Workflow Mining: Current Status and Future Directions

A.K.A. de Medeiros, W.M.P. van der Aalst, and A.J.M.M. Weijters

Department of Technology Management, Eindhoven University of Technology
P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands.
{a.k.medeiros, w.m.p.v.d.aalst, a.j.m.m.weijters}@tm.tue.nl

Abstract. Current workflow management systems require the explicit design of the workflows that express the business process of an organization. This process design is very time consuming and error prone. Considerable work has been done to develop heuristics to mine event-data logs to produce a process model that can support the workflow design process. However, all the existing heuristic-based mining algorithms have their limitations. To achieve more insight into these limitations the starting point in this paper is the α-algorithm [3] for which it is *proved* under which conditions and process constructs the algorithm works. After presentation of the α-algorithm, a classification is given of the process constructs that are difficult to handle for this type of algorithms. Then, for some constructs (i.e. short loops) it is illustrated in which way the α-algorithm can be extended so that it can correctly discover these constructs.

Keywords: Process mining, workflow mining, Petri nets, workflow Petri nets.

1 Introduction

Every company wants to produce more in less time. One way to accomplish this is having a well-defined business process model that reflects the dependencies among tasks and also tasks that can be processed in parallel. *Workflow management*(WFM) systems offer the functionality to design and enact operational processes.

In an ideal situation, well-defined business processes should be designed before enactment is possible and, redesigned whenever changes happen. However, in practice a lot of time is spent on modelling business process while the resulting workflow models are typically still error prone, because knowledge about the whole process is scattered among employees and paper procedures.

To avoid the above mentioned difficulties, instead of starting with a process design, our process mining starts by gathering information about the processes as they take place. We assume that it is possible to record events such that (i) each event refers to a task (i.e., a well-defined step in the process), (ii) each event refers to a case (i.e., a process instance), and (iii) events are totally ordered. Any information system using transactional systems such as ERP (Enterprise

Resource Planning), CRM (Customer Relationship Management), B2B (Business to Business), SCM (Supply Chain Management) and WFM systems will offer this information in some form. Note that we do not assume the presence of a WFM system. The only assumption we make, is that it is possible to collect a process log that records the order in which the events take place.

Table 1. A process log.

case identifier	task identifier
case 1	task A
case 2	task A
case 3	task A
case 3	task B
case 1	task B
case 1	task C
case 2	task C
case 4	task A
case 2	task B
case 2	task D
case 5	task E
case 4	task C
case 1	task D
case 3	task C
case 3	task D
case 4	task B
case 5	task F
case 4	task D

To illustrate the principle of process mining, we consider the process log shown in Table 1. This log contains information about five cases (i.e., process instances) and six tasks (A..F). Based on the information shown in Table 1 and by making some assumptions about the completeness of the log (i.e., assuming that the cases are representative and a sufficient large subset of possible behaviors is observed) we can deduce for example the process model shown in Figure 1. The model is represented in terms of a Petri net [17]. After executing A, tasks B and C are in parallel. Note that for this example we assume that two tasks are in parallel if they appear in any other. By distinguishing between start events and end events for tasks it is possible to explicitly detect parallelism. Instead of starting with A the process can also start with E. Task E is always followed by task F. Table 1 contains the minimal information we assume to be present.

For this simple example, it is quite easy to construct a process model that is able to regenerate the process log. For larger process models this is much more difficult. For example, if the model exhibits alternative and parallel routing, then the process log will typically not contain all possible combinations. Moreover, certain paths through the process model may have a low probability and therefore remain undetected. Noisy data (i.e., logs containing exceptions) can further

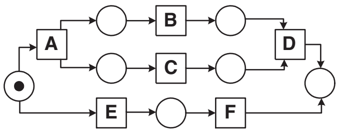

Fig. 1. A process model corresponding to the process log.

complicate matters. These are just some of the problems that we need to face in process mining research.

The focus of most research in the domain of process mining is on mining heuristics based on ordering relations of the events in the process log (cf. Section 5). Considerable work has been done on heuristics to mine event-data logs to produce a process model that can support the workflow design process. However, all the existing heuristic-based mining algorithms have their limitations. Typically, more advanced process constructs are difficult to handle for existing mining algorithms. Some of these problematic constructs are common in workflows and, therefore, need to be addressed to enable practical application. To achieve more insight into these limitations, the focus of this paper is a more analytical approach. The starting point of this paper is the α-algorithm [3]. Also the α-algorithm is primarily based on the ordening relations between events. However, the mining algorithm is not based on a heuristic, but on a formal algorithm for which it is *proved* under which conditions and process constructs the algorithm works. By discussing the weaknesses and strengths of the α-algorithm, we show how concepts in workflow mining could be improved in order to allow the correct mining of common constructs that appear in workflow system (loops, duplicate tasks, implicit places, non-free-choice constructs, etc.). Our final goal is to extend the α-algorithm so that the class of constructs for which we can *prove* that we can mine them correctly becomes larger. For some constructs (i.e. short loops) it is illustrated how the α-algorithm can be extended so that it can correctly handle these constructs.

The rest of the paper is organized as follows. In Section 2, the α-algorithm is explained. Problematic constructs that are not adequately tackled by α-algorithm are explained in Section 3. Possible ways to tackle these constructs are discussed in Section 4. Section 5 discusses related work on process mining. The final observations and comments are given in Section 6.

2 Workflow Mining: The α-Algorithm

The α-algorithm receives as input an event log and returns as output a Place/-Transition net (P/T-net) [17]. This section shows the main concepts required to understand the α-algorithm. A complete description and its properties is given in [3].

In the more theoretical approach, we do not focus on issues such as noise. We assume that there is no noise and that the workflow log contains "sufficient" information. Under these ideal circumstances we investigate whether the α algorithm is possible to *rediscover* the workflow process, i.e., for which class of workflow models is it possible to accurately construct the model by merely looking at their logs. The α algorithm is based on four ordering relations which can be derived from the log: $>_W$, $\rightarrow_W$, $\#_W$, and $\|_W$.

