Mining Process Models with Prime Invisible Tasks

Lijie Wen[1,2], Jianmin Wang[1,4,5], Wil M.P. van der Aalst[3], Biqing Huang[2], and Jiaguang Sun[1,4,5]

[1] School of Software, Tsinghua University, Beijing, China
wenlj00@mails.thu.edu.cn,{jimwang, hbq, sunjg}@tsinghua.edu.cn
[2] Department of Automation, Tsinghua University, Beijing, China
[3] Department of Mathematics & Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands.
w.m.p.v.d.aalst@tue.nl
[4] Key Laboratory for Information System Security, Ministry of Education
[5] Tsinghua National Laboratory for Information Science and Technology (TNList)

Abstract. Process mining is helpful for deploying new business processes as well as auditing, analyzing and improving the already enacted ones. Most of the existing process mining algorithms have problems in dealing with invisible tasks, i.e., such tasks that exist in a process model but not in its event log. This is a problem since invisible tasks are mainly used for routing purpose but must not be ignored. In this paper, a new process mining algorithm named $\alpha^{\#}$ is proposed, which extends the mining capacity of the classical α algorithm by supporting the detection of prime invisible tasks from event logs. Prime invisible tasks are divided into three types according to their structural features, i.e., SKIP, REDO and SWITCH. After that, the new ordering relation for detecting mendacious dependencies between tasks that reflects prime invisible tasks is introduced. A reduction rule for identifying redundant "mendacious" dependencies is also considered. Then the construction algorithm to insert prime invisible tasks of SKIP/REDO/SWITCH types is presented. The proposed $\alpha^{\#}$ algorithm has been evaluated using both artificial and real-life logs and the results are promising.

1 Introduction

Process mining aims at the *automatic* discovery of *valuable* information from an event log. The first dedicated process mining algorithms were proposed in [10, 11]. The discovered information can be used to deploy new systems that support the execution of business processes or as an analytical tool that helps in auditing, analyzing and improving already enacted business processes. The main benefit of process mining techniques is that information is *objectively* compiled. In other words, process mining techniques are helpful because they gather information about what is *actually* happening according to an event log of an organization, and not what people *think* is happening in this organization.

The scenarios of applying process mining technology are illustrated in Figure 1. This figure shows that process mining can be used in two ways, i.e., before and after an information system supporting business processes is deployed. In the first scenario, the event logs are collected manually. Process mining can be used to mine a process model (i.e., *process visualization*) from a given log and compare the behaviors of the mined model with the given log (i.e., *conformance testing*). In the second scenario, process mining can be used in a similar way except that the event logs are recorded by information systems automatically. Moreover, we can compare the mined process model with the predefined one to find the discrepancies between them (i.e., *delta analysis*).

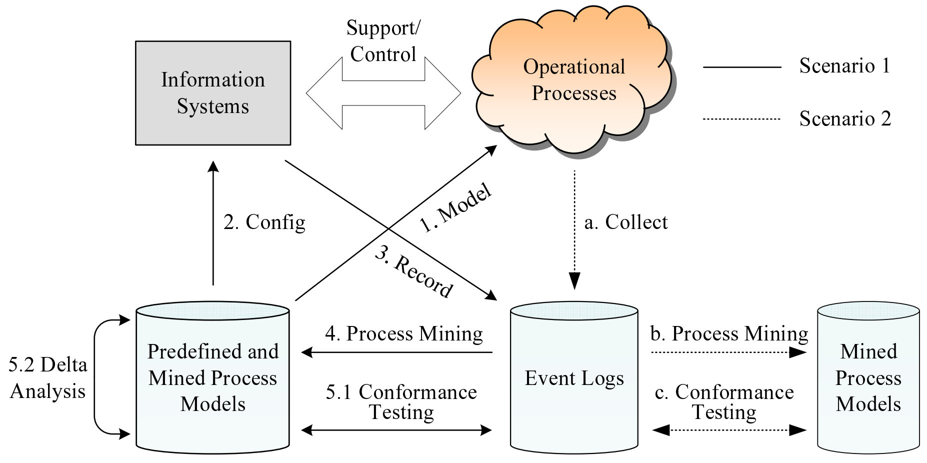

Fig. 1. The scenarios of applying process mining technology

Although quite a lot of work has been done on process mining, there are still some challenging problems left [4–7, 20], i.e., short loops, duplicated tasks, invisible tasks, non-free-choice constructs, time, noise, incompleteness, etc. The issue of short loops is solved in [21]. For discussions about duplicated tasks, readers are referred to [16, 17, 22]. [22, 27, 28] attempt to resolve most kinds of non-free-choice constructs. Time aspects are partially considered in [2]. Noise and incompleteness are discussed in [19]. In this paper, we will investigate how to mine process models with prime invisible tasks from event logs.

1.1 Motivation

Invisible tasks are such tasks that exist in a process model but not in its event log. They are difficult to mine because they do not appear in any event trace of an event log. However, they are very common in real-life process models. The following situations can lead to process models containing invisible tasks:

– There are tasks in process models that are used for routing purpose only. The executions of such routing tasks are not recorded automatically or manually.

– There are real tasks that have been executed but they are systematically lost in event traces (i.e., not recorded).
– The enactment services of processes allow skipping or redoing current task and jumping to any previous task as a result of human interventions to improve the flexibility of execution[8]. But such human interventions cannot be expressed in the control flow of the process model in order to keep the model as simple and clear as possible.

There are quite a number of real-life business processes containing invisible tasks. When deploying our Product Lifecycle Management system named TiPLM[6] for Shaoxing Electric Power Bureau[7], Zhejiang Province, China, we found that the deployed 43 process models corresponding to business processes in this organization all contain many invisible tasks. TiPLM contains a business process management subsystem named TiWorkflow and there are routing tasks in process models whose executions are not recorded in event traces. Figure 2 shows one of these process models, which is modeled in the system specific language of TiWorkflow.

For each process model in TiWorkflow similar to that shown in Figure 2, there should be only one start node (the green circle) and only one end node (the red round rectangle). The diamonds represent the task nodes and the other kinds of nodes are routing nodes, i.e., AND-split, AND-join, XOR-split and XOR-join. It is the change process of design drawings. There are totally three kinds of virtual or real objects in this process, i.e., *change request*, *change task* and *changed drawings*. The *change request* will pass through *apply*, *audit* tasks. If the result of *audit* is negative, the process ends. Otherwise, the *change task* will pass through *dispatch*, *assign*, *modify* and *verify* tasks. After that, the *changed drawings* will pass through *review*, *review technically* and *print and dispatch* tasks. If one of the results of the two *review* tasks is negative, the process will come back to the *assign* task. When the process instances are executed, TiWorkflow engine only records the executions of task nodes. The executions of routing nodes are totally ignored. In the business environment before TiPLM was deployed, similar things happened and the event traces were recorded manually on paper.

For the process model shown in Figure 2, it can be easily seen that the number of routing nodes is about half of the number of all nodes, i.e., 12/27≈44.4%. Table 1 shows the proportions for routing nodes out of all nodes in each of the above-mentioned 43 process models taken from Shaoxing Electric Power Bureau. The average proportion nearly approaches 40%. In other organizations (such as Xiamen King Long United Automotive Industry Co. Ltd.[8] and Hebei Zhongxing Automobile Manufacture Co. Ltd.[9], etc.) that have deployed TiPLM, similar ratios are found. It shows that the need for mining process models with invisible tasks from event logs is high, i.e., one cannot assume that the mined process only contains visible tasks.

[6] http://www.thit.com.cn/TiPLM/TiPLM.htm
[7] http://www.sx.zpepc.com.cn
[8] http://www.xmklm.com.cn/english/js6.jsp
[9] http://www.zxauto.com

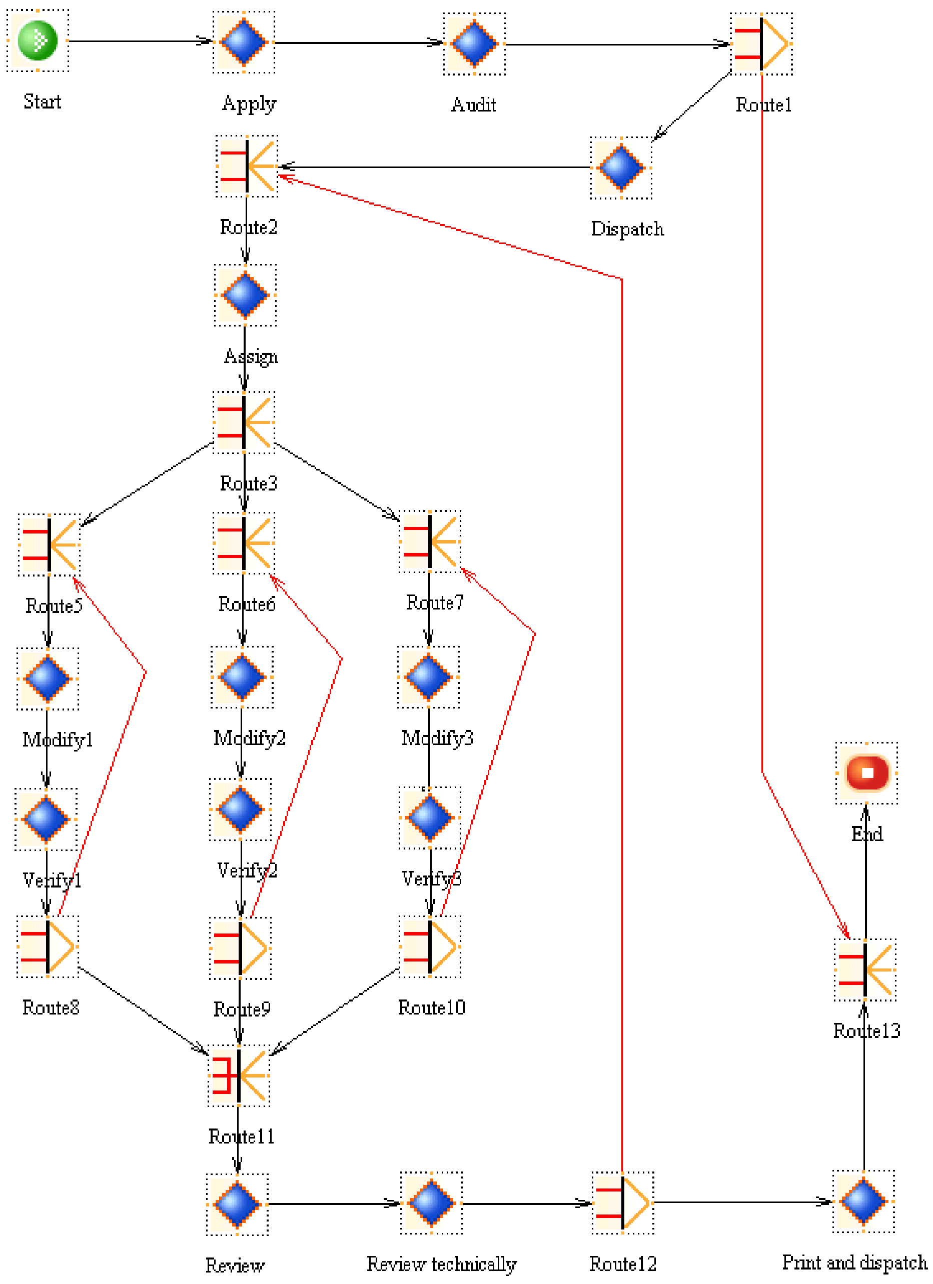

Fig. 2. The change process of design drawings defined in TiWorkflow

Table 1. The proportion for routing tasks out of all tasks in real-life process models: N_{RT}-the number of routing tasks, N_{AT}-the number of all tasks

The name of each process model	N_{RT}	N_{AT}	**Proportion**
Change process	7	40	17.5%
Technical change notification process	8	39	20.5%
Change process of design drawings	12	27	44.4%
Debug process of drawings in control center	21	40	52.5%
Debug process of drawings in circuitry work area	21	40	52.5%
Debug process of drawings in remedy work area	21	40	52.5%
Debug process of drawings in operation work area	21	40	52.5%
Process for checking completion on rebuilt engineering	62	136	45.6%
Process for checking completion on new engineering	64	139	46.0%
...	...	...	...
Summary	**1317**	**3306**	**39.8%**

1.2 Problems related to invisible tasks

The problems encountered when mining process models using the classical α algorithm [7] from event logs generated by WF-nets (see [1]) containing invisible tasks will be investigated in this subsection. For example, in Figure 3 and Figure 4, N_1 to N_6 are the original WF-nets and N'_1 to N'_6 are the corresponding mined models derived from the complete event logs W_1 to W_6 respectively by using α algorithm[10]. The black block transitions without labels represent invisible tasks. All the original WF-nets are sound, but the mined models have various problems.

