Mining Process Models with Non-Free-Choice Constructs

Lijie Wen[1], Wil M.P. van der Aalst[2], Jianmin Wang[1], and Jiaguang Sun[1]

[1] School of Software, Tsinghua University, 100084, Beijing, China
wenlj00@mails.tsinghua.edu.cn,{jimwang,sunjg}@tsinghua.edu.cn
[2] Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
w.m.p.v.d.aalst@tm.tue.nl

Abstract. Process mining aims at extracting information from event logs to capture the business process as it is being executed. Process mining is particularly useful in situations where events are recorded but there is no system enforcing people to work in a particular way. Consider for example a hospital where the diagnosis and treatment activities are recorded in the hospital information system, but where healthcare professionals determine the “careflow”. Many process mining approaches have been proposed in recent years. However, in spite of many researchers’ persistent efforts, there are still several challenging problems to be solved. In this paper, we focus on mining non-free-choice constructs, i.e., situations where there is a mixture of choice and synchronization. Although most real-life processes exhibit non-free-choice behavior, existing algorithms are unable to adequately deal with such constructs. Using a Petri-net-based representation, we will show that there are two kinds of causal dependencies between tasks, i.e., explicit and implicit ones. We propose an algorithm that is able to deal with both kinds of dependencies. The algorithm has been implemented in the ProM framework and experimental results shows that the algorithm indeed significantly improves existing process mining techniques.

1 Introduction

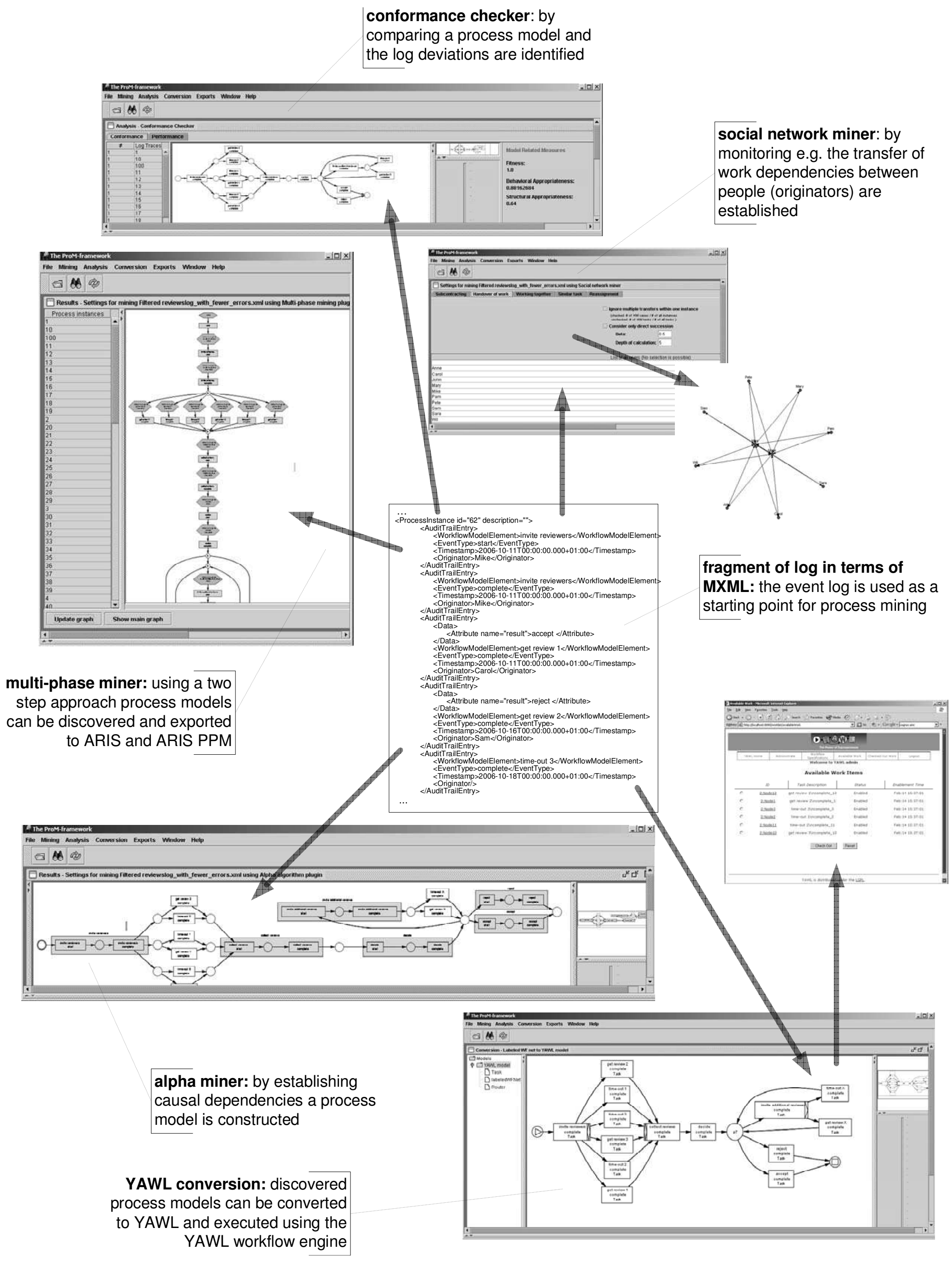
Today’s information systems are logging events that are stored in so-called “event logs”. For example, any user action is logged in ERP systems like SAP R/3, workflow management systems like Staffware, and case handling systems like FLOWer. Classical information systems have some centralized database for logging such events (called transaction log or audit trail). Modern service-oriented architectures record the interactions between web services (e.g., in the form of SOAP messages). Moreover, today’s organizations are forced to log events by national or international regulations (cf. the Sarbanes-Oxley (SOX) Act [47] that is forcing organizations to audit their processes). As a result of these developments, there is an abundance of process-related data available. Unfortunately, today’s organizations make little use of all of the information recorded. Buzzwords such as BAM (Business Activity Monitoring), BOM (Business Operations Management), BPI (Business Process Intelligence) illustrate the interest in techniques that extract knowledge from event logs. However, most organizations are still unaware of the possibilities that exist and most of software solutions only offer basic tools to measure some key performance indicators.