Definition 2.1. (Log-based ordering relations) Let W be a workflow log over T, i.e., $W \in \mathcal{P}(T^*)$. Let $a, b \in T$:

- $a >_W b$ if and only if there is a trace $\sigma = t_1 t_2 t_3 \ldots t_{n-1}$ and $i \in \{1, \ldots, n-2\}$ such that $\sigma \in W$ and $t_i = a$ and $t_{i+1} = b$,
- $a \rightarrow_W b$ if and only if $a >_W b$ and $b \not>_W a$,
- $a \#_W b$ if and only if $a \not>_W b$ and $b \not>_W a$, and
- $a \|_W b$ if and only if $a >_W b$ and $b >_W a$.

Relation $\rightarrow_W$ suggests causality and relations $\|_W$ and $\#_W$ are used to differentiate between parallelism and choice. Since all relations can be derived from $>_W$, we assume the log to be complete with respect to $>_W$ (i.e., if one task can follow another task directly, then the log should have registered this potential behavior). Structured Workflow Petri nets (SWF-nets) are a subclass of workflow nets (WF-nets) in which the net structure explicitly shows its behavior. Consequently, in SWF-nets (i) choice and synchronization are not mixed, and (ii) if there is a synchronization, all of its preceding transitions will have fired. These constraints are illustrated in Figure 2. Additionally, SWF-nets do not allow for implicit places in the net structure [3].

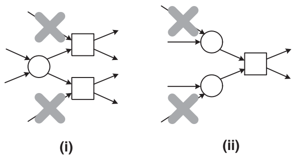

Fig. 2. Constructs not allowed in SWF-nets.

To formally define the α algorithm we introduce some basic terminology.

Definition 2.2. ($\in$, *first*, *last*) Let T be a set of tasks. Let $\sigma = a_1 a_2 \ldots a_n \in T^*$ a sequence over T of length n. $\in$, *first*, and *last* are defined as follows:

1. $a \in \sigma$ if and only if $a \in \{a_1, a_2, \ldots a_n\}$,
2. if $n \geq 1$, then $\mathit{first}(\sigma) = a_1$ and $\mathit{last}(\sigma) = a_n$.

Now we can give the formal definition of the α algorithm followed by a more intuitive explanation.

Definition 2.3. (Mining algorithm α) Let W be a workflow log over T. $\alpha(W)$ is defined as follows.

1. $T_W = \{t \in T \mid \exists_{\sigma \in W} t \in \sigma\}$,
2. $T_I = \{t \in T \mid \exists_{\sigma \in W} t = \mathit{first}(\sigma)\}$,

3. $T_O = \{t \in T \mid \exists_{\sigma \in W} t = last(\sigma)\}$,
4. $X_W = \{(A,B) \mid A \subseteq T_W \ \wedge \ B \subseteq T_W \ \wedge \ \forall_{a \in A} \forall_{b \in B} a \rightarrow_W b \ \wedge \ \forall_{a_1,a_2 \in A} a_1 \#_W a_2 \ \wedge \ \forall_{b_1,b_2 \in B} b_1 \#_W b_2\}$,
5. $Y_W = \{(A,B) \in X_W \mid \forall_{(A',B') \in X_W} A \subseteq A' \wedge B \subseteq B' \Longrightarrow (A,B) = (A',B')\}$,
6. $P_W = \{p_{(A,B)} \mid (A,B) \in Y_W\} \cup \{i_W, o_W\}$,
7. $F_W = \{(a, p_{(A,B)}) \mid (A,B) \in Y_W \ \wedge \ a \in A\} \ \cup \ \{(p_{(A,B)}, b) \mid (A,B) \in Y_W \ \wedge \ b \in B\} \ \cup \{(i_W, t) \mid t \in T_I\} \ \cup \{(t, o_W) \mid t \in T_O\}$, and
8. $\alpha(W) = (P_W, T_W, F_W)$.

The α-algorithm works as follows. First, it examines the log traces and (Step 1) creates the set of transitions (T_W) in the workflow, (Step 2) the set of output transitions (T_I) of the source place , and (Step 3) the set of the input transitions (T_O) of the sink place[1]. In steps 4 and 5, the α-algorithm creates sets (X_W and Y_W, respectively) used to define the places of the mined workflow net. In Step 4, it discovers which transitions are causally related. Thus, for each tuple (A,B) in X_W, each transition in set A causally relates to *all* transitions in set B, and no transitions within A (or B) follow each other in some firing sequence. These constraints to the elements in sets A and B allow the correct mining of AND-split/join and OR-split/join constructs. Note that the OR-split/join requires the fusion of places. In Step 5, the α-algorithm refines set X_W by taking only the largest elements with respect to set inclusion. In fact, Step 5 establishes the exact amount of places the mined net has (excluding the source place i_W and the sink place o_W. The places are created in Step 6 and connected to their respective input/output transitions in Step 7. The mined workflow net is returned in Step 8.

Definition 2.4. (Ability to rediscover) Let $N = (P,T,F)$ be a sound WF-net and let the α be a mining algorithm which maps workflow logs of N onto sound WF-nets. If for any complete workflow log W of N the mining algorithm returns N (modulo renaming of places), then the α is able to *rediscover* N.

An algorithm/heuristic is said to rediscover a workflow net if this algorithm is able to regenerate the *exact net structure* of the original net, abstracting from the place labels (see Definition 2.4). The α-algorithm is proved to (re)discover all SWF-nets if the SWF-net does not contain short-loops. That means that short loops are a first limitation of the α-algorithm. However, if the notion of *ability to rediscover* is relaxed to *behaviorally equivalent* (i.e. both generate the same log traces), then the α-algorithm is able to mine other sound WF-nets, like the one in Figure 3. This net is not an SWF-net because transition G can be enabled without the firing of transitions E and F. However, even with this relaxed notion of *ability to rediscover* and the restriction on short-loops, the α-algorithm cannot be proved to correctly mine all sound WF-nets.
The next section classifies the situations to which the α-algorithm fails to mine sound WF-nets. Understanding the limitations of α-algorithm helps in developing new algorithms/heuristics to tackle these limitations.