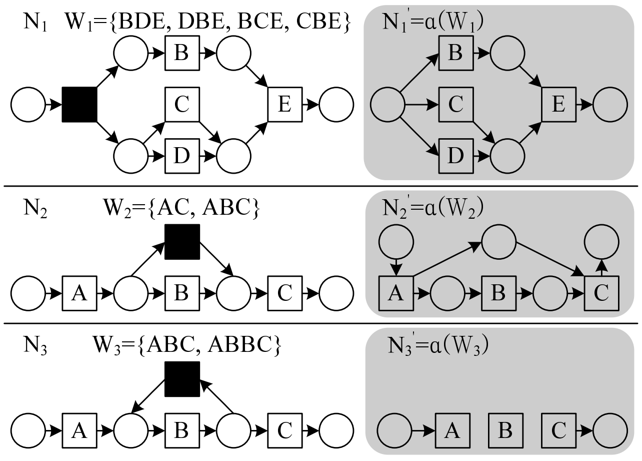

Fig. 3. Problems encountered when mining process models using α algorithm: case one

Tasks B and C as well as B and D are parallel in N_1, while they are mutually exclusive in N'_1. Here N'_1 is not a sound WF-net because a deadlock will always

[10] A log is complete if it contains information about all causal dependencies [7]. This notion will be explained later.

occur. Although N_2' is a sound WF-net, C cannot directly follow A and N_2' contains an implicit place. As a result, the behavior of N_2' is not equivalent with that of N_2 because it cannot produce the event trace AC. In N_3, B behaves like a length-one-loop task. However, C never directly follows A. There will not a place connecting A and C in N_3', which should be the only place associated with B. Here N_3' is not a WF-net at all. N_4', N_5' and N_6' are all WF-nets but not sound. N_4 is more general than N_2 and N_3 is a special case of N_5. The invisible task in N_6 connects two separate execution branches.

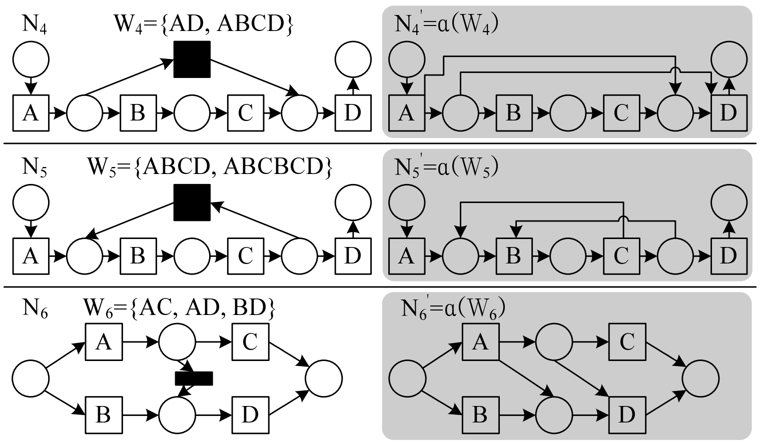

Fig. 4. Problems encountered when mining process models using α algorithm: case two

The steps for constructing the mined model in α algorithm result in the above issues. Each mined process model has one of the following features.

1. It is a sound WF-net, but it cannot reproduce the given log, e.g., N_2'.
2. It is a WF-net but not sound and cannot reproduce the given log, e.g., N_1', N_4', N_5' and N_6'.
3. It is not a WF-net at all. It cannot reproduce the given log and may even generate lots of redundant event traces, e.g., N_3'.

In this paper, a new mining algorithm will be proposed based on the α algorithm, in which most invisible tasks can be derived correctly and efficiently. We choose the α algorithm as a base for its theoretical foundation. Still the correctness of detection method for invisible tasks will be proved and the classification of invisible tasks will be provided.

The remainder of the paper is organized as follows. Section 2 introduces related work on mining invisible tasks. Section 3 gives the classification of invisible tasks according to their structural features, i.e., typical patterns encountered when dealing with invisible tasks. The detection methods of invisible tasks are proposed in Section 4. The new mining algorithm $\alpha^{\#}$ is presented in Section 5. Section 6 shows the evaluation results on the new algorithm using both artificial and real-life event logs. Section 7 concludes the paper and gives future work.

2 Related work

Here only process mining algorithms based on Petri nets [12, 23] are considered. For other process mining algorithms not based on Petri net, which do not need to concern invisible tasks, their emphases are focused on the efficient identification of relationships (i.e., AND-split, XOR-split, AND-join and XOR-join) between each pair of input/output arcs of the same task [9–11, 14, 15, 26].

A Synchro-net based mining algorithm is proposed in [18]. The authors state that short loops and invisible tasks can be handled with ease. However, neither the original model nor the mined model contains any invisible task. In the mining algorithm, no steps seem to handle invisible tasks.

[25] attempts to mine decisions from process logs, which emphasizes detecting data dependencies that affect the routings of cases. When interpreting the semantics of the control flows in the mined decisions, the authors propose a descriptive method to identify decision branches starting from invisible tasks. This method cannot handle all kinds of invisible tasks. Even when there are other decision points with join nodes on one decision branch, the method fails.

The genetic mining algorithm (GA for short) is the only method that natively supports the detection of invisible tasks [22]. It uses the basic idea of the genetic algorithm and defines two genetic operators for process mining, i.e., crossover and mutation. It aims at supporting all the common control flow constructs in current business processes, especially duplicated tasks, invisible tasks, non-free-choice constructs. The genetic mining algorithm can partially handle these three constructs. However, this algorithm needs many user-defined parameters and it cannot always guarantee to return the most appropriate results within limited time. The number of invisible tasks in the mined model is often much greater than that of the original model. As a result, the behavior of the mined model is not equivalent with that of the original model or does not conform to the event log. Furthermore, the biggest disadvantage of the genetic mining algorithm is its huge time consumption. Even for a small event log generated by a simple process model, the algorithm takes at least several minutes to mine an acceptable model.

In summary, there is still no efficient mining algorithm that can handle invisible tasks well. This paper will focus on mining process models from event logs with invisible tasks based on the classical α algorithm proposed in [7]. It is also an extension of the work done in [29] by extending the motivation, related definitions and theorems and the experimental results.

3 Definitions and classification of invisible tasks

First some definitions about invisible tasks in a WF-net will be given (Subsection 3.1). After that, we will introduce the classification of invisible tasks in detail (Subsection 3.2). Note that we assume the reader to be familiar with WF-nets and soundness. WF-nets are a special class of Petri nets with a source place and a sink place. A WF-net is sound if it is always possible to move a token from the source place to the sink place, i.e., no deadlocks and livelocks. Moreover, all parts of the WF-net are executable (i.e., there are no dead tasks) and the net is safe (i.e., a place should never hold two or more tokens).

3.1 Definitions about invisible tasks

Before going into the details about invisible tasks, a function about all traces of a WF-net should be introduced first.

Definition 1 (Trace function *traces***).** *Let $N = (P, T_V \bigcup T_{IV}, F)$ be a sound WF-net. $traces(N) \subseteq T_V^*$ is the set of all firing sequences leading from the marking with a token in the source place i to the marking with a token in the sink place o by removing all tasks in T_{IV} from each firing sequence, i.e., $\forall t \in T_V, \exists \sigma \in traces(N) : t \in \sigma$ and $\forall \sigma \in traces(N), \forall t \in T_{IV} : t \notin \sigma$. Such a WF-net N is call IWF-net, here we use a new notation $N = (P, T_V, T_{IV}, F)$.*

Based on an IWF-net and the trace function *traces*, the conceptual invisible task can be defined as follows.

Definition 2 (Invisible task). *Let $N = (P, T_V, T_{IV}, F)$ be a sound IWF-net and $W \subseteq traces(N)$ be an event log of N. For any task $t \in T_{IV}$, t is an invisible task with respect to W.*

Not all invisible tasks can be rediscovered from the corresponding event logs. In fact, if an invisible task does not affect the behavior of a WF-net, it cannot be detected by any mining algorithm. We will define so-called "prime invisible task" formally in the following, which has effects on the behavior of a WF-net. Before this, the following concept and two auxiliary notations are given.

Definition 3 (Invisible elementary path). *Let $N = (P, T_V, T_{IV}, F)$ be a sound IWF-net. An elementary path is a sequence $EP = (n_0, n_1, \ldots, n_k)$ where $\forall_{0 \le i < k} : (n_i, n_{i+1}) \in F$, $n_0, n_k \in P$ and $\forall_{0 \le i,j \le k} : (n_i = n_j) \Rightarrow (i = j)$. IEP is invisible iff $\forall_{0 \le i \le k} : (n_i \in T_V \bigcup T_{IV}) \Rightarrow n_i \in T_{IV}$. IEP_i and IEP_o denote the start place and the end place of IEP respectively.*

An elementary path in an IWF-net is a directed acyclic path, which starts from a place and walks along the direction of the arc between each pair of successive nodes and ends with another place. No two nodes on the path are the same. An invisible elementary path contains only invisible tasks. This concept will be used in the forthcoming notations. All invisible elementary paths of an IWF-net N is denoted as $\mathcal{IEP}_N$.