Process mining aims at a more fine grained analysis of processes based on event logs [10, 12, 25, 54]. The goal of process mining is to extract information about processes from these logs [10]. We typically 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 also referred to as *activity*), (ii) each event refers to a *case* (i.e., a process instance), (iii) each event can have a *performer* also referred to as *originator* (the person executing or initiating the task), (iv) events have a *timestamp*, (v) events can have associated *data*, and (vi) events are totally ordered. Moreover, logs may contain transactional information (e.g., events refereing to the start, completion, or cancellation of tasks).

In process mining, we distinguish three different perspectives: (1) the process perspective, (2) the organizational perspective and (3) the case perspective. The *process perspective* focuses on the control-flow, i.e., the ordering of tasks. The goal of mining this perspective is to find a good characterization of all possible paths, e.g., expressed in terms of a Petri net, an Event-driven Process Chain (EPC, [36]), or a UML activity diagram. The *organizational perspective* focuses on the originator field, i.e., which performers are involved and how are they related. The goal is to either structure the organization by classifying people in terms of roles and organizational units or to show relations between individual performers (i.e., build a social network). The *case perspective* focuses on properties of cases. Cases can be characterized by their path in the process or by the originators working on a case. However, cases can also be characterized by the values of the corresponding data elements. For example, if a case represents a replenishment order, it is interesting to know the supplier or the number of products ordered.

For each of the three perspectives there are both *discovery approaches* and *conformance checking approaches*. Discovery aims at deriving a model without a-priori knowledge, e.g., the construction of a Petri net, social network, or decision tree based on some event log. Conformance checking assumes some a-priori model and compares this model with the event log. Conformance checking is not used to discover a model but aims at discovering discrepancies between the model and a log.