[1] In a workflow net, the source place i has no input transitions and the sink place o has no output transitions.

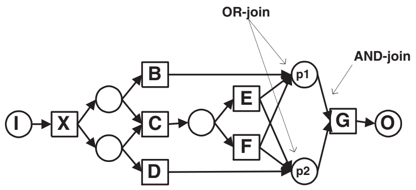

Fig. 3. A WF-net that can be rediscovered by the α-algorithm, although it is not an SWF-net.

3 Limitations of the α-Algorithm: Loops, Invisible Tasks, and Duplicate Tasks

As motivated in the previous section the α-algorithm can successfully mine SWF-nets that do not contain short-length loops. But, the α-algorithm has also serious limitations. Although it is possible to represent many real workflows using SWF-nets, these nets do not support other common constructs like *invisible tasks* and *duplicate tasks*. In this section we present a classification of possible common constructs the α-algorithm cannot mine correctly, and relations between these constructs. Some of the constructs are within the scope of SWF-nets (like *short loops*), but others are beyond the scope of SWF-nets (like *duplicate tasks*).

To find out the constructs the α-algorithm cannot mine correctly, it is necessary to understand how it works. Basically, the α-algorithm has the following behavior:

– A task *exists* in the resulting net if it *is in any* log trace;
– A task has *ingoing* arc(s) in the resulting net if (i) this task is the *first* task in a log trace, or (ii) this task *causally follows* another task.
– A task has *outgoing* arc(s) in the resulting net if (i) this task is the *last* task in a log trace, or (ii) this task *is causally followed* by another task.

If a task is not the first or last task in any trace log, *and* is not involved in any causal relation, the α-algorithm does not generate ingoing and outgoing arcs for this task. For instance, see net N_3 in Figure 4. Note that task B is not connected to any place in the resulting net. However, even if all the transitions are connected in the resulting net, this does not guarantee that the α-algorithm correctly mined the net. For instance, see the original and resulting nets in figures 5, 6, 7 and 8.

Places are created based on the *causal* ($\rightarrow_W$) and *exclusive* ($\#_W$) relations. However, in some situations the resulting net does not have the same number of places the original net has. For instance, consider the net in Figure 3 and net N_1 in Figure 5. Both nets are non-SWF-net and have similar net structures. In fact, these nets are not the same because each task E and F has only *one* outgoing arc in net N_1. This slight difference in the net structure leads to the inferring of different causal and exclusive relations to these two nets. Consequently, the α-algorithm cannot correctly mine N_1, but it can mine the net in Figure 3.

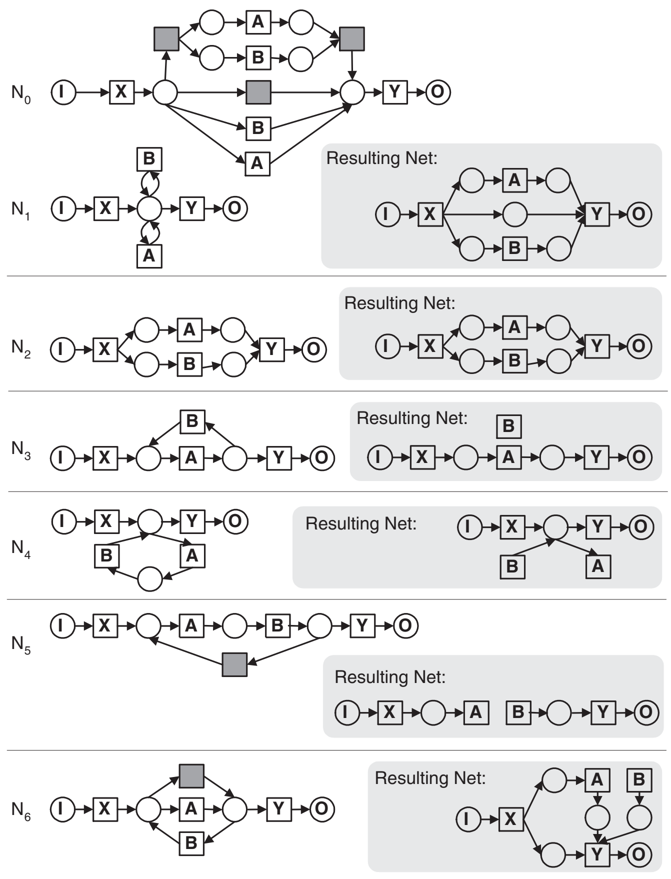

Fig. 4. Example of the existing relations between duplicate tasks, invisible tasks and one/two-length loops.

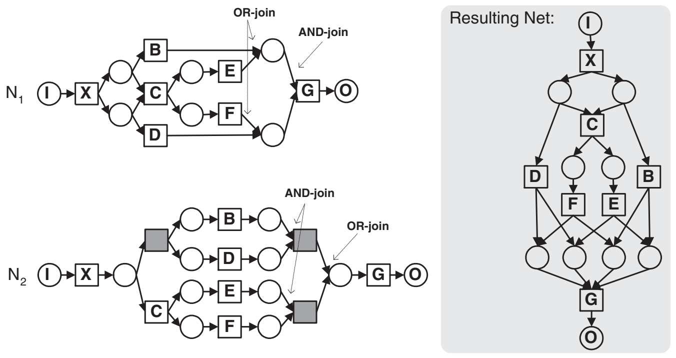

Fig. 5. Mined and original nets have different number of places.