Definition 4 (Input/output visible task set). *Let $N = (P, T_V, T_{IV}, F)$ be a sound IWF-net. For any $p \in P$, we have:*

- $\circ p = \{t \in T_V | t \in \bullet p \vee \exists IEP \in \mathcal{IEP}_N : (t \in \bullet IEP_i \wedge p = IEP_o)\}$, *i.e., p's input visible task set.*
- $p\circ = \{t \in T_V | t \in p \bullet \vee \exists IEP \in \mathcal{IEP}_N : (t \in IEP_o \bullet \wedge p = IEP_i)\}$, *i.e., p's output visible task set.*

Definition 5 (Prime invisible task). *Let $N = (P, T_V, T_{IV}, F)$ be a sound IWF-net and $t \in T_{IV}$ be an invisible task. t is prime if the following requirements hold at the same time:*

1. ***Surround.*** *$\forall p \in \bullet t, \exists t' \in \bullet p : t' \in T_V$ and $\forall p \in t\bullet, \exists t' \in p\bullet : t' \in T_V$, i.e., each invisible task should have at least one direct preceding visible task and one direct successive visible task.*
2. ***Succession.*** *$\forall t' \in \{\circ p | p \in \bullet t\}, \forall t'' \in \{p \circ | p \in t\bullet\} : (\exists \sigma \in traces(N), 0 < i < |\sigma| : t' = \sigma_i \wedge t'' = \sigma_{i+1})$, i.e., if two visible tasks a and b are connected via an invisible elementary path, b can directly follow a in some traces.*
3. ***Necessity.*** *$\exists t' \in \{\circ p | p \in t\bullet\}, \exists t'' \in \{p \circ | p \in \bullet t\} : (\forall \sigma \in traces(N), 0 < i < |\sigma| : t' = \sigma_i \Rightarrow t'' \neq \sigma_{i+1})$, i.e., the invisible task cannot be removed directly by merging its input and output places without introducing additional causal dependencies between visible tasks.*

Figure 5 lists some WF-nets containing invisible tasks that are not prime because these WF-nets can be further reduced by simply removing some invisible tasks or being changed to other ones with equivalent behaviors.

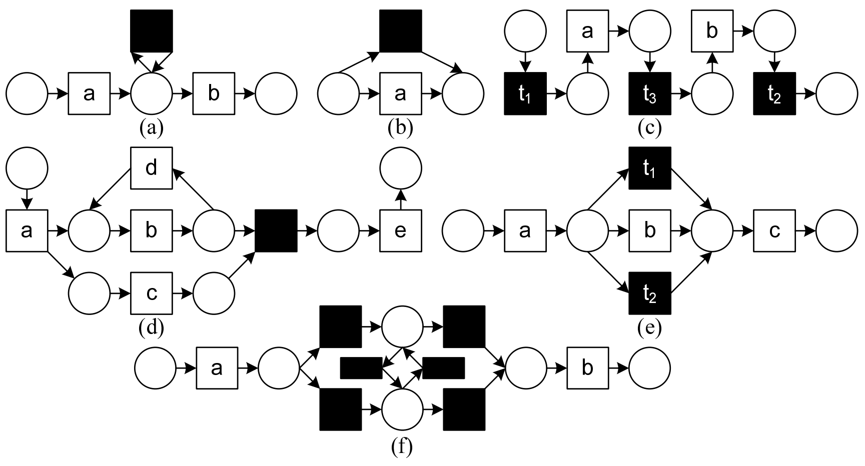

Fig. 5. Some invisible tasks that are not prime

For the WF-net shown in Figure 5(a), the only invisible task violates Requirement 3 and it can be removed directly. The invisible task shown in Figure 5(b) does not obey Requirement 2 and can be removed too. While for the WF-net shown in Figure 5(c), t_1 and t_2 violate requirements 2 and 3 at the same time. But t_3 only violates Requirement 3. The invisible task shown in Figure 5(d) also violates Requirement 3. In Figure 5(e), t_1 and t_2 have the same function and both are prime. But none of them violates requirements 2 and 3. However, one of them should be removed to avoid redundency. The reduction rule will be illustrated in Section 4.2. All invisible tasks in Figure 5(f) are not prime because they all violate Requirement 3.

The problem of mining prime invisible tasks can be formalized as follows.

Definition 6 (Mining problem). *Let $N = (P, T_V, T_{IV}, F)$ be a potential sound IWF-net and $W \subseteq traces(N)$ be an event log of N. The problem of mining prime invisible tasks is to construct an IWF-net $N' = (P', T_V, T'_{IV}, F')$ from W such that N' is behavior equivalent with N with respect to W and contains only minimal prime invisible tasks.*

To tell the truth, the original IWF-net N is not necessary to be present and the log W is enough for a mining algorithm. We can compare the mined model N' and W to determine whether the mining result is good enough.

3.2 Classification of prime invisible tasks

Before detecting prime invisible tasks from event logs, we will first classify prime invisible tasks into several types by their structural features. All types (i.e., SKIP, REDO and SWITCH) of prime invisible tasks were already shown in Figure 3 and Figure 4. The formal definitions are given one by one in the following.

Definition 7 (Invisible task of SKIP type). *Let $N = (P, T_V, T_{IV}, F)$ be an IWF-net with the source place i and the sink place o. For any task $t \in T_{IV}$, t is an invisible task of SKIP type iff there is an elementary path $(n_0, n_1, \ldots, n_k)$ such that $n_0 \neq i$, $n_k \neq o$, $n_0 \in \bullet t$, $n_k \in t\bullet$ and $\forall_{0 \leq i \leq k} : n_i \neq t$. If $k = 2$, t is of SHORT-SKIP type. Otherwise $(k > 2)$, t is of LONG-SKIP type.*

The invisible task in N_2 is of SHORT-SKIP type and the one in N_4 is of LONG-SKIP type. The union of these two subtypes is the SKIP type. Invisible tasks of this type are used to skip the executions of one or more tasks.

Definition 8 (Invisible task of REDO type). *Let $N = (P, T_V, T_{IV}, F)$ be an IWF-net. For any task $t \in T_{IV}$, t is an invisible task of REDO type iff there is an elementary path $(n_0, n_1, \ldots, n_k)$ such that $n_0 \in t\bullet$ and $n_k \in \bullet t$. If $k = 2$, t is of SHORT-REDO type. Otherwise $(k > 2)$, t is of LONG-REDO type.*

The invisible task in N_3 is of SHORT-REDO type and the one in N_5 is of LONG-REDO type. The union of these two subtypes is of the REDO type. Invisible tasks of this type are used to repeat the executions of one or more tasks. A WF-net only containing invisible tasks of SHORT-REDO type may generate behaviors similar (but not the same) to another WF-net only containing length-1-loops. So the new mining algorithm should have the ability to distinguish the two totally different structures but with similar behaviors.

Definition 9 (Invisible task of SWITCH type). *Let $N = (P, T_V, T_{IV}, F)$ be an IWF-net. For any task $t \in T_{IV}$, t is an invisible task of SWITCH type iff there are two elementary paths $(n_0, n_1, \ldots, n_k)$ and $(m_0, m_1, \ldots, m_j)$ such that $\forall_{0<u<k, 0<v<j} : n_u \neq m_v$, $\exists_{0<x<k} : n_x \in \bullet t$, $\exists_{0<y<j} : m_y \in t\bullet$ and there is no elementary path from n_x to m_y.*

The invisible task in N_6 is of SWITCH type and invisible tasks of this type are used to switch the execution chances among multiple alternative branches.

Although the definitions of invisible tasks of SKIP, REDO and SWITCH type are different, they have similar effects on the behaviors of WF-nets. We can see this from the detection method illustrated in the next section.

In fact, there are still some invisible tasks that are not prime but can affect the behavior of an IWF-net, e.g., the invisible task in N_1 in Figure 3. Such an invisible task takes either the source place i as its input or the sink place o as its output but not both (i.e., invisible task of SIDE type). By manually adding a begin event to the begin and an end event to the end of each event trace of a given event log (i.e., pre-processing), such an invisible task can be transformed to a prime one or just be replaced. In post-processing, the manually added tasks can be just removed or reserved as an invisible task. For the added begin task, if it has more than one output arcs or the only output place has more than one input arc in the mined model, it must be reserved. For the added end task, the similar post-processing happens.

4 Detection of prime invisible tasks

In this section, the detection methods for invisible tasks will be introduced. Based on basic ordering relations between tasks, advanced ordering relations for mendacious dependencies between tasks associated with invisible tasks are derived (Subsection 4.1). The reduction rule for identifying redundant mendacious dependencies is given in Subsection 4.2.

4.1 Ordering relations for mendacious dependencies

When there are invisible tasks in process models, the causal dependencies between tasks detected from event logs are not always correct any more. Such dependencies are called *mendacious dependencies*. The most important step of detecting invisible tasks from event logs is identifying all the mendacious dependencies out of the causal dependencies. The basic ordering relations between tasks derived from event logs are first listed below. For a more detailed explanation about these basic ordering relations, readers are referred to [7, 21].

Definition 10 (Basic ordering relations). *Let $N = (P, T_V, T_{IV}, F)$ be a potential sound IWF-net and $W \subseteq traces(N)$ be an event log of N. Let $a, b \in T_V$, then:*

- $a >_W b$ *iff* $\exists \sigma = t_1 t_2 t_3 \cdots t_n \in W, i \in \{1, \dots, n-1\} : t_i = a \wedge t_{i+1} = b$,
- $a \triangle_W b$ *iff* $\exists \sigma = t_1 t_2 t_3 \cdots t_n \in W, i \in \{1, \dots, n-2\} : t_i = t_{i+2} = a \wedge t_{i+1} = b$,
- $a \diamond_W b$ *iff* $a \triangle_W b \wedge b \triangle_W a$,
- $a \rightarrow_W b$ *iff* $a >_W b \wedge (b \not>_W a \vee a \diamond_W b)$,
- $a \#_W b$ *iff* $a \not>_W b \wedge b \not>_W a$, *and*
- $a \parallel_W b$ *iff* $a >_W b \wedge b >_W a \wedge a \not\diamond_W b$.

From Definition 10, it can be seen that $>_W$ and $\triangle_W$ are the most basic ordering relations. All other four ordering relations are based on them. $>_W$ reflects that two tasks can are executed successively and $\triangle_W$ will be used to distinguish length-2-loop from parallel routing. To prove the correctness of the detection method for invisible tasks, it should be assumed that any given event log is complete. Otherwise the minimal ordering relations between tasks cannot be identified successfully. The requirement for the completeness of an event log is the same as the one proposed in [21] (i.e., loop-complete). The definition of completeness is just based on $>_W$ and $\triangle_W$.

Definition 11 (Complete event log). *Let $N = (P, T_V, T_{IV}, F)$ be a potential sound IWF-net and $W \subseteq traces(N)$ be an event log of N. W is complete iff:*

1. *$\forall W' \subseteq traces(N) : >_{W'} \subseteq >_W$,*
2. *$\forall W' \subseteq traces(N) : \triangle_{W'} \subseteq \triangle_W$, and*
3. *$\forall t \in T_V, \exists \sigma \in W : t \in \sigma$.*

Definition 11 shows that completeness does not require an event log to contain all traces that could be generated by the corresponding process model. It only demands that the event log contains all possible basic relations (i.e., $>_W$ and $\triangle_W$) between any pair of tasks and each visible task should appear in some event trace. This completeness notion has shown to be realistic in real-life applications.

Now an advanced ordering relation for mendacious dependencies can be derived from the basic ordering relations. This ordering relation will serve as the basis for detecting invisible tasks of SKIP, REDO, and SWITCH type.

Definition 12 (Advanced ordering relation). *Let $N = (P, T_V, T_{IV}, F)$ be a potential sound IWF-net and $W \subseteq traces(N)$ be an event log of N. For $\forall a, b \in T_V$, $a \rightsquigarrow_W b$ iff $a \rightarrow_W b \land \exists x, y \in T_V : a \rightarrow_W x \land y \rightarrow_W b \land y \not>_W x \land x \nparallel_W b \land a \nparallel_W y$.*

$\rightsquigarrow_W$ reflects the mendacious dependencies associated with invisible tasks of SKIP, REDO and SWITCH types and this kind of ordering relation can be used to construct invisible tasks. Figure 6 illustrates the basic idea behind $\rightsquigarrow_W$. Because of the invisible task t, task a can be directly followed by task b in the log. Hence, existing mining algorithms (such as the α algorithm) try to connect a and b via a place. This leads to incorrect results as shown in figures 3 and 4. Consider for example logs W_2 to W_6 shown in these two figures. For log W_2, we distinguish the mendacious dependency $A \rightsquigarrow_W C$, i.e., A may be followed by C but this cannot be modeled by inserting a place as done by the α algorithm.