Figure 1 shows a small part of a log in the MXML format (center of figure) [26]. This is the format used by the ProM (Process Mining) framework [25]. Using the ProMimport tool one can convert event logs from the following systems: Eastman Workflow, FLOWer, PeopleSoft, Staffware, Websphere, Apache HTTP Server, CPN Tools, CVS, and Subversion to MXML [26]. The ProM framework is a plugable environment where it is easy to add new process mining approaches or other types of analysis. Figure 1 shows only 5 of the more than 70 plugins available in ProM 3.0. As shown, the plug-ins focus on all different perspectives and on both discovery and conformance. Each of the screenshots shows a plug-in


conformance checker: by comparing a process model and the log deviations are identified
social network miner: by monitoring e.g. the transfer of work dependencies between people (originators) are established
fragment of log in terms of MXML: the event log is used as a starting point for process mining
multi-phase miner: using a two step approach process models can be discovered and exported to ARIS and ARIS PPM
alpha miner: by establishing causal dependencies a process model is constructed
YAWL conversion: discovered process models can be converted to YAWL and executed using the YAWL workflow engine

```
...
<ProcessInstance id="62" description="">
        <AuditTrailEntry>
            <WorkflowModelElement>invite reviewers</WorkflowModelElement>
            <EventType>start</EventType>
            <Timestamp>2006-10-11T00:00:00.000+01:00</Timestamp>
            <Originator>Mike</Originator>
        </AuditTrailEntry>
        <AuditTrailEntry>
            <WorkflowModelElement>invite reviewers</WorkflowModelElement>
            <EventType>complete</EventType>
            <Timestamp>2006-10-11T00:00:00.000+01:00</Timestamp>
            <Originator>Mike</Originator>
        </AuditTrailEntry>
        <AuditTrailEntry>
            <Data>
                <Attribute name="result">accept </Attribute>
            </Data>
            <WorkflowModelElement>get review 1</WorkflowModelElement>
            <EventType>complete</EventType>
            <Timestamp>2006-10-11T00:00:00.000+01:00</Timestamp>
            <Originator>Carol</Originator>
        </AuditTrailEntry>
        <AuditTrailEntry>
            <Data>
                <Attribute name="result">reject </Attribute>
            </Data>
            <WorkflowModelElement>get review 2</WorkflowModelElement>
            <EventType>complete</EventType>
            <Timestamp>2006-10-16T00:00:00.000+01:00</Timestamp>
            <Originator>Sam</Originator>
        </AuditTrailEntry>
        <AuditTrailEntry>
            <WorkflowModelElement>time-out 3</WorkflowModelElement>
            <EventType>complete</EventType>
            <Timestamp>2006-10-18T00:00:00.000+01:00</Timestamp>
            <Originator/>
        </AuditTrailEntry>
 ...
```

Fig. 1. Overview of process mining illustrated by the different plug-ins in ProM working on the MXML log shown in the center

working on the log in the center of the diagram. Only a fragment of the log is shown (the whole log contains information from 100 cases each having dozens of events). The *alpha miner* was one of the first process mining algorithms able to deal with concurrency, it focuses on discovery in the process perspective. The *multi-phase miner* uses a two-step approach to discover process models (in EPC or Petri net format) that exploits the so-called "OR connector" construct to be more robust than the alpha miner. The *social network miner* focuses on discovery in the organizational perspective. The *conformance checker* in ProM detects discrepancies between some model and a log. Using ProM it is possible to import from and export to a variety of tools. For example, it is possible to import/export models from/to CPN Tools, ARIS, and YAWL, i.e., it is possible to use CPN Tools to do simulations [39] and the workflow management system YAWL [6] is able to enact any process model discovered with one of the mining algorithms in ProM (including the algorithm presented in this paper).

ProM has been used in various domains, e.g., governmental agencies, municipalities, hospitals, ERP systems, etc. In this paper, we will not discuss concrete applications. Nevertheless, it is important to see the wide application possibilities of process mining. Consider for example the *diagnosis and treatment processes in a hospital.* Today's hospital information systems record a variety of events (e.g., blood tests, operations, appointments, X-rays, etc.). However, hospital managers are only provided with data at the level of an individual patient or at an aggregated level (e.g., the number of knee operations). Somehow, current hospital information systems completely miss the notion of a *process*, i.e., each patient is seen as a unique case. To improve service and to use the hospital resources more efficiently, it is important to analyze the *careflows*, i.e., typical diagnosis and treatment trajectories, in more detail. Process mining can be used for this purpose. Another example, is the *monitoring of webservices in an E-business setting.* Languages such as BPEL and CDL have been proposed to describe the way services can interact from a behavioral perspective, e.g., abstract BPEL [15] can be used to describe business protocols. As shown in [4] tools such as ProM can be used to do conformance testing in this setting, i.e., verifying whether one or more parties stick to the agreed-upon behavior by observing the actual behavior, e.g., the exchange of messages between all parties. Note that it is possible to translate BPEL business protocols to Petri nets and to relate SOAP messages to transitions in the Petri net. As a result, the ProM conformance checker can be used to quantify fitness (whether the observed behavior is possible in the business protocol). Of course the SOAP messages exchanged between the webservices can also be used to directly discover the actual behavior of each party. These two examples illustrate the wide applicability of process mining.