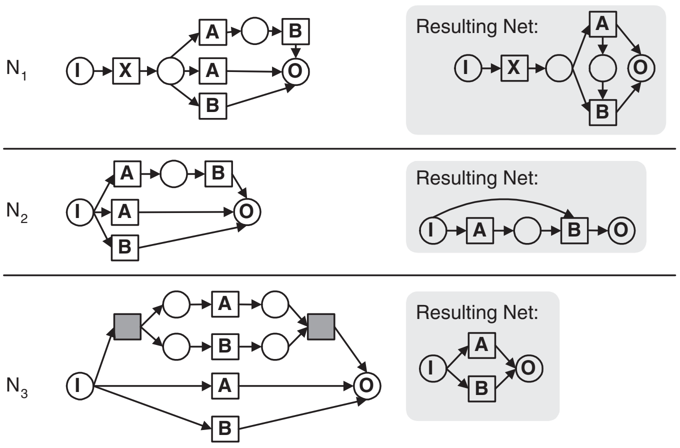

Fig. 6. Nets with duplicate tasks.

There are problems in the resulting net the α-algorithm produces when its *input* is incomplete and/or has noise (because different relations may be inferred). But even if the log is noise free and complete, there are a number of workflow constructs that causes problems for the α-algorithm. Below we will discuss them.
One-length loop. In a one-length loop, the same task can be executed multiple times in sequence. Thus, all ingoing places of this task are also its outgoing places in the WF-net. In fact, for SWF-nets, a one-length-loop task can only have one single place connected to it. As an example, see net N_5 in Figure 7, and also net N_1 in Figure 4. Note that in the resulting nets, the one-length-loop transitions do not have the same place as its ingoing and outgoing place. This happens because, to generate a place with a common ingoing and outgoing task, the α-algorithm requires the causal relation $task \rightarrow_W task$. But it is impossible to have $task >_W task$ and $task \not>_W task$ at the same time.
Two-length loop. In this case, the α-algorithm infers the two involved tasks are in parallel and, therefore, no place is created between them. For instance, see nets N_3 and N_4 in Figure 4. Note that there are no arcs between tasks A and B in the resulting net. However, the α-algorithm would correctly mine both N_3 and N_4 if the relations $A \rightarrow_W B$ and $B \rightarrow_W A$ were inferred, instead of the relation $A||_W B$.
Invisible Tasks. Invisible tasks do not appear in any log trace. Consequently, they do not belong to T_W (set of transitions in the mined net) and cannot be present in the net the α-algorithm generates. Two situations lead to invisible tasks: (i) a task is not registered in the log, for instance, because it may have only a routing purpose (e.g., see tasks without label in net N_2, Figure 5), or (ii) there is noise in the log generation and real tasks are missing in the log traces.
Duplicate Tasks. Sometimes a task appear more than once in the same workflow. In this case, the same label is given (and thus registered in the log) to more than one task. This can happen, for instance, when modelling the booking

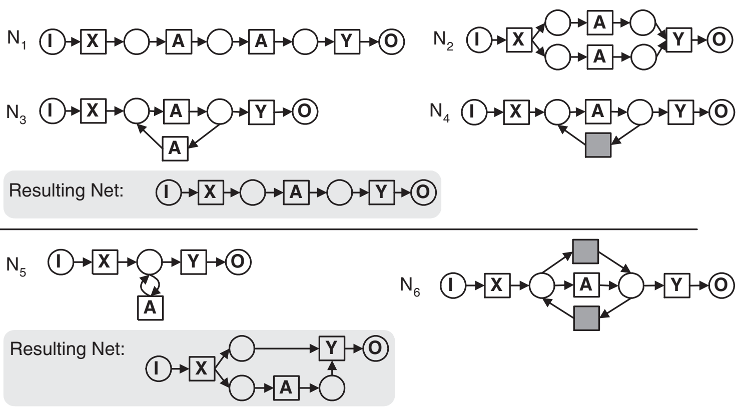

Fig. 7. Example of the existing relations between duplicate tasks, invisible tasks and one-length loops.

process in a travel agency. Clients can go there to *book a flight* only, *book a hotel* only, or *both*. Thus, a workflow model describing this booking situation could be like net N_3, in Figure 6 (assume $A =$"book flight" and $B =$"book hotel"). Note that the resulting net for net N_3 contains only one task with label A and one with B. The α-algorithm will never capture task duplication because it cannot distinguish different task with the same label (see also the other nets in Figure 6). In fact, in an SWF-net it is assumed that tasks are uniquely identifiable. Thus, a heuristic to capture duplicate tasks will have to generate WF-nets in which tasks can have identical labels.

Implicit Places. SWF-nets do not have implicit places. Places are implicit if their presence or absence does not affect the possible log traces of a workflow. For example, places $p3$ and $p4$ are implicit in net N_3 (see Figure 8). Note that the same causal relations are inferred when these implicit places are present or absent. However, the α-algorithm creates places according to the existing causal relations. Thus, implicit places cannot be captured because they do not influence causal relations between tasks. Note also that this same reason prevents the α-algorithm of generating *explicit* places between tasks that do not have a causal relation. As an example, see places $p1$ and $p2$ in net N_2 (also in Figure 8). Both places constrain the execution of tasks D and E because the choice between the execution of these tasks is made after the execution of A or B, respectively, and not after the execution of C. In fact, if the places $p1$ and $p2$ are removed from N_2, net N_4 is obtained (see Figure 8). However, in N_4, the choice between the execution of tasks D and E is made after the execution of task C. Consequently, a log trace like $XACEY$ can be generated by N_4, but cannot by N_2.