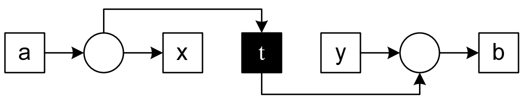

Fig. 6. Illustration for the derivation of $\rightsquigarrow_W$

In Figure 6, there is an invisible task t in the snippet of a WF-net and assume that t can be detected from the corresponding log. The correctness of the detection method corresponding to $\rightsquigarrow_W$ can be proved theoretically as will be shown later. If y is equal to x, t is of SHORT-SKIP type. If y is reachable from x, t is of LONG-SKIP type. If a is equal to b, t is of SHORT-REDO type. If a is reachable from b, t is of LONG-REDO type. Otherwise, t is of SWITCH type, i.e., a to x and y to b are two alternative paths.

To prove the correctness of the detection method for the relationship between an invisible task and a mendacious dependency, we provide the related theorem and derive the mining capacity of the detection method to be a subclass of WF-net named DIWF-net.

Definition 13 (Direct WF-net with prime invisible tasks). *Let* $N = (P, T_V, T_{IV}, F)$ *be an IWF-net with the source place i and the sink place o. N is a direct WF-net with prime invisible tasks (DIWF-net for short) iff:*

1. $\forall a, b \in T_V \bigcup T_{IV} : a\bullet \cap \bullet b \neq \emptyset \Rightarrow \exists M \in [N, [i]\rangle : (N, M)[a\rangle \wedge (N, M - \bullet a \cap a\bullet)[b\rangle$, *i.e., if a and b are connected via a place, there should be firing sequences such that a is directly followed by b.*
2. *There are no invisible tasks that are not prime and implicit places, i.e., it should not be possible to remove tasks or places without changing the observable behavior of the IWF-net.*

Definition 13 limits the capacity of the above detection method to a reasonable subclass of WF-net. The mendacious dependencies in non-DIWF-nets may not be detected correctly. As a result, our algorithm may fail to find a sound WF-net covering the given log or return a DIWF-net with equivalent behaviors but different structures compared to the original WF-net. Figure 7 shows four sound WF-nets that are not DIWF-nets.

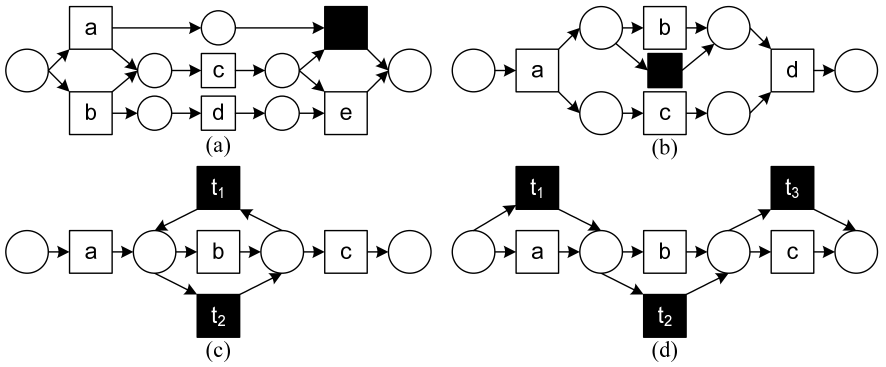

Fig. 7. Some non-DIWF-nets containing invisible tasks

The invisible task in Figure 7(a) cannot be detected correctly because of the non-free-choice constructs involving it and e, which leads to that the invisible task cannot follow a directly. The mendacious dependency $a \rightsquigarrow_W d$ in Figure 7(b) cannot be detected because a is never directly followed by d. There are DIWF-nets containing a length-1-loop about b, which have equivalent behaviors with those of the WF-net in Figure 7(c). The essential reason is that the two invisible tasks t_1 and t_2 are not prime. A similar observation holds for the WF-net in Figure 7(d). There are DIWF-nets having more invisible tasks than this WF-net but with equivalent behaviors.

The goal of $\rightsquigarrow_W$ is to detect the presence of prime invisible tasks. The next theorem shows that this is indeed the case for DIWF-nets. It is assumed that the invisible tasks of SIDE type have been detected and constructed successfully by now using the method illustrated at the end of Section 3.2.

Theorem 1. *Let $N = (P, T_V, T_{IV}, F)$ be a sound DIWF-net, W be a complete event log of N and $a, b \in T_V$ be two visible tasks. There is a prime invisible task $t \in T_{IV}$ such that $a \bullet \cap \bullet t \neq \emptyset$ and $t \bullet \cap \bullet b \neq \emptyset$ iff $a \rightsquigarrow_W b$.*

Proof. The theorem is proven to be correct in both directions.

1. Assume that there exists a prime invisible task $t \in T$ such that $a \bullet \cap \bullet t \neq \emptyset$ and $t \bullet \cap \bullet b \neq \emptyset$. We need to prove that $a \rightsquigarrow_W b$ holds. Let $p_{in} \in a \bullet \cap \bullet t, p_{out} \in t \bullet \cap \bullet b$. According to Requirement 2 in Definition 2, $a >_W b$ holds. Assume $a \parallel_W b$, there must be a marking M such that $M \in [N, [i]\rangle$ and $(N, M)[a\rangle$ and $(N, M)[b\rangle$ and $(N, M - \bullet a + a\bullet)[t\rangle$. Because $\bullet a \cap \bullet b = \emptyset$ (assuming $a \parallel_W b$) and $\bullet t \cap \bullet b = \emptyset$ (Requirement 3 in Definition 5), after t fires in marking $M - \bullet a + a\bullet$, there will be two tokens in p_{out} which violates the notion of soundness assumed here. Hence we get a contradiction and $a \parallel_W b$ cannot hold. As a result, $a \rightarrow_W b$ holds because $a >_W b$ and $a \nparallel_W b$. Because t is prime, $|p_{in} \bullet| > 1$ and $|\bullet p_{out}| > 1$ hold (Requirement 3 in Definition 5). There will be at least a visible task x such that $p_{in} \in \bullet x$ or an invisible task t' such that $t' \neq t$ and $p_{in} \in \bullet t'$. In the latter case, there will be at least a visible task x' such that $t' \bullet \cap \bullet x' \neq \emptyset$ (Requirement 1 in Definition 5). Similarly, there will be at least a visible task y such that $p_{out} \in y\bullet$ or an invisible task t'' such that $t'' \neq t$ and $p_{out} \in t''\bullet$. In the latter case, there will be at least a visible task y' such that $y' \bullet \cap \bullet t'' \neq \emptyset$. Here we only prove the simplest case, i.e., $p_{in} \in \bullet x$ and $p_{out} \in y\bullet$. Similar to the proof of $a \rightarrow_W b$, in all the four cases, $a \rightarrow_W x$, $y \rightarrow_W b$, $a \nparallel_W y$, $x \nparallel_W b$ and $y \nparallel_W x$ hold. Now we still need to prove $y \not\rightarrow_W x$. Assume that there is a causal relation between any p_{out}'s input transition y and any p_{in}'s output transition x $(x \neq t)$, i.e., $y \rightarrow_W x$. According to Requirement 3 in Definition 5, the assumption does not hold. Hence, we have shown that $a \rightarrow_W b$ and $\exists_{x,y \in T} a \rightarrow_W x \wedge y \rightarrow_W b \wedge y \not>_W x \wedge x \nparallel_W b \wedge a \nparallel_W y$.
2. Assume $a \rightsquigarrow_W b$ holds, i.e., $a \rightarrow_W b$ and there exist two tasks x, y such that $a \rightarrow_W x$, $y \rightarrow_W b$, $y \not>_W x$, $x \nparallel_W b$ and $a \nparallel_W y$. We now need to prove that there exists an invisible task $t \in T$ such that $a \bullet \cap \bullet t \neq \emptyset$ and $t \bullet \cap \bullet b \neq \emptyset$. Assume there is no invisible task between any of a's output places and any of b's input places, i.e., we try to obtain a contradiction. If there is no such invisible task, $a \rightarrow_W b$ implies $a \bullet \cap \bullet b \neq \emptyset$ (Theorem 4.1 in [7]). $a \rightarrow_W x$ implies $a \bullet \cap \bullet x \neq \emptyset$ or there is an invisible elementary path from a's output places to x's input places. Here we only prove the simplest case, i.e., $a \bullet \cap \bullet x \neq \emptyset$. Similarly, $y \rightarrow_W b$ implies $y \bullet \cap \bullet b \neq \emptyset$. Because

$y \not>_W x$, it can be concluded that $y \bullet \cap \bullet x = \emptyset$. Thus there will be four basic constructs to reflect the above-mentioned features, which are listed in Figure 8. Now we will show that all these constructs have various issues. In Figure 8(a), $a \#_W y$ does not hold. If this is not the case, $a \rightarrow_W b$, $y \rightarrow_W b$ and $a \#_W y$ will imply $a \bullet \cap y \bullet \cap \bullet b \neq \emptyset$ (Theorem 4.4 in [7]). In this case, either $a \rightarrow_W y$ or $y \rightarrow_W a$ holds ($a \not\parallel_W y$ and not $a \#_W y$). If $a \rightarrow_W y$ holds, there will be a place connecting a and y. After a is executed, y will always occur before b. Hence we get a contradiction. Similarly, if $y \rightarrow_W a$ holds, we can get a contradiction with $y \rightarrow_W b$. As a result, the construct in Figure 8(a) is not sound. Similarly, the other two constructs in Figure 8(b) and (d) are not sound either. In Figure 8(c), after y is executed, b cannot occur directly unless there the place connecting a, y and b is not safe. This violates $y \rightarrow_W b$ and we still get a contradiction. The assumption that $a \rightsquigarrow_W b$ and there is no invisible task connecting a and b is wrong and the construct can only be similar to the one shown in Figure 6.

Hence, the theorem is proven to be correct in both directions. □

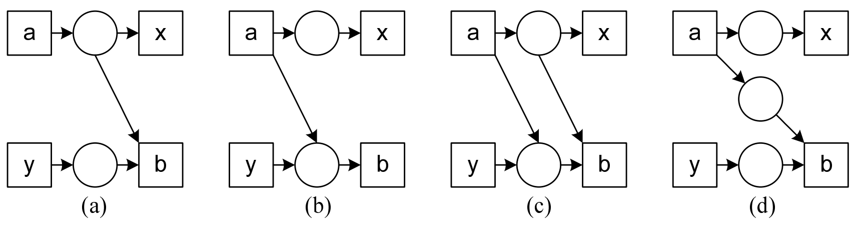

Fig. 8. Possible constructs related to $a \rightsquigarrow_W b$ not containing invisible tasks

After detecting all mendacious dependencies between tasks, the real causal dependencies should be distinguished, which is defined below.