This paper focuses on process discovery and is related to plugins such as the alpha miner and the multi-phase miner in Figure 1 [10, 12, 25, 54]. We will abstract from the organizational perspective and the case perspective and focus on the discovery of process models. As a representation we will use Petri nets [23, 24].[3] As indicated in [40], one of the *main problems of existing process mining approaches is their inability of discovering non-free-choice processes.* Non-free-choice processes contain a mixture of synchronization and choice, i.e., synchronization and choice are not separated which may create *implicit dependencies.* Existing algorithms have no problems discovering explicit dependencies but typically fail to discover implicit dependencies. Note that the term "free-choice" originates from the Petri net domain, i.e., free-choice Petri nets are a subclass of Petri nets where transitions consuming tokens from the same place should have identical input sets [23]. Many real-life processes do not have this property. Therefore, it is important to provide techniques able to successfully discover non-free-choice processes. This paper proposes a new algorithm (named α^{++}) to discover non-free choice Petri nets.

The remainder of this paper is organized as follows. Section 2 reviews related work and argues that this work extends existing approaches in a significant way. Section 3 gives some preliminaries about process mining. Section 4 lists the sample process models that current mining algorithms can not handle. Section 5 defines explicit and implicit dependencies between tasks and gives all cases in which implicit dependencies must be detected correctly. Section 6 gives three methods for detecting implicit dependencies. In Section 7, we propose the algorithm α^{++} for constructing process models. Experimental results are given in Section 8. Section 9 concludes the paper.

2 Related work

The work proposed in this project proposal is related to existing work on process-aware information systems [27], e.g., WFM systems [5, 35, 37] but also ERP, CRM, PDM, and case handling systems [13].

Clearly, this paper builds on earlier work on process mining. The idea of applying process mining in the context of workflow management was first introduced in [14]. A similar idea was used in [22]. Cook and Wolf have investigated similar issues in the context of software engineering processes using different approaches [20]. In [18, 19], they extend the previous work. A technique to find the points in the system that demonstrate mutually exclusive and synchronized behavior is presented in [18]. The emphasis is on how to discover thread interaction points in a concurrent system. In [19], the techniques based on a probabilistic analysis of the event traces are presented to discover patterns of concurrent behavior from these traces of workflow events. Besides immediate event-to-event dependencies, these techniques are also able to infer some high order dependencies as well as one-task and two-task loops. However, only direct dependencies between events are considered and indirect ones which we call implicit dependencies are not involved at all. Herbst and Karagiannis address the issue of process mining in the context of workflow management using an inductive approach [33].

[3] Note that we use this as an internal representation. In the context of ProM it is easy to convert this to other format such as EPCs [36] that can be loaded into the ARIS toolset [49] or YAWL models that can be enacted by the YAWL workflow engine [6].

They use stochastic task graphs as an intermediate representation and generate a workflow model described in the ADONIS modeling language. The mined workflow model allows tasks having duplicate names and captures concurrency. The α algorithm [12] theoretically constructs the final process model in WF-nets, which is a subset of Petri nets. This algorithm is proven to be correct for a large class of processes, but like most other techniques it has problems in dealing with noise and incompleteness. Therefore, more heuristic approaches have been developed [53, 54] and, recently, also genetic approaches have been explored and implemented [7]. The topic of process mining is also related to the synthesis of models and systems from sequence diagrams (e.g., UML sequence diagrams or classical Message Sequence Diagrams) [32, 38]. Note that the tool used in this paper (ProM) also allows for the synthesis of sequence diagrams. It is also interesting to note the relationship between process mining and classical approaches based on finite state automata [17, 16, 44, 45]. The main difference between these approaches and process mining such as it is considered in this paper is the notion of concurrency and explicit dependencies. Clearly it is possible to translate a finite state automata into a Petri net. However, either the Petri net is very large and has no concurrent transitions or the theory of regions [28] is needed to fold the Petri net. The latter approach is often not realistic because it requires the observation of all possible execution sequences.