Non-free choice. The non-free choice construct combines synchronization and choice. Thus, it is not allowed in SWF-nets because it corresponds to construct (i) in Figure 2. Nets containing non-free choice constructs are not always mined correctly by the α-algorithm. For instance, consider the non-free-choice net N_2, Figure 8. The α-algorithm does not mine correctly N_2 because this net cannot

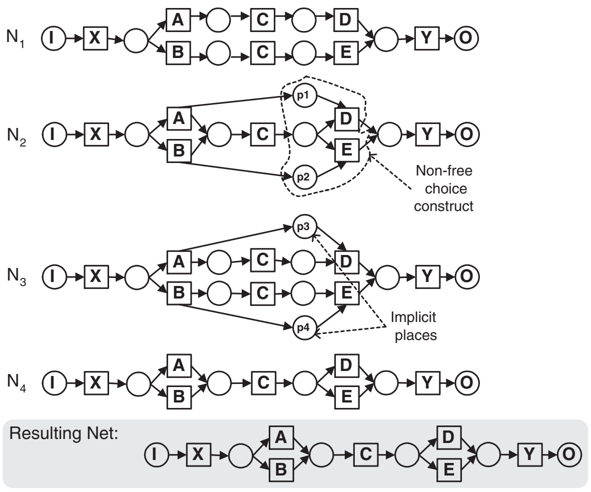

Fig. 8. Example of existing relations between duplicate tasks, non-free choice nets, implicit places and SWF-nets.

generate any log trace with the substring AD and/or BE. Consequently, there is no causal relation $A \rightarrow_W D$ and $B \rightarrow_W E$, and no creation of the respective places $p1$ and $p2$ in the resulting net. However, there are non-free choice constructs which the α-algorithm can mine correctly. As an example, consider net N_1 in Figure 9. This net is similar to net N_2 in Figure 8, but N_1 has two additional tasks F and G. The α-algorithm can correctly mine N_1 because there is a causal relation $F \rightarrow_W D$ (enabling the creation of place $p1$) and $G \rightarrow_W E$ (enabling the creation of $p2$). Thus, the α-algorithm can correctly mine non-free-choice constructs as far as the causal relations can be inferred.

Synchronization of OR-join places. The *synchronization of OR-join places* is a non-SWF-net construct because it correspond to construct (ii) in Figure 2. However, although this is a non-SWF-net construct, sometimes the α-algorithm can correctly mine it. For instance, see the WF-net in Figure 3. Places $p1$ and $p2$ are OR-join places. $p1$ is an OR-join place because it contains a token if task B *or* E *or* F is executed. Similarly, $p2$ if task D *or* E *or* F is executed. Besides, both $p1$ and $p2$ are synchronized at task G, since this task can happen only when there is a token in both $p1$ *and* $p2$. Note that this construct corresponds to a non-SWF-net because task G can be executed whenever *some* of the tasks that precede it have been executed. If the net in Figure 3 were an SWF-net, task G could be executed only after the execution of tasks B, D, E and F. However, although the net in Figure 3 is a non-SWF-net, the α-algorithm can correctly mine it because the necessary and sufficient *causal* ($\rightarrow_W$) and *exclusive*($\#_W$) relations are inferred. However, for some *synchronization of OR-join places* constructs, the inferred causal and exclusive relations are not enough to correctly mine the net. For instance, consider net N_1 in Figure 5. The resulting net the α-algorithm mines is not equal to N_1 because it contains two additional places among tasks B, D, E, F and task G. This net structure with extra places derives from the inferred relations. Note that because $B \parallel_W D$ and $E \parallel_W F$ in net N_1, but $B \#_W E$, $B \#_W F$, $D \#_W E$ and $D \#_W F$, the places $p_{(\{B,E\},\{G\})}$, $p_{(\{B,F\},\{G\})}$, $p_{(\{D,E\},\{G\})}$ and $p_{(\{D,F\},\{G\})}$ are created by the α-algorithm, when only places $p_{(\{B,E\},\{G\})}$ and $p_{(\{D,F\},\{G\})}$ would do. Thus, in this case, the inferred relations do not allow the α-algorithm to correctly mine the net. However, the resulting net is *behaviorally* equivalent to the original net, even if their structures are different because both nets generate exactly the same set of traces.

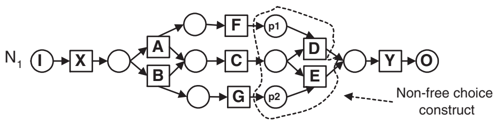

Fig. 9. Example of a non-free choice net which the α-algorithm can mine correctly.

There are relations among the problematic constructs that imply in trade-offs. The problematic constructs are related because (i) the same set of log traces can satisfy the current notion of log completeness, and/or (ii) the same set of ordering relations can be inferred when the original net contains one of the constructs. Therefore, no mining algorithm can detect which of the constructs are in the original net. In fact, any mining algorithm must choose which one of the related constructs is going to be used in the resulting net. Some examples demonstrating that the problematic constructs are related:

Duplicate Tasks (Sequence vs Parallel vs Choice). Duplicate tasks can be in *sequential*, *parallel*, or *choice* structures in the WF-net. These duplicate task structures are related because the same complete log can satisfy different WF-nets containing them. As an example, see the respective nets N_1, N_2 and N_3 in Figure 7. Note that a log containing *only* the trace $XAAY$ would be complete for the three nets N_1, N_2, and N_3. Thus, given this input trace, it is impossible for a mining algorithm to determine which duplicate task structure really exists in the original net.

Invisible Tasks vs Duplicate Tasks. WF-nets with the same ordering relations can be created either using invisible tasks or using duplicate tasks. For instance, consider nets N_3 and N_4 in Figure 7. Their ordering relations are the same whatever the workflow log. Additionally, note that a log containing *only* the trace $XAAY$ would be complete also for nets N_{1-3} and N_4.

Invisible Tasks vs Loops. Behaviorally equivalent WF-nets can be created either using invisible tasks or using loops. For instance, consider nets N_5 and N_6 in Figure 7. These nets generate exactly the same set of log traces.

Invisible Tasks vs Synchronization of OR-join places. See nets N_1 and N_2 in Figure 5. The α-algorithm generates the same resulting net for both N_1 and N_2 because these nets are behaviorally equivalent.