Definition 14 (Real causal dependency). *Let $N = (P, T_V, T_{IV}, F)$ be a potential sound IWF-net and $W \subseteq traces(N)$ be an event log of N. For $a, b \in T_V$, $a \mapsto_W b$ iff $a \rightarrow_W b$ and $a \not\rightsquigarrow_W b$.*

4.2 Identifying redundant mendacious dependencies

Not all the mendacious dependencies detected from event logs are meaningful to the mined process model. There may be some mendacious dependencies leading to redundant invisible tasks, which are called *redundant mendacious dependencies*. One reduction rule is proposed in this subsection to identify such mendacious dependencies so as to separate them from those necessary ones.

Figure 9 shows two WF-nets (i.e., N_7 and N_8) involving redundant invisible tasks. In both N_7 and N_8, the function of t_3 can be replaced by the combinational function of t_1 and t_2. When mining process models from W_7 and W_8, whether constructing t_3 depends on user decision (by user-defined parameter). From the

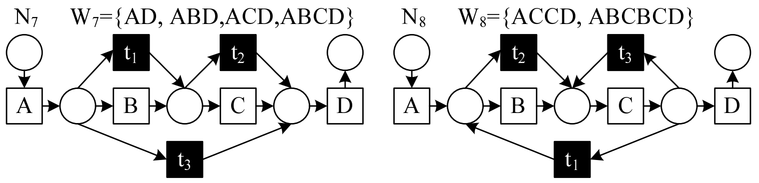

Fig. 9. WF-nets involving redundant invisible tasks

semantics of a process model, the redundant invisible tasks may be necessary when they involve multiple parallel branches.

$A \rightsquigarrow_W C$, $B \rightsquigarrow_W D$ and $A \rightsquigarrow_W D$ can be derived from W_7 and $A \rightsquigarrow_W D$ is redundant. Similarly, $C \rightsquigarrow_W B$, $A \rightsquigarrow_W C$ and $C \rightsquigarrow_W C$ can be derived from W_8 and $C \rightsquigarrow_W C$ is redundant. The following reduction rule referred to as Rule 1 hereafter can be used to identify such dependencies.

$$\begin{aligned}&\forall a, b \in T_W : (a \rightsquigarrow_W b \wedge \exists c, d \in T_W : (c \rightarrow_W d \wedge a \rightsquigarrow_W d \wedge c \rightsquigarrow_W b))\\&\Rightarrow a \rightsquigarrow_W b \; is \; redundant\end{aligned} \quad (1)$$

In Rule 1, W is an event log and $T_W = \{t|\exists \sigma \in W : t \in \sigma\}$. This rule implies that all mendacious dependencies which can be combined by other ones are redundant. $A \rightsquigarrow_W D$ and $C \rightsquigarrow_W C$ prove to be redundant, which are derived from W_7 and W_8 respectively in Figure 9.

5 The mining algorithm $\alpha^{\#}$

This section first analyzes how to construct invisible tasks from mendacious dependencies. Then the mining algorithm for constructing the mined process model is introduced in detail.

5.1 Construction of invisible tasks

For process models containing only causal relations between tasks, there is a one-to-one relationship between invisible tasks and mendacious dependencies. However, this is not always the case because selective and parallel relations are so common in real-life processes. Constructing invisible tasks is not such a trivial task. See Figure 10 for detail explanation.

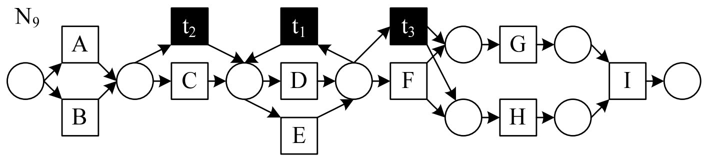

Fig. 10. The one-to-many relationship between invisible tasks and $\rightsquigarrow_W$

The process model N_9 is a sound DIWF-net and one of its complete log is $W_9 = \{ACDDFGHI, BCEEFHGI, ADEDEGHI, AEDGHI, BEDHGI, BD$

$EHGI\}$. t_1 corresponds to $D \rightsquigarrow_W D$, $D \rightsquigarrow_W E$, $E \rightsquigarrow_W D$ and $E \rightsquigarrow_W E$. Similar things happen to t_2 and t_3. On the contrary, there are situations where multiple invisible tasks correspond to one mendacious dependency. For a variation of the DIWF-net shown in Figure 10, $F \rightsquigarrow_W I$ would correspond to two parallel invisible tasks skipping G and H respectively.

The algorithm for constructing prime invisible tasks will be given below, which is the core of the $\alpha^{\#}$ algorithm. The operators about the relations between a task and an event log (i.e., $\in$, $first$ and $last$) are borrowed from [7] directly. The two functions $PreSet$ and $PostSet$ are used to construct the input and output places of a task. When generating the places here, the $\rightarrow_W$ relations related to mendacious dependencies will not be considered because they do not reflect real causal dependencies.

Definition 15 (Construction algorithm ConIT). *Let $N = (P, T_V, T_{IV}, F)$ be a potential sound IWF-net and $W \subseteq traces(N)$ be a complete event log of N. ConIT(W) that is used to construct prime invisible tasks is defined as follows.*

1. $T_W = \{t \in \sigma | \sigma \in W\}$,
2. $T_I = \{first(\sigma) | \sigma \in W\}$,
3. $T_O = \{last(\sigma) | \sigma \in W\}$,
4. $D_M = \{(a, b) | a \in T_W \wedge b \in T_W \wedge a \rightsquigarrow_W b\}$,
5. $R_M = \{(a, b) \in D_M | (a, b) \text{ is redundant}\}$,
6. $D_M = D_M - R_M$,
7. $X_I = \{(P_{in}, P_{out}) | (\forall (A, X) \in P_{in}, (Y, B) \in P_{out} : (\forall a \in A, b \in B : (a, b) \in D_M \wedge (A, X) \in PostSet(a) \wedge (Y, B) \in PreSet(b)) \wedge (\forall x \in X, y \in Y : x \nparallel_W y)) \wedge (\forall (A_1, X_1), (A_2, X_2) \in P_{in} : (\exists a_1 \in A_1, a_2 \in A_2 : a_1 \parallel_W a_2)) \wedge (\forall (Y_1, B_1), (Y_2, B_2) \in P_{out} : (\exists b_1 \in B_1, b_2 \in B_2 : b_1 \parallel_W b_2))\}$,
8. $Y_I = \{(P_{in}, P_{out}) \in X_I | \forall (P'_{in}, P'_{out}) \in X_I : P_{in} \subseteq P'_{in} \wedge P_{out} \subseteq P'_{out} \Rightarrow (P_{in}, P_{out}) = (P'_{in}, P'_{out})\}$,
9. $X'_I = \{(P_{in}, P_{out}) | (\forall (A, X) \in P_{in}, (Y, B) \in P_{out} : (\forall a \in A, b \in B : (a, b) \in R_M \wedge (A, X) \in PostSet(a) \wedge (Y, B) \in PreSet(b)) \wedge (\forall x \in X, y \in Y : x \nparallel_W y)) \wedge (\forall (A_1, X_1), (A_2, X_2) \in P_{in} : (\exists a_1 \in A_1, a_2 \in A_2 : a_1 \parallel_W a_2)) \wedge (\forall (Y_1, B_1), (Y_2, B_2) \in P_{out} : (\exists b_1 \in B_1, b_2 \in B_2 : b_1 \parallel_W b_2))\}$,
10. $Y'_I = \{(P_{in}, P_{out}) \in X'_I | (\nexists (P_{in1}, P_{out1}), \ldots, (P_{ink}, P_{outk}) \in Y_I \cup X'_I, k > 1 : P_{in} \cap P_{in1} \neq \emptyset \wedge P_{out1} \cap P_{in2} \neq \emptyset \wedge \ldots \wedge P_{outk} \cap P_{out} \neq \emptyset) \wedge (\forall (P'_{in}, P'_{out}) \in X'_I : P_{in} \subseteq P'_{in} \wedge P_{out} \subseteq P'_{out} \Rightarrow (P_{in}, P_{out}) = (P'_{in}, P'_{out}))\}$,
11. $D_S = \{(t_{(P_{in}, P_{out})}, t_{(P'_{in}, P'_{out})}) | (P_{in}, P_{out}), (P'_{in}, P'_{out}) \in Y_I \cup Y'_I \wedge P_{out} \cap P'_{in} \neq \emptyset\} \cup \{(a, t_{(P_{in}, P_{out})}) | (P_{in}, P_{out}) \in Y_I \cup Y'_I \wedge \exists (A, X) \in P_{in} : a \in A\} \cup \{(t_{(P_{in}, P_{out})}, b) | (P_{in}, P_{out}) \in Y_I \cup Y'_I \wedge \exists (Y, B) \in P_{out} : b \in B\}$,
12. $D_P = \{(t_{(P_{in}, P_{out})}, t_{(P'_{in}, P'_{out})}) | (P_{in}, P_{out}), (P'_{in}, P'_{out}) \in Y_I \cup Y'_I \wedge \forall (A, X) \in P_{in}, (A', X') \in P'_{in} : \exists a \in A, a' \in A', x \in X, x' \in X' : a \parallel_W a' \vee x \parallel_W x'\} \cup \{(t, t_{(P_{in}, P_{out})}) | t \in T_W \wedge (P_{in}, P_{out}) \in Y_I \cup Y'_I \wedge \forall (A, X) \in P_{in} : \exists a \in A, x \in X : a \parallel_W t \vee x \parallel_W t\} \cup \{(t_{(P_{in}, P_{out})}, t) | t \in T_W \wedge (P_{in}, P_{out}) \in Y_I \cup Y'_I \wedge \forall (A, X) \in P_{in} : \exists a \in A, x \in X : a \parallel_W t \vee x \parallel_W t\}$,
13. $T_W = T_W \cup \{t_{(P_{in}, P_{out})} | (P_{in}, P_{out}) \in Y_I \cup Y'_I\}$, *and*
14. $ConIT(W) = (T_W, T_I, T_O, D_S, D_P)$.

The algorithm *ConIT* works as follows. Steps 1, 2 and 3 are borrowed from [7, 21] directly. They are used to construct the sets of all tasks, first tasks and last tasks, i.e., T_W, T_I and T_O. All mendacious dependencies between tasks are detected and the redundant ones are identified and excluded in steps 4 to 6. Step 7 to Step 10 are used to construct invisible tasks of SKIP/REDO/SWITCH types (stored in Y_I and Y_I') reflected by the mendacious dependencies. These four steps are the most important ones in the whole algorithm. The only difference between Step 7 and Step 9 is that the latter needs to check that no new invisible task can be composed by other ones. In steps 11 and 12, new causal and parallel relations between invisible tasks as well as the ones between invisible tasks and visible tasks are added. Finally, the task set T_W are extended by new constructed invisible tasks in Step 13 and Step 14 returns the necessary results.

5.2 Construction of the mined process model

Based on the algorithms proposed in the above subsection, the mining algorithm named $\alpha^{\#}$ can be defined as follows. It returns the mined model in DIWF-net.