Process mining is not limited to the control-flow perspective. For example, in [9] it is shown that event logs can be used to construct social networks [50, 52].

The notion of conformance has also been discussed in the context of security [8], business alignment [2], and genetic mining [7]. In [46] it is demonstrated how conformance can be defined and describes the corresponding ProM plugin. In [29] the process mining problem is faced with the aim of deriving a model which is as compliant as possible with the log data, accounting for fitness (called completeness) and also behavioral appropriateness (called soundness).

Process mining can be seen in the broader context of Business (Process) Intelligence (BPI) and Business Activity Monitoring (BAM). In [30, 31, 48] a BPI toolset on top of HP's Process Manager is described. The BPI toolset includes a so-called "BPI Process Mining Engine". In [42] Zur Muehlen describes the PISA tool which can be used to extract performance metrics from workflow logs. Similar diagnostics are provided by the ARIS Process Performance Manager (PPM) [34]. The latter tool is commercially available and a customized version of PPM is the Staffware Process Monitor (SPM) [51] which is tailored towards mining Staffware logs.

For more information on process mining we refer to a special issue of Computers in Industry on process mining [11] and a survey paper [10].

Starting point for the approach described in this paper is α algorithm [12]. Improvements of the basic α algorithm have been proposed in [40, 41] and [55, 56]. In [40], the limitations of the α algorithm are explored and an approach to deal with short-loops is proposed. The resulting algorithm, named α^+, is described in [41]. In this paper, we take the α^+ algorithm as a starting point and extend it to deal with non-free-choice constructs as well as detect implicit dependencies between tasks. In [55], an approach is proposed to explicitly exploit event types. But this requires a start and complete event for each activity. The work done in this paper is an extension of the work presented in [56]. In that paper, the authors only give theorems to detect implicit dependencies between tasks and do not involve eliminating redundant implicit dependencies as well as giving the algorithm for constructing the final process model.

3 Preliminaries

In this section, we give some definitions used throughout this paper. First, we introduce a process modeling language (WF-nets) and its relevant concepts. Then we discuss the notion of an event log in detail and give an example. Finally, we give a very brief introduction to the classical α algorithm [12].

3.1 WF-net

In this paper, WF-nets are used as the process modeling language [1]. WF-nets form a subset of Petri nets [23, 24]. Note that Petri net provides a graphical but formal language designed for modeling concurrency. Moreover, Petri nets provide all kinds of routings supported by a variety of process-aware information systems (e.g., WFM, BPM, ERP, PDM, and CRM systems) in a natural way. WF-nets are Petri nets with a single source place (start of process) and a single sink place (end of process) describing the life-cycle of a single case (process instance). In this paper, we will only consider *sound* WF-nets, i.e., WF-nets that once started for a case can always complete without leaving tokens behind. As shown in [1], soundness is closely related to well-known concepts such as liveness and boundedness [23, 24].

Figure 2 gives an example of a workflow process modeled in WF-net. This model has a non-free-choice construct. The transitions (drawn as rectangles) T_1, T_2, $\cdots$, T_5 represent tasks and the places (drawn as circles) P_1, P_2, $\cdots$, P_6 represent causal dependencies. A place can be used as pre-condition and/or post-condition for tasks. The arcs (drawn as directed edges) between transitions and places represent flow relations. In this process model, there is a non-free-choice construct, i.e., the sub-construct composed of P_3, P_4, P_5, T_4 and T_5. For T_4 and T_5 , their common input set is not empty but their input sets are not the same.

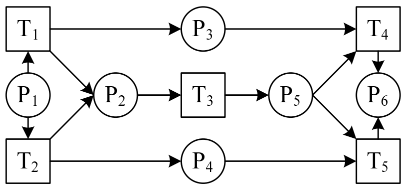


Fig. 2. An example of a workflow process in WF-net

We adopt the formal definitions and properties (such as soundness and safeness) of WF-net and SWF-net from [1, 12]. Some related definitions (such as implicit place), properties and firing rules about Petri nets are also described there.