Non-Free Choice vs Duplicate Tasks. Nets N_1 and N_2 in Figure 8 are behaviorally equivalent.

Loops vs Invisible Tasks together with Duplicate Tasks. Nets with equal sets of ordering relations can be created if loops or invisible tasks in combination with duplicate tasks are used. For instance, see nets N_0 and N_1 in Figure 4. Net N_0 has duplicate tasks and invisible tasks in its structure. Net N_1 has two one-length loops, involving tasks A and B. These two nets lead to the same set of ordering relations because, whatever the complete log, the inferred causal and parallel ordering relations will always be $X \rightarrow_W A$, $X \rightarrow_W B$, $X \rightarrow_W Y$, $B \rightarrow_W Y$, $A \rightarrow_W Y$, and $A \|_W B$.
In fact, these relations raise questions like: Is it possible to develop heuristics that detect both loops and invisible tasks? Duplicate tasks and invisible tasks? If it is not, what problematic constructs should have priority in the mining? In what situations? These are the kind of questions our current research is trying to answer. In the following section we explain possible approaches to tackle the classes of structural constructs the α-algorithm cannot mine correctly. Additionally, we give examples on how to apply these approaches.

4 Approaches to Tackle Structural Problematic Constructs

Process mining can be viewed as a three-phase process: *pre-processing*, *processing* and *post-processing*. In the pre-processing phase, based on the assumption that the input log satisfies the required notion of log completeness, the ordering relations are inferred. The processing phase corresponds to the execution of the mining algorithm, given the log and the ordering relations as input. In our case, the mining algorithm is the α-algorithm. During post-processing, the mined Petri-net can be fine-tuned and a graphical representation can be build. Possible approaches to tackle structural problematic constructs focus on one or more of these phases.

In this section, we use the problematic constructs *one- and two-length loops* in SWF-nets to exemplify how approaches can be developed to tackle problematic constructs. We chose to tackle them first because in this way we can extend the α-algorithm to mine all SWF-nets (including short loops). Subsection 4.1 contains an approach to tackle one-length loops. This is a mixed approach that focusses both on the pre- and post-processing phases. Subsection 4.2 presents an approach to tackle two-length loops. This approach focusses on the pre-processing phase.

4.1 Example of a Mixed Approach Focusing on the Pre- and Post-processing Phases

To develop an approach to tackle one-length loops in SWF-nets, we first determine (i) how one-length loops can be identified in the input log and (ii) what kind of patterns can be used to build them in SWF-nets.
Identification. One-length loops can be identified by checking if there are log traces containing the substring $t_1 t_1$. For instance, any complete log for N_5 in Figure 7 contains the trace $XAAY$.

WF-structure. For SWF-nets, it can be proven that one-length-loop tasks are connected to a single place. The WF-structure is illustrated in Figure 10.

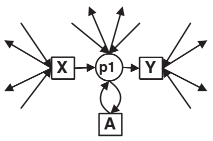

Fig. 10. Structure of one-length loops in SWF-nets.

The reasoning used to identify this single structure is as follows. Let task A be in a one-length loop. First, A can never be connected to source/sink places in an SWF-net because the source place i has no ingoing task and the sink place o has no outgoing task. Second, task A cannot have more than one ingoing place (see N_1 in Figure 11) because SWF-nets do not allow for synchronization and choice to mixed (recall construct (i) in Figure 2). Third, task A cannot be connected to places that are only its outgoing places (see N_2 in Figure 11) because these places can contain more than one token. All places in SWF-nets contain *at most* one token. Finally, at least two other tasks (X and Y) are necessary. The X task puts a token in the place connected to A (all tasks are live in SWF-nets). The Y removes a token from this place (in SWF-nets, no tasks can execute after the process termination).

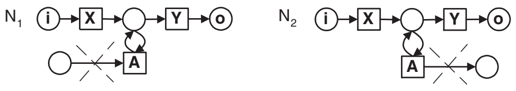

Fig. 11. Illustration of the reasoning used to determine the single structure of one-length loops in SWF-nets.

The unique structure in which one-length loops appears in SWF-nets is represented in Figure 10. Three *distinct* tasks can be distinguished: the one-length-loop task (A), one *ingoing* task (X) and one *outgoing* task (Y). Consequently, for every one-length-loop pattern, there are *at least* the causal relations: $X \rightarrow_W Y$, $X \rightarrow_W A$ and $A \rightarrow_W Y$. Besides, every one-length-loop task A is connected to a single place ($p1$ in Figure 10) because we are working with SWF-nets. Thus, if we remove A from this pattern, it is still possible to mine $p1$ in this pattern because the causal relation $X \rightarrow_W Y$ still exists. In order words, it is possible to mine the basic SWF-net structure of the workflow process without considering the one-length-loop tasks when inferring the ordering relations. This reasoning is the base for the following mixed approach.

First, in a *pre-processing* phase the one-length-loop tasks and their respective neighbors are identified and recorded. Then, the one-length-loop tasks are eliminated from the log. Secondly, the α-algorithm is applied to the pre-processed log.

The result is a WF-net with the $X \rightarrow_W Y$ causal relation and the p_1 place. In the *post-processing* phase, based on the recorded data, the one-length loop tasks are connected to the right places in the WF-net generated by α-algorithm. Note that this approach does not modify the processing phase (i.e. the α algorithm) itself.

4.2 Example of an Approach Focusing on the Pre-processing Phase

To build an approach to tackle two-length loops in SWF-nets, we first need to set (i) how two-length loops can be identified and (ii) what kind of patterns can be used to build them in SWF-nets.
Identification. The current notion of log completeness does not allow the differentiation between tasks in parallel and tasks in a two-length loop. This happens because a log can be complete without having one trace in which the two-length-loop tasks follow one another in a row. In other words, if t_1 and t_2 belong to a two-length loop in an SWF-net, a log can be complete without having the pattern $t_1t_2t_1$. For instance, see net N_1 in Figure 12. The log containing the traces XAY, $XABW$, XW, ZBW, $ZBAY$ and ZY is complete. However, this log does not contain the pattern ABA. Thus, any approach to tackle two-length loops in SWF-nets requires a new notion of log completeness.