Definition 16 (Mining algorithm $\alpha^{\#}$). *Let W be a loop-complete event log over a task set T (i.e., $W \subseteq T^*$). $\alpha^{\#}(W)$ is defined as follows.*

1. $(T_W, T_I, T_O, D_S, D_P) = ConIT(W)$,
2. $X_W = \{(A,B) | A \subseteq T_W \wedge B \subseteq T_W \wedge (\forall a \in A, b \in B : a \mapsto_W b \vee (a,b) \in D_S) \wedge (\forall a_1, a_2 \in A : (a_1 \#_W a_2 \wedge (a_1,a_2) \notin D_P) \vee (a_1 \mapsto_W a_2 \wedge a_2 >_W a_2) \vee (a_2 \mapsto_W a_1 \wedge a_1 >_W a_1)) \wedge (\forall b_1, b_2 \in B : (b_1 \#_W b_2 \wedge (b_1,b_2) \notin D_P) \vee (b_1 \mapsto_W b_2 \wedge b_1 >_W b_1) \vee (b_2 \mapsto_W b_1 \wedge b_2 >_W b_2))\}$,
3. $Y_W = \{(A,B) \in X_W | \forall (A',B') \in X_W : A \subseteq A' \wedge B \subseteq B' \Rightarrow (A,B) = (A',B')\}$,
4. $P_W = \{P_{(A,B)} | (A,B) \in Y_W\} \cup \{i_W, o_W\}$,
5. $F_W = \{(a, P_{(A,B)}) | (A,B) \in Y_W \wedge a \in A\} \cup \{(P_{(A,B)}, b) | (A,B) \in Y_W \wedge b \in B\} \cup \{(i_W, t) | t \in T_I\} \cup \{(t, o_W) | t \in T_O\}$, *and*
6. $N_W = (P_W, T_W, F_W)$.

The $\alpha^{\#}$ algorithm is relatively simple and easy to understand, which works as follows. Step 1 invokes the algorithm *ConIT* to construct all invisible tasks and fix the causal/parallel relations between tasks. All pairs of task sets related to possible places are constructed in Step 2. This step takes into account invisible tasks and length-one-loop tasks at the same time. Steps 3 to 6 are directly borrowed from [7], in which the places together with the connecting arcs are constructed and the mined process model in WF-net is returned.

To illustrate the $\alpha^{\#}$ algorithm, we show the result of each step using the log W_9. The original model corresponding to W_9 is shown in Figure 10. Here we only concentrate on the steps in *ConIT* because there is no invisible tasks of SIDE type and the steps in $\alpha^{\#}$ algorithm is straightforward.

1. $T_W = \{A, B, C, D, E, F, G, H, I\}$,
2. $T_I = \{A, B\}$,

3. $T_O = \{I\}$,
4. $D_M = \{(A, D), (A, E), (B, D), (B, E), (D, D), (D, E), (D, G), (D, H), (E, D), (E, E), (E, G), (E, H)\}$,
5. $R_M = \emptyset$,
6. $D_M = D_M$,
7. $X_I = \{(\{(\{A\}, \{C\})\}, \{(\{C\}, \{D\})\}), (\{(\{A\}, \{C\})\}, \{(\{C\}, \{E\})\}), (\{(\{B\}, \{C\})\}, \{(\{C\}, \{D\})\}), (\{(\{B\}, \{C\})\}, \{(\{C\}, \{E\})\}), \ldots, (\{(\{A, B\}, \{C\})\}, \{(\{C\}, \{D, E\})\}), (\{(\{D\}, \{F\})\}, \{(\{C\}, \{D\})\}), (\{(\{E\}, \{F\})\}, \{(\{C\}, \{E\})\}), \ldots, (\{(\{D, E\}, \{F\})\}, \{(\{C\}, \{D, E\})\}), (\{(\{D\}, \{F\})\}, \{(\{F\}, \{G\})\}), \ldots, (\{(\{E\}, \{F\})\}, \{(\{F\}, \{H\})\}), (\{(\{D, E\}, \{F\})\}, \{(\{F\}, \{G\}), (\{F\}, \{H\})\})\}$,
8. $Y_I = \{t_2 = (\{(\{A, B\}, \{C\})\}, \{(\{C\}, \{D, E\})\}), t_1 = (\{(\{D, E\}, \{F\})\}, \{(\{C\}, \{D, E\})\}), t_3 = (\{(\{D, E\}, \{F\})\}, \{(\{F\}, \{G\}), (\{F\}, \{H\})\})\}$,
9. $X_I' = \emptyset$,
10. $Y_I' = \emptyset$,
11. $D_S = \{(A, t_2), (B, t_2), (t_2, D), (t_2, E), (D, t_1), (E, t_1), (t_1, D), (t_1, E), (D, t_3), (E, t_3), (t_3, G), (t_3, H)\}$,
12. $D_P = \emptyset$,
13. $T_W = \{A, B, C, D, E, F, G, H, I, t_1, t_2, t_3\}$, and
14. $ConIT(W) = (T_W, T_I, T_O, D_S, D_P)$.

Comparing the above results with the DIWF-net N_9 shown in Figure 10, we can find that all the three invisible tasks are correctly detected. The ordering relations among new invisible tasks as well as the ones between invisible tasks and visible tasks are established successfully too. If these results are taken as the input of the $\alpha^{\#}$ algorithm, the mined model will be the same as N_9.

6 Experimental evaluation of the work

First we introduce the implementation of the $\alpha^{\#}$ algorithm (Subsection 6.1). Then the evaluation criteria for conformance testing is illustrated (Subsection 6.2). Thirdly, evaluation results are explained in detail in Subsection 6.3. Finally, the limitations of the $\alpha^{\#}$ algorithm are discussed in Subsection 6.4.

6.1 Implementation of the $\alpha^{\#}$ algorithm

The $\alpha^{\#}$ algorithm has been implemented as a mining plug-in of ProM [3, 13] and can be downloaded from www.processmining.org. The current version of ProM is 5.0 and it is an open-source extensive framework for process mining, which is implemented in Java. Currently, there are already more than 210 mining, analysis and conversion plug-ins. It takes an event log in an extensible, XML-based format (i.e., MXML) as input and uses a process mining plug-in to mine a process model from that log. The mined process model will be shown to the end user graphically. A snapshot of ProM is given in Figure 11, which shows an DIWF-net constructed by the $\alpha^{\#}$ algorithm. The mining result can also be converted to an Event-driven Process Chain and be analyzed for soundness by analysis plug-in. Furthermore, the result can be exported to tools such as CPN Tools, ARIS, YAWL, ARIS PPM, Yasper, EPC Tools, woflan, etc.

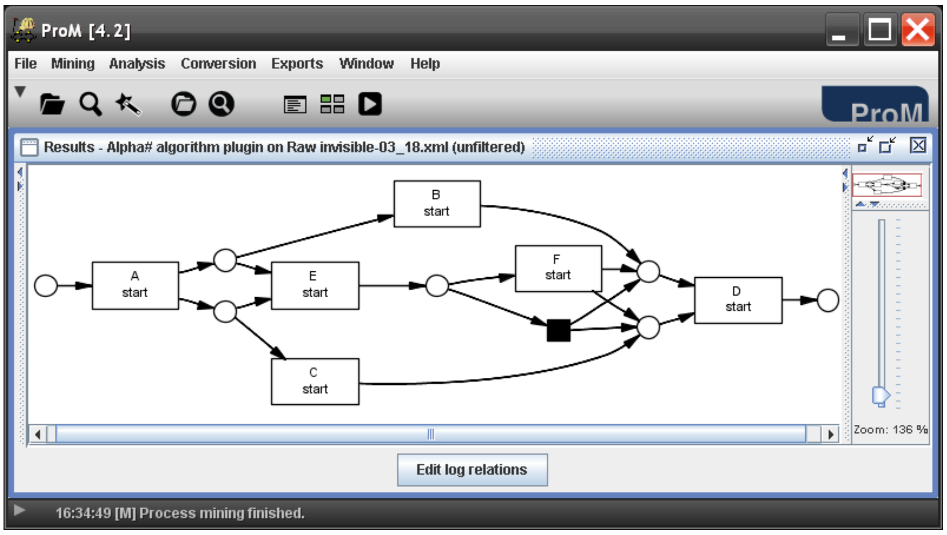
ProM [4.2]
File Mining Analysis Conversion Exports Window Help
ProM
Results - Alpha# algorithm plugin on Raw invisible-03_18.xml (unfiltered)
A start
B start
C start
D start
E start
F start
Zoom: 136 %
Edit log relations
16:34:49 [M] Process mining finished.

Fig. 11. The snapshot of ProM when mining a process model using the $\alpha^{\#}$ plug-in

6.2 Evaluation criteria

Although visual inspection of the mined model and the original model can be used to see whether the mining result is correct, there are also some problems related to this visual inspection. Firstly, it works well only for small examples. Secondly, we cannot assume that the original model is always present. Finally, the original model and the mined model may have different structures but have exactly the same behaviors. Therefore the following metrics are introduced, which are used to test the conformance between the mined model and the given log[24].

The metric f is determined by replaying the log in the model, i.e., for each case the "token game" is played as suggested by the log. For this, the replay of every event trace starts with marking the initial place in the model and then the transitions that belong to the logged events in the trace are fired one after another. While doing so, one counts the number of tokens that had to be created artificially (i.e., the transition belonging to the logged event was not enabled and therefore could not be *successfully executed*) and the number of tokens that were left in the model (they indicate that the process has not *properly completed*). Only if there were neither tokens left nor missing, the fitness measure evaluates to 1.0, which indicates 100% fitness. In other words, fitness reflects the extent to which the event traces can be associated with execution paths specified by the process model. Thus if $f = 1$ then the log can be parsed by the model without any error. The token-based fitness metric f is formalized as follows:

$$f = \frac{1}{2}\left(1 - \frac{\sum_{i=1}^{k} n_i m_i}{\sum_{i=1}^{k} n_i c_i}\right) + \frac{1}{2}\left(1 - \frac{\sum_{i=1}^{k} n_i r_i}{\sum_{i=1}^{k} n_i p_i}\right) \qquad (2)$$

Here k is the number of different traces from the aggregated log. For each event trace i ($1 \leq i \leq k$): n_i is the number of process instances combined into the current trace, m_i is the number of missing tokens, r_i is the number of remaining tokens, c_i is the number of consumed tokens, and p_i is the number of produced tokens during log replay of the current trace. Note that for all i, $m_i \leq c_i$ and $r_i \leq p_i$, and therefore $0 \leq f \leq 1$. The maximum value of the fitness metric will be used as an evaluation criteria, i.e., $f = 1$.

The other two conformance testing metrics are aB (*behavioral appropriateness*) and aS (*structural appropriateness*). Appropriateness reflects the degree of accuracy in which the process model describes the observed behavior (i.e., aB), combined with the degree of clarity in which it is represented (i.e., aS). For all the three metrics, their values are between 0.0 and 1.0. For any successful mining, the value of f should be 1.0 and the values of aB and aS should be as big as possible. Another important evaluation criteria is that the mined model should be sound. From the viewpoint of practical application, the soundness of any process model is a necessary requirement.

6.3 Evaluation results

In an experimental setting, logs can be obtained in three ways: (1) as a download or conversion from an operational information system (i.e., a real log), (2) a manually created or collected log, and (3) a log resulting from a simulation. For evaluation of the $\alpha^{\#}$ algorithm, we have used all three possibilities. In this section, we show the results of our experimental evaluation of the $\alpha^{\#}$ algorithm.

A lot of experiments have been done to evaluate the proposed methods together with the implemented algorithm. The $\alpha^{\#}$ plug-in of ProM has been applied to several real-life logs and many smaller artificial logs. There are totally 96 artificial examples in DIWF-nets evaluated. The corresponding complete logs are generated manually. The maximum number of tasks in one process model is less than 20 and the number of cases in one event log is less than 30. All types of invisible tasks are modeled in a separate or combined manner. Of all the examples, 24 models involve invisible tasks of SIDE type, 12 models involve length-1-loops, length-2-loops and invisible tasks of SHORT-REDO type, and 4 models do not contain any invisible task (the corresponding logs are *L23*, *L24*, *L48*, and *L54*). The conformance testing results are shown in Figure 12.

