 90%. However, since the process model discovered by IMin is block-structured, some hidden transitions can be added when building the model. These hidden transitions may cause the model to generate additional behaviors, which may lead to a certain deviation between the discovered model and the original model. Thus, for the IMin method, even if the log completeness ratio reaches 90%, the consistency degree between the discovered model and the original model is still less than 1.

As the results of our experiments show, the proposed method has two advantages compared to the others. One advantage is that our method can handle incomplete event logs better than the IMin algorithm. However, the IM and HM approaches are heavily dependent on the log completeness. Although both the proposed method and the COM method can deal with the incompleteness caused by high concurrency, the COM method has limitations in discovering the loop structure, whereas the proposed method can also discover complex loop structures in addition to sequential, exclusive, and concurrent structures. The other advantage is that the proposed method can discover long-distance dependencies from the event logs. As the experiment shows, IMin, IM, and HM do not have an inability to discover long-distance dependencies.

![Figure 13. The degree of consistency comparison.](image)

7. Conclusions and Future Work

Based on [6], a method for discovering a process model with multiple structure types from incomplete event logs is presented that can discover not only sequential, concurrent, and conflicting structures but also complex cyclic structures. This paper analyzes the relationship between co-occurrence classes from log and co-occurrence classes from the model and addresses communication behavior profiles between co-occurrence classes from a log using the technology of the behavior profile. A series of propositions are presented to point out how to construct the substructure through the co-occurrence class. Subsequently, the whole process model is constructed from the event log. The proposed method can still discover the original process model from an incomplete event log, even if the model contains a complex cyclic structure. In addition, it also can correctly discover a long-distance dependency implied in the log. For large event logs, it will be time-consuming to construct co-occurrence classes and analyze their communication behavior profiles. In the future, we will consider using co-occurrence classes to construct frequent patterns from event logs to analyze local behaviors in business processes.
Author Contributions: Methodology, L.W.; formal analysis, L.W.; writing—original draft preparation, L.W.; writing—review and editing, X.F.; visualization, C.S.; project administration, X.F.; supervision, X.F. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Natural Science Foundation, China (No. 61572035, 61402011), the Leading Backbone Talent Project in Anhui Province, China (No. 2020-1-12), the Natural Science Foundation of Anhui Province, China (No. 2008085QD178), Anhui Province Academic and Technical Leader Foundation (No. 2019H239), Anhui Province College Excellent Young Talents Fund Project of China (No. gxyqZD202020), Open Research Fund of Anhui Province Engineering Laboratory for Big Data Analysis and Early Warning Technology of Coal Mine Safety (NO. CSBD2022-ZD03).

Data Availability Statement: All the data in this paper were obtained by simulating the model. Experimental data and code related to this paper can be obtained by contacting the corresponding author.

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

References
3. Van Der Aalst, W. Process Mining: A 360 Degree Overview; Springer: Cham, Switzerland, 2022, pp. 3–34.