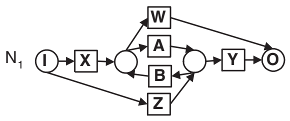

Fig. 12. Example of an SWF-net for which the new notion of log completeness is required to correctly capture 2-length loops.

WF-structure. Recall that the current definition of ordering relations infers that tasks in two-length loops are in parallel. In contrast to one-length-loops the possible structure for two-length loops is not completely clear. Our discoveries so far shows that two-length loops can be mined correctly if the causal relations of tasks involved in the two-length loop are correctly mined.

Our proposed solution for the two-length-loop problem is an adaptation of the original definition of log completeness and an adaptation of the definition of some the basic relations.

In the original definition of log completeness (Section 2), we assume the log to be complete with respect to $>_W$ (i.e., if one task can follow another task directly, then the log should have registered this potential behavior). In the adapted version not only the binary $>_W$ relation, but also triples are involved. If the pattern $t_1t_2t_1$ is possible, a complete log must contain this triple.

Using this insight, we redefine Definition 2.1, i.e., we provide new definitions for the four basic ordering relations $>_W$, $\rightarrow_W$, $\#_W$, and $\|_W$.

Definition 4.1. (Ordering relations capturing two-length loops) Let W be a loop-complete workflow log over T, i.e., $W \in \mathcal{P}(T^*)$. Let $a, b \in T$:

- $a >''_W b$ if and only if there is a trace $\sigma = t_1 t_2 t_3 \dots t_{n-1}$ and $i \in \{1, \dots, n-2\}$ such that $\sigma \in W$ and $t_i = a$ and $t_{i+1} = b$,
- $a \rightarrow''_W b$ if and only if $(a >_W b$ and $(b \not>_W a$ or $\exists_{\sigma \in W}[\sigma = t_1 t_2 t_3 \dots t_n$ and $i \in \{1, \dots, n-2\}$ and $t_i = t_{i+2} = a$ and $t_{i+1} = b]))$,
- $a \#''_W b$ if and only if $a \not>_W b$ and $b \not>_W a$, and
- $a \|''_W b$ if and only if $a >_W b$ and $b >_W a$ and $\neg\exists_{\sigma \in W}[\sigma = t_1 t_2 t_3 \dots t_n$ and $i \in \{1, \dots, n-2\}$ and $t_i = t_{i+2} = a$ and $t_{i+1} = b])$.

Note that Definition 4.1 considers the new notion of log completeness. The main idea is that two tasks t_1 and t_2 (with $t_1 \neq t_2$), will be in parallel if, and only if, there is no log trace containing the substring $t_1 t_2 t_1$. If the α-algorithm is applied using the new Definition 4.1, an SWF-net containing two-length loops can be mined. Examples are the nets N_3 and N_4 in Figure 4, and net N_1 in Figure 12. Note that this approach enables the mining of SWF-nets with two-length-loops by modifying only the pre-processing phase (establishing the basic relations $\rightarrow_W$, $\#_W$, and $\|_W$).

5 Literature on Process Mining

The idea of process mining is not new [4,6,7,8,10,11,12,14,15,18,19,2,20,3]. Cook and Wolf have investigated similar issues in the context of software engineering processes. In [6] they describe three methods for process discovery: one using neural networks, one using a purely algorithmic approach, and one Markovian approach. The authors consider the latter two the most promising approaches. The purely algorithmic approach builds a finite state machine where states are fused if their futures (in terms of possible behavior in the next k steps) are identical. The Markovian approach uses a mixture of algorithmic and statistical methods and is able to deal with noise. Note that the results presented in [6] are limited to sequential behavior. Cook and Wolf extend their work to concurrent processes in [7]. They propose specific metrics (entropy, event type counts, periodicity, and causality) and use these metrics to discover models out of event streams. However, they do not provide an approach to generate explicit process models. Recall that the final goal of the approach presented in this paper is to find explicit representations for a broad range of process models, i.e., we want to be able to generate a concrete Petri net rather than a set of dependency relations between events. In [8] Cook and Wolf provide a measure to quantify discrepancies between a process model and the actual behavior as registered using event-based data. The idea of applying process mining in the context of workflow management was first introduced in [4]. This work is based on workflow graphs, which are inspired by workflow products such as IBM MQSeries workflow (formerly known as Flowmark) and InConcert. In this paper, two problems are defined. The first problem is to find a workflow graph generating events appearing in a given workflow log. The second problem is to find the definitions of edge conditions. A concrete algorithm is given for tackling the first problem. The approach is quite different from other approaches: Because the nature of workflow graphs there is no need to identify the nature (AND or OR) of joins and splits. As shown in [13], workflow graphs use true and false tokens which do not allow for cyclic graphs. Nevertheless, [4] partially deals with iteration by enumerating all occurrences of a given task and then folding the graph. However, the resulting conformal graph is not a complete model. In [15], a tool based on these algorithms is presented. Schimm [18,19] has developed a mining tool suitable for discovering hierarchically structured workflow processes. This requires all splits and joins to be balanced. Herbst and Karagiannis also address the issue of process mining in the context of workflow management [11,10,12] using an inductive approach. The work presented in [12] is limited to sequential models. The approach described in [11,10] also allows for concurrency. It uses stochastic task graphs as an intermediate representation and it generates a workflow model described in the ADONIS modeling language. In the induction step task nodes are merged and split in order to discover the underlying process. A notable difference with other approaches is that the same task can appear multiple times in the workflow model, i.e., the approach allows for duplicate tasks. The graph generation technique is similar to the approach of [4,15]. The nature of splits and joins (i.e., AND or OR) is discovered in the transformation step, where the stochastic task graph is transformed into an ADONIS workflow model with block-structured splits and joins. In contrast to the previous papers, our work [14,20] is characterized by the focus on workflow processes with concurrent behavior (rather than adding ad-hoc mechanisms to capture parallelism). In [20] a heuristic approach using rather simple metrics is used to construct so-called "dependency/frequency tables" and "dependency/frequency graphs". The preliminary results presented in [20] only provide heuristics and focus on issues such as noise. In [1] the EMiT tool is presented which uses an extended version of α-algorithm to incorporate timing information. For a detailed description of the α-algorithm and a proof of its correctness we refer to [3].