From Figure 12, we can see that all 96 models are mined successfully from their corresponding logs (i.e., $f = 1$). Given the 96 complete event logs generated by different DIWF-nets, the correct rate of mining process models from logs by the $\alpha^{\#}$ algorithm approaches 100%. While for the α algorithm, the correct rate is only $4/96 \approx 4.2\%$. Only the 4 models that do not contain any invisible task were mined successfully. It shows that the $\alpha^{\#}$ algorithm is a good extension of the α algorithm to handle invisible tasks. At the same time, the $\alpha^{\#}$ algorithm can distinguish invisible tasks of SHORT-REDO type and length-1-loops correctly.

Ten real-life logs are obtained from Kinglong Company in Xiamen, Fujian province, China, which are all about processes for routing engineering document. The mined process model from $L10$ is shown in Figure 13. The source place and the sink place are not visible in this figure for space limitation. All the conformance testing results are shown in Table 2. It is obvious that all the

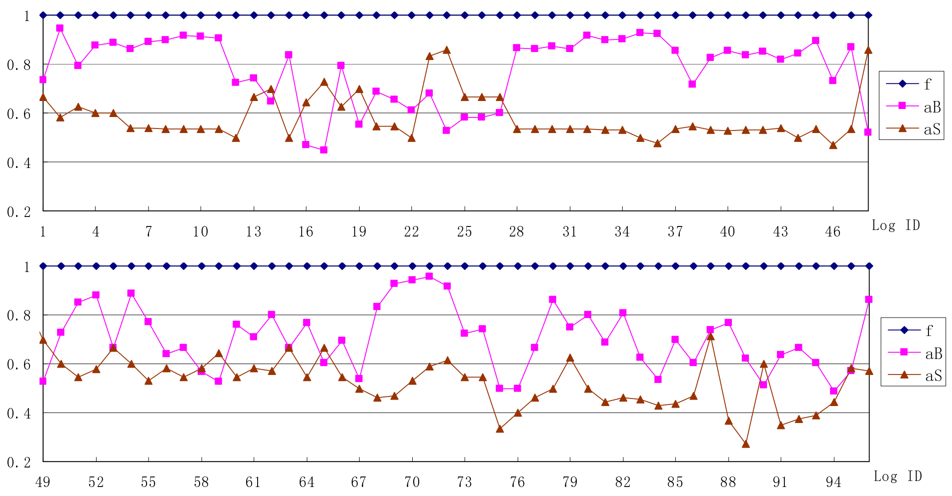

Fig. 12. Evaluation of $\alpha^{\#}$ algorithm using 96 artificial logs

experiments are successful. Compared with the given logs, the mining results of the $\alpha^{\#}$ will not introduce new ordering relations between invisible tasks. The proportion for invisible tasks out of all tasks is 77/(77+88)=46.7%. This proportion is very near to that computed from the process models in Section 1 and verify the correctness of the $\alpha^{\#}$ to some extent.

Table 2. Conformance testing results based on real-life logs: f-fitness,aB-behavioral appropriateness,aS-structural appropriateness,NoI-the number of invisible tasks,NoC-the number of cases,NoE-the number of events,NoT-the number of visible tasks

	$L1$	$L2$	$L3$	$L4$	$L5$	$L6$	$L7$	$L8$	$L9$	$L10$
f	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
aB	0.823	0.983	0.876	0.902	0.811	0.933	0.791	0.953	0.804	0.809
aS	0.321	0.5	0.375	0.5	0.45	0.478	0.286	0.529	0.391	0.271
NoI	10	3	9	3	5	4	14	2	7	20
NoC	8	6	5	11	40	42	30	42	297	44
NoE	43	52	59	84	324	469	221	288	2020	531
NoT	7	15	10	7	7	9	8	7	7	11

In Section 2, we have shown that the GA algorithm can mine process models with invisible tasks too. However, the main goal of GA algorithm is to find a suitable process model to fit the given event log. Besides invisible tasks, it can also handle duplicate tasks, non-free-choice constructs and noise. It is based on the generic algorithm and has more than ten parameters. Because the $\alpha^{\#}$ algorithm do not need any parameter, when mining an event log using the GA algorithm, all default values of the parameters will be used. Many previous example logs (i.e., 96 artificial ones and 10 real-life ones) can be mined successfully by the GA algorithm (i.e., $f = 1$). However, there are still many logs on which

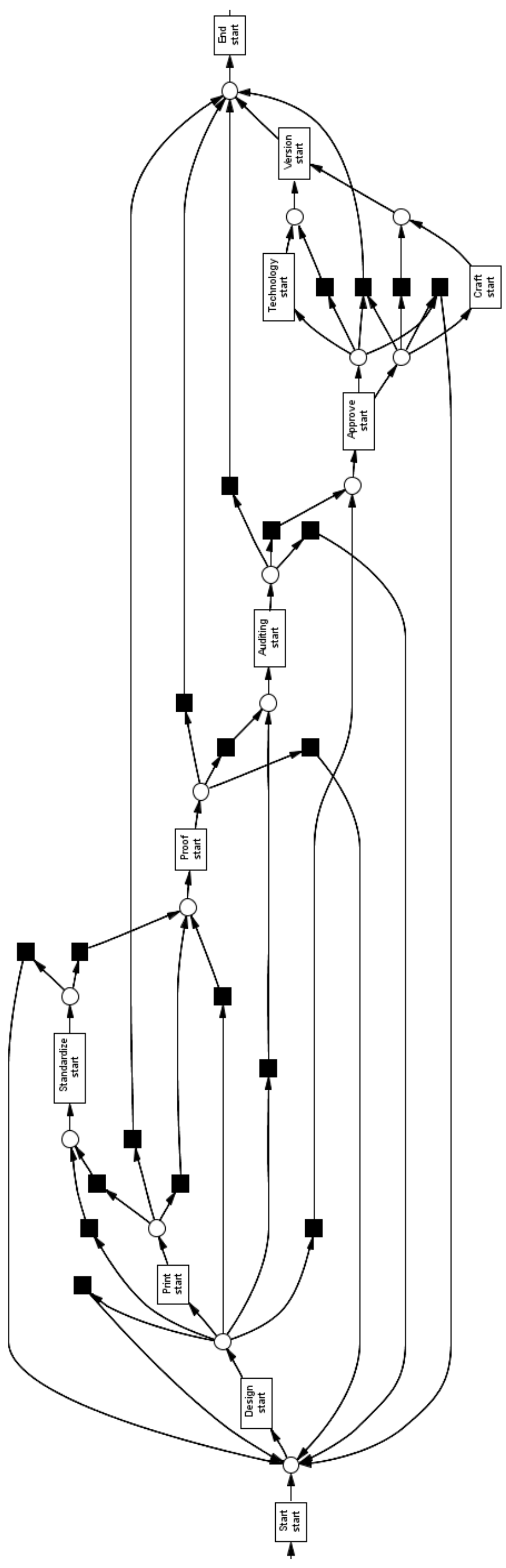

Fig. 13. A real-life process model mined from L10 by the $\alpha^{\#}$ algorithm

the $\alpha^{\#}$ algorithm performs better than the GA algorithm. Figure 14 lists the comparison results of the $\alpha^{\#}$ algorithm and the GA algorithm. All the testing logs are chosen from the above artificial and real-life logs.

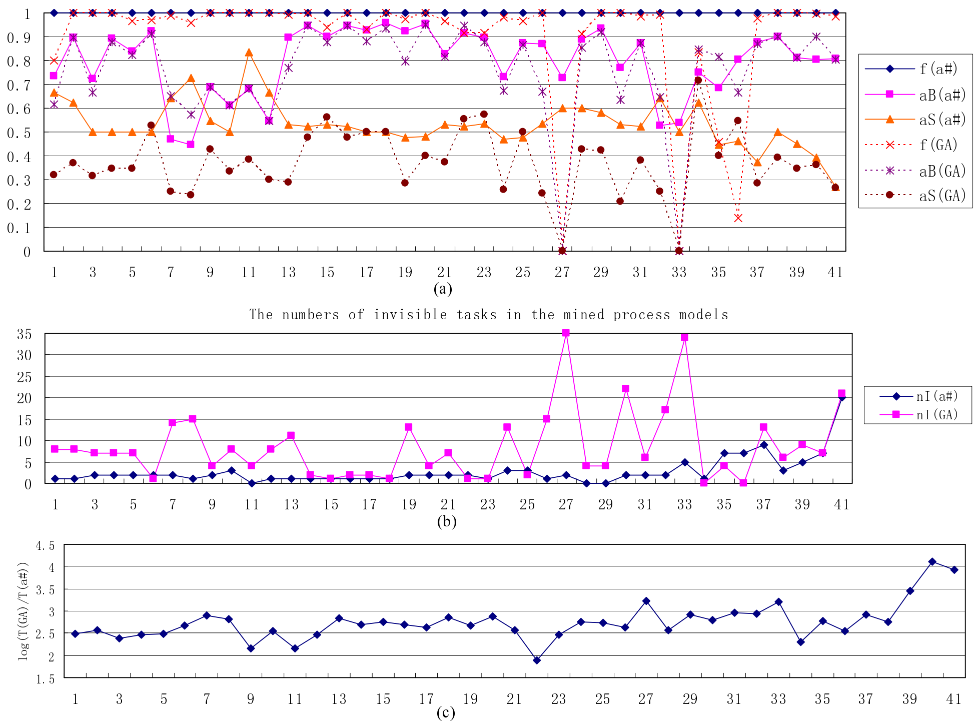

Fig. 14. Comparison results of the $\alpha^{\#}$ algorithm and the GA algorithm

Three perspectives are concerned when comparing the mining results of the $\alpha^{\#}$ algorithm and the GA algorithm, which are explained as follows:

1. The values of f, aB and aS between the each mined model and its corresponding log. The importance priorities of the three metrics from big to small are f, aB and aS. The bigger the value of f is, the better the result is. If $f = 1$ is true for both algorithm, aB and aS will be considered. From Figure 14(a), we can see that on the chosen 41 example logs, the $\alpha^{\#}$ algorithm performs better than the GA algorithm. Whether $f_{\alpha^{\#}} = 1 > f_{GA}$ or $f_{\alpha^{\#}} = f_{GA} = 1 \wedge aB_{\alpha^{\#}} > aB_{GA}$ or $f_{\alpha^{\#}} = f_{GA} = 1 \wedge aB_{\alpha^{\#}} = aB_{GA} \wedge aS_{\alpha^{\#}} > aS_{GA}$ is true. The GA algorithm are difficult to handle event logs generated by the process models containing some special constructs, such as length-1-loops, invisible tasks of SIDE type, invisible tasks of SKIP type between AND-split task and AND-join task, invisible tasks of SHORT-REDO type, invisible tasks involved in non-free-choice constructs, invisible tasks in series, invisible before/after parallel tasks, AND-split tasks connecting the source place, and AND-join tasks connecting the sink place, etc.