In this paper, we demand that each task (i.e., transition) has an unique name in one process model. However, each task can appear multiple times in one process instance for the presence of iterative routings.

3.2 Event log

As described in Section 1 the goal of process mining is to extract information about processes from transactional event logs. In the remainder of this paper, 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. Note that the MXML format [26] mentioned in Section 1 and used by ProM [25] can store much more information (cf. timestamps, originators, transactional information, data, etc.). However, the algorithm presented in this paper does not need this additional information. Clearly, most information systems (e.g., WFM, ERP, CRM, PDM systems) will offer this minimal information in some form [12].

By sorting all the events in an event log by their process identifier and completion time, we can assume that an event has just two attributes, i.e., task name and case identifier. Table 1 gives an example of an event log.

Table 1. An event log for the process shown in Figure 2

Case id	Task name	Case id	Task name
1	T_1	2	T_2
1	T_3	2	T_3
1	T_4	2	T_5

This log contains information about two cases. The log shows that for case 1, T_1, T_3 and T_4 are executed. For case 2, T_2, T_3 and T_5 are executed. In fact, no matter how many cases there are in the event log, there are always only two distinct event traces, i.e., $T_1T_3T_4$ and $T_2T_3T_5$. Thus for the process model shown in Figure 2, this event log is a minimal and complete one. Here we adopt the definitions of (event) trace and event log from [12].

3.3 The classical α algorithm

As indicated in the introduction, many process mining approaches have been developed in recent years. Most of the classical approaches use simple process models such as finite state automata. The α algorithm was one of the first approaches to take concurrency into account (i.e., explicit causal dependencies and parallel tasks). Moreover, unlike many other theoretical approaches, a weaker form of completeness was assumed.

The α algorithm starts by analyzing the event log and then construct various dependency relations. To describe these relations we introduce the following notations. Let W be an event log over T, i.e., $W \subseteq T^*$. Let $a, b \in T$:

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

Consider some event log $W = \{ABCD, ACBD, AED\}$. Relation $>_W$ describes which tasks appeared in sequence (one directly following the other). Clearly, $A >_W B$, $A >_W C$, $A >_W E$, $B >_W C$, $B >_W D$, $C >_W B$, $C >_W D$, and $E >_W D$. Relation $\rightarrow_W$ can be computed from $>_W$ and is referred to as the *(direct) causal relation* derived from event log W. $A \rightarrow_W B$, $A \rightarrow_W C$, $A \rightarrow_W E$, $B \rightarrow_W D$, $C \rightarrow_W D$, and $E \rightarrow_W D$. Note that $B \not\rightarrow_W C$ because $C >_W B$. Relation $\|_W$ suggests concurrent behavior, i.e., potential parallelism. For log W tasks B and C seem to be in parallel, i.e., $B \|_W C$ and $C \|_W B$. If two tasks can follow each other directly in any order, then all possible interleavings are present and therefore they are likely to be in parallel. Relation $\#_W$ gives pairs of transitions that never follow each other directly. This means that there are no direct causal relations and parallelism is unlikely.

Based on these relations, the α algorithm starts constructing the corresponding Petri net. The algorithm assumes that two tasks a and b (i.e, transitions) are connected through some place if and only if $a \rightarrow_W b$. If tasks a and b are concurrent, then they can occur in any order, i.e., a may be directly followed by b or vice versa. Therefore, the α algorithm assumes that tasks a and b are concurrent if and only if $a \|_W b$. If $x \rightarrow_W a$ and $x \rightarrow_W b$, then there have to be places connecting x and a on the one hand and x and b on the other hand. This can be one place or multiple places. If $a \|_W b$, then there should be multiple places to enable concurrency, i.e., both a and b are triggered by x through separate places. If on the other hand $a \#_W b$, then there should be a single place to ensure that only one branch is chosen. This way it is possible to decide on the nature of a split. Similarly, one can decide on the nature of a join and this way construct the entire Petri net. It should be noted that the α algorithm can deal with much more complicated structures than a simple AND/XOR-split/join. For example, it is possible to start things in parallel and make a choice at the same time.

Although the α algorithm is able to deal with various forms of concurrency, it typically has problems correctly discovering implicit dependencies. These dependencies stem from a particular use of the non-free choice construct in Petri nets. For example, the α algorithm is unable to discover Figure 2 based on Table 1. The reason is that T_1 is never directly followed by T_4 and that T_2 is never directly followed by T_5. Hence, $T_1 \not\rightarrow_W T_4$ and $T_2 \not\rightarrow_W T_5$ (because $T_1 \not>_W T_4$ and $T_2 \not>_W T_5$) and the model discovered by the α algorithm is the WF-net without places P_3 and P_4. The resulting Petri net is sound but allows for too much behavior. This is only one example where the α algorithm fails to capture an implicit dependency. As we will show in the next section, there are many types of implicit dependencies that are even more difficult to handle. Yet this is crucial because these behaviors occur in many real-life processes.

4 Problems

To illustrate the difficulties of mining process models with non-free-choice constructs, we give some situations in which current process mining algorithms usually fail. See Figure 3 below. There are three WF-nets in the figure, i.e., N_1, N_2 and N_3. All nets in the left part are the original nets, while others in the right part are their corresponding mined nets using α-algorithm. Here the α-algorithm is chosen because it is the basis for the approaches described in this paper. The mining results of other process mining algorithms are structurally similar. For convenience, the mining results of N_1, N_2 and N_3 are called $N_1^{'}$, $N_2^{'}$ and $N_3^{'}$ respectively.

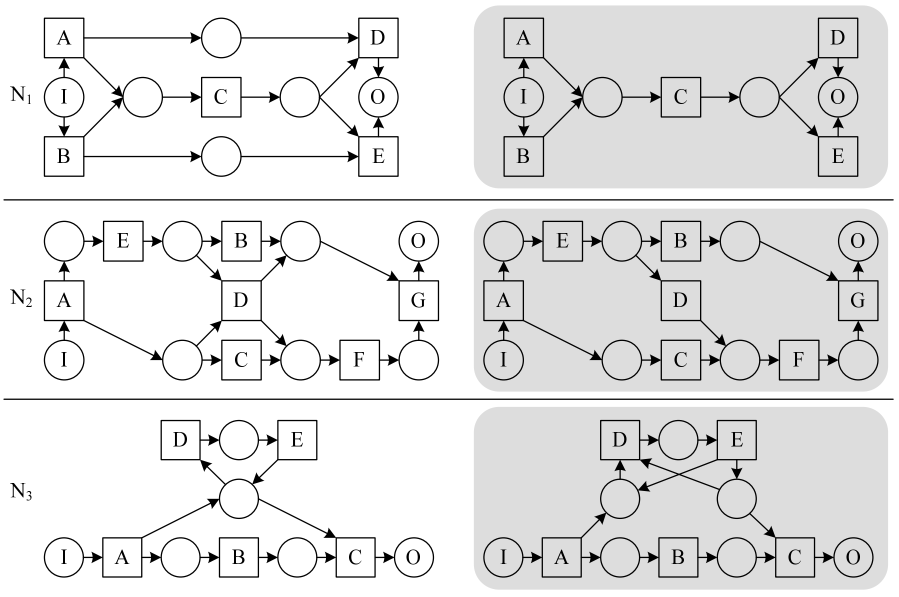


Fig. 3. Three pairs of process models: the models on the left contain non-free-choice constructs and cannot be discovered by traditional algorithms while the models on the right are the (incorrect) models generated by the α/α^+ algorithm

Before we discuss the WF-nets N_1, N_2, N_3, $N_1^{'}$, $N_2^{'}$ and $N_3^{'}$ shown in Figure 3, it is important to consider different notions of "correctness" in the context of process mining. In real-life situations, the real process is often not known a-priori (i.e., it exists but is not made explicit). Therefore, it is difficult to judge the correctness of the mining result. In [46] notions such as fitness and appropriateness are defined in the context of conformance checking. However, to determine the quality of a process mining technique in a more scientific setting, fitness and appropriateness are not very convenient because they compare an event log and the "discovered model" rather than the "real model" and the "discovered model". Moreover, given a log it is fairly easy to find over-generalized or over-specific models that can regenerate the observed behavior (see [46] for examples). Therefore, we want to compare some a-priori model with the a-posteriori (i.e., discovered) model