More from a theoretical point of view, the rediscovery problem discussed in this paper is related to the work discussed in [5,9,16]. In these papers the limits of inductive inference are explored. For example, in [9] it is shown that the computational problem of finding a minimum finite-state acceptor compatible with given data is NP-hard. Several of the more generic concepts discussed in these papers could be translated to the domain of process mining. It is possible to interpret the problem described in this paper as an inductive inference problem specified in terms of rules, a hypothesis space, examples, and criteria for successful inference. The comparison with literature in this domain raises interesting questions for process mining, e.g., how to deal with negative examples (i.e., suppose that besides log W there is a log V of traces that are not possible, e.g., added by a domain expert). However, despite the many relations with the work described in [5,9,16] there are also many differences, e.g., we are mining at the net level rather than sequential or lower level representations (e.g., Markov chains, finite state machines, or regular expressions). For a survey of existing research, we also refer to [2].

6 Discussion and Future Work

The focus of this paper has been on process mining algorithms and heuristics primarily based on binary ordering relations of the events in a process log. As an representative example of this type of algorithms we introduced the α-algorithm and we explained why it cannot correctly mine *short loops, invisible tasks, duplicate tasks, implicit places, non-free choice* and *synchronization of OR-join places*, which are all common constructs in workflows. It is important to note that these limitations are not specific for the α-algorithm but apply to most of the approaches described in literature.

Additionally, we have showed how two problematic constructs (i.e., loops of length one and length two) can be handled by adapting one or more process mining phases: *pre-processing*, *processing* or *post-processing*.

Our future research will be driven by the problems identified in this paper. First, we want to extend the class of WF-nets the α-algorithm can correctly mine. Secondly, we want to extend our mining algorithm in such a way that it can handle workflows beyond the scope of WF-nets (for instance workflows with duplicate or invisible tasks). Finally, we try to combine formal results with more practical approaches in which we try to develop mining heuristics so that we can handle more workflow logs (i.e., logs with noise and logs that are incomplete).

References

1. 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 (EDCIS 2002)*, volume 2480 of *Lecture Notes in Computer Science*, pages 45–63. Springer-Verlag, Berlin, 2002.
2. 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, Accepted for publication, 2003.
3. 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 (TKDE), Accepted for publication, 2003.
4. R. Agrawal, D. Gunopulos, and F. Leymann. Mining Process Models from Workflow Logs. In *Sixth International Conference on Extending Database Technology*, pages 469–483, 1998.
5. D. Angluin and C.H. Smith. Inductive Inference: Theory and Methods. *Computing Surveys*, 15(3):237–269, 1983.
6. 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.
7. J.E. Cook and A.L. Wolf. Event-Based Detection of Concurrency. In *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, pages 35–45, 1998.
8. J.E. Cook and A.L. Wolf. Software Process Validation: Quantitatively Measuring the Correspondence of a Process to a Model. *ACM Transactions on Software Engineering and Methodology*, 8(2):147–176, 1999.

9. E.M. Gold. Complexity of Automaton Identification from Given Data. *Information and Control*, 37(3):302–320, 1978.
10. J. Herbst. Dealing with Concurrency in Workflow Induction. In U. Baake, R. Zobel, and M. Al-Akaidi, editors, *European Concurrent Engineering Conference*. SCS Europe, 2000.
11. J. Herbst. *Ein induktiver Ansatz zur Akquisition und Adaption von Workflow-Modellen*. PhD thesis, Universität Ulm, November 2001.
12. J. Herbst and D. Karagiannis. Integrating Machine Learning and Workflow Management to Support Acquisition and Adaptation of Workflow Models. *International Journal of Intelligent Systems in Accounting, Finance and Management*, 9:67–92, 2000.
13. B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows (submitted)*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2002. Available via http://www.tm.tue.nl/it/research/patterns.
14. L. Maruster, A.J.M.M. Weijters, W.M.P. van der Aalst, and A. van den Bosch. Process Mining: Discovering Direct Successors in Process Logs. In *Proceedings of the 5th International Conference on Discovery Science (Discovery Science 2002)*, volume 2534 of *Lecture Notes in Artificial Intelligence*, pages 364–373. Springer-Verlag, Berlin, 2002.
15. M.K. Maxeiner, K. Küspert, and F. Leymann. Data Mining von Workflow-Protokollen zur teilautomatisierten Konstruktion von Prozeßmodellen. In *Proceedings of Datenbanksysteme in Büro, Technik und Wissenschaft*, pages 75–84. Informatik Aktuell Springer, Berlin, Germany, 2001.
16. L. Pitt. Inductive Inference, DFAs, and Computational Complexity. In K.P. Jantke, editor, *Proceedings of International Workshop on Analogical and Inductive Inference (AII)*, volume 397 of *Lecture Notes in Computer Science*, pages 18–44. Springer-Verlag, Berlin, 1889.
17. 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.
18. G. Schimm. Process Mining. http://www.processmining.de/.
19. G. Schimm. Process Miner – A Tool for Mining Process Schemes from Event-based Data. In S. Flesca and G. Ianni, editors, *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA)*, volume 2424 of *Lecture Notes in Computer Science*, pages 525–528. Springer-Verlag, Berlin, 2002.
20. A.J.M.M. Weijters and W.M.P. van der Aalst. Rediscovering Workflow Models from Event-Based Data using Little Thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, 2003.