2. The number of invisible tasks in the process model mined by the two algorithms. When the values of the three metrics are the same, the smaller the number of invisible tasks is, the better the result is. From Figure 14(b), we can see that for most of the testing example logs, the number of invisible tasks mined by the $\alpha^{\#}$ algorithm is smaller than that mined by the GA algorithm. Only for a few logs, the GA algorithm can mine a model with less invisible tasks. However, the mined models are not sound or they are not as good as the ones mined by the $\alpha^{\#}$ algorithm.
3. The time spent by the two mining algorithms. If the previous two perspectives cannot distinguish which results are better, the time spent by the corresponding algorithm can be considered. The less the time is spent, the better the result is. Figure 14(c) shows that for all the chosen logs, the mining time spent on each log by the GA algorithm is almost 100 to 10000 times as that spent by the $\alpha^{\#}$ algorithm.

The evaluation results show that so long as the event logs are complete, the $\alpha^{\#}$ algorithm can mine all useful invisible tasks in DIWF-nets successfully. Compared to the GA algorithm that can handle invisible tasks, the $\alpha^{\#}$ algorithm shows good performance (e.g., mining capacity, quality and efficiency).

6.4 Limitations of the $\alpha^{\#}$ algorithm

From Definition 13 and Theorem 1, it is obvious to see that the mining capacity of the $\alpha^{\#}$ algorithm is limited to DIWF-nets and the given event log is assumed to be complete. Although DIWF-nets is a large subclass of WF-nets, there are still some sound non-DIWF-nets that cannot be mined by the $\alpha^{\#}$ algorithm (e.g., the two WF-nets N_{10} and N_{11} shown in Figure 7(a) and Figure 7(b) respectively). For N_{10}, one of its complete event log is $W_{10} = \{ac, bcde, bdce\}$. Taking W_{10} as input, the $\alpha^{\#}$ algorithm constructs the WF-net named N'_{10} as shown in Figure 15(a). After one token is put in the source place of N'_{10}, there will be a free-choice between a and b. If a is chosen, the net will terminate normally. Otherwise, after b executes, c and d can execute concurrently. The executions of c and d will not affect the soundness of N'_{10}. However, if the only invisible task executes after c finishes execution, there will be a deadlock at e. Thus the mined WF-net N'_{10} is not structural sound though it is a DIWF-net. The reason for such a mining error is that although there is a place connecting a and the only invisible task t' in N_{10}, t' has no chance to execute immediately after a. But the essential reason is the non-free-choice construct between e and t'. For N_{11}, one of its complete event log is $W_{11} = \{acd, abcd, acbd\}$. The corresponding mined WF-net by the $\alpha^{\#}$ algorithm is N'_{11} as shown in Figure 15(b). After analyzing the structure of N'_{11} , we can draw the conclusion that it is structural sound. However, N_{11} and N'_{11} are not behavioral equivalent. N'_{11} cannot generate the trace acd and hence it cannot cover the given log W_{11}. The essential reason is that although a and d are connected by an invisible elementary path, d cannot execute immediately after a because there is another parallel path without invisible tasks between them.

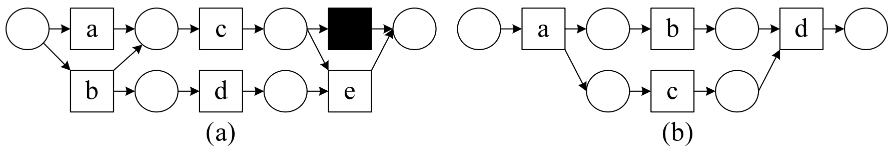

Fig. 15. Two mined models from the complete logs generated by two non-DIWF-nets

The correctness of the $\alpha^{\#}$ algorithm depends on the assumption that the potential WF-nets are DIWF-nets and the given log is complete. These two assumptions should be relaxed further in future works.

7 Conclusion

Based on the analysis of mining problems encountered using the classical α algorithm, a new mining algorithm based on Petri net named $\alpha^{\#}$ algorithm is proposed. Invisible tasks are classified into four types according to their functionalities for the first time, i.e., SIDE, SKIP, REDO and SWITCH. The universal detection method for invisible tasks of SKIP/REDO/SWITCH types is illustrated in detail and the correctness of the method can be proved theoretically. The construction algorithms for all types of invisible tasks and the process models in WF-nets are proposed and explained too. The $\alpha^{\#}$ algorithm has been implemented as a plug-in of ProM and evaluated using a lot of artificial logs and a few real-life logs. The evaluation results show that the algorithm can mine appropriate DIWF-nets with invisible tasks successfully so long as the corresponding event logs are complete.

Our future work will mainly focus on the following two aspects. Firstly, more real-life logs will be gathered for further evaluating the $\alpha^{\#}$ algorithm and the implemented plug-in. Secondly, theoretical analysis will be done to explore the exact mining capacity of the $\alpha^{\#}$ algorithm.

Acknowledgements

The work is supported by the National Basic Research Program of China (No. 2002CB312006 and No. 2007CB310802), the National Natural Science Foundation of China (No. 60373011 and No. 60473077) and the Program for New Century Excellent Talents in University.

The authors would like to thank Ton Weijters, Ana Karla Alves de Medeiros, Boudewijn van Dongen, Minseok Song, Laura Maruster, Eric Verbeek, Monique Jansen-Vullers, Hajo Reijers, Michael Rosemann, and Peter van den Brand for their on-going work on process mining techniques and tools at Eindhoven University of Technology.

References

1. W.M.P. van der Aalst. The application of petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

2. W.M.P. van der Aalst and B.F. van Dongen. Discovering Workflow Performance Models from Timed Logs. In Y. Han, S. Tai, and D. Wikarski, editors, *International Conference on Engineering and Deployment of Cooperative Information Systems*, volume 2480 of *Lecture Notes in Computer Science*, pages 45–63. Springer-Verlag, Berlin, 2002.
3. W.M.P. van der Aalst, B.F. van Dongen, C.W. Günther, R.S. Mans, A.K. Alves de Medeiros, A. Rozinat, V. Rubin, M. Song, H.M.W. Verbeek, and A.J.M.M. Weijters. Prom 4.0: Comprehensive support for real process analysis. In J. Kleijn and A. Yakovlev, editors, *The 28th International Conference on Applications and Theory of Petri Nets (ICATPN 2007)*, volume 4546 of *Lecture Notes in Computer Science*, pages 484–494. Springer-Verlag, Berlin, 2007.
4. W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A.J.M.M. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003.
5. W.M.P. van der Aalst and A.J.M.M. Weijters, editors. *Process Mining*, Special Issue of Computers in Industry, Volume 53, Number 3. Elsevier Science Publishers, Amsterdam, 2004.
6. W.M.P. van der Aalst and A.J.M.M. Weijters. Process Mining: A Research Agenda. *Computers in Industry*, 53(3):231–244, 2004.
7. W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.
8. W.M.P. van der Aalst, M. Weske, and D. Grünbauer. Case Handling: A New Paradigm for Business Process Support. *Data and Knowledge Engineering*, 53(2):129–162, 2005.
9. R. Agrawal, D. Gunopulos, and F. Leymann. Mining process models from workflow logs. In I. Ramos, G. Alonso, and H.J. Schek, editors, *the Sixth International Conference on Extending Database Technology*, pages 469–483, 1998.
10. J.E. Cook and A.L. Wolf. Automating process discovery through event-data analysis. In *Proceedings of the 17th international conference on Software engineering*, pages 73–82. ACM, New York, NY, USA, 1995.
11. J.E. Cook and A.L. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.
12. J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1995.
13. B.F. van Dongen, A.K. Alves de Medeiros, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. van der Aalst. The prom framework: A new era in process mining tool support. In G. Ciardo and P. Darondeau, editors, *International Conference on Application and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 444–454. Springer-Verlag, Berlin, 2005.
14. G. Greco, A. Guzzo, G. Manco, and D. Saccá. Mining and reasoning on workflows. *IEEE Transactions on Knowledge and Data Engineering*, 17(4):519–534, 2005.
15. G. Greco, A. Guzzo, L. Pontieri, and D. Saccá. Discovering expressive process models by clustering log traces. *IEEE Transactions on Knowledge and Data Engineering*, 18(8):1010–1027, 2006.
16. M. Hammori, J. Herbst, and N. Kleiner. Interactive workflow miningrequirements, concepts and implementations. *Data and Knowledge Engineering*, 56:41–63, 2006.
17. J. Herbst and D. Karagiannis. Workflow Mining with InWoLvE. *Computers in Industry*, 53(3):245–264, 2004.

18. X.Q. Huang, L.F. Wang, W. Zhao, S.K. Zhang, and C.Y. Yuan. A workflow process mining algorithm based on synchro-net. *Journal of Computer Science and Technology*, 21(1):66–71, 2006.
19. L. Maruster, A.J.M.M. Weijters, W.M.P. van der Aalst, and A. van der Bosch. A Rule-Based Approach for Process Discovery: Dealing with Noise and Imbalance in Process Logs. *Data Mining and Knowledge Discovery*, 13(1):67–87, 2006.
20. A.K.A. de Medeiros, W.M.P. van der Aalst, and A.J.M.M. Weijters. Workflow Mining: Current Status and Future Directions. In R. Meersman, Z. Tari, and D.C. Schmidt, editors, *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 389–406. Springer-Verlag, Berlin, 2003.
21. A.K.A. de Medeiros, B.F. van Dongen, W.M.P. van der Aalst, and A.J.M.M. Weijters. Process Mining for Ubiquitous Mobile Systems: An Overview and a Concrete Algorithm. In L. Baresi, S. Dustdar, H. Gall, and M. Matera, editors, *Ubiquitous Mobile Information and Collaboration Systems*, pages 154–168, 2004.
22. A.K.A. de Medeiros, A.J.M.M. Weijters, and W.M.P. van der Aalst. Genetic process mining: an experimental evaluation. *Data Mining and Knowledge Discovery*, 14(2):245–304, 2007.
23. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.
24. A. Rozinat and W.M.P. van der Aalst. Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models. In C. Bussler et al., editor, *BPM 2005 Workshops*, volume 3812 of *Lecture Notes in Computer Science*, pages 163–176. Springer-Verlag, Berlin, 2006.
25. A. Rozinat and W.M.P. van der Aalst. Decision Mining in ProM. In S. Dustdar, J.L. Faideiro, and A. Sheth, editors, *the Fourth International Conference on Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pages 420–425. Springer-Verlag, Berlin, 2006.
26. G. Schimm. Mining exact models of concurrent workflows. *Computers in Industry*, 53(3):265–281, 2004.
27. L.J. Wen, W.M.P. van der Aalst, J.M. Wang, and J.G. Sun. Mining process models with non-free-choice constructs. *Data Mining and Knowledge Discovery*, 15(2):145–180, 2007.
28. L.J. Wen, J.M. Wang, and J.G. Sun. Detecting implicit dependencies between tasks from event logs. In X. Zhou, X. Lin, and H. Lu et al., editors, *The 8th Asia-Pacific Web Conference (APWeb 2006)*, volume 3841 of *Lecture Notes in Computer Science*, pages 591–603. Springer-Verlag, Berlin, 2006.
29. L.J. Wen, J.M. Wang, and J.G. Sun. Mining invisible tasks from event logs. In G.Z. Dong, X.M. Lin, W. Wang, and Y. Yang, editors, *the Joint Conference of the 9th Asia-Pacific Web Conference and the 8th International Conference on Web-Age Information Management*, volume 4505 of *Lecture Notes in Computer Science*, pages 358–365. Springer-Verlag, Berlin, 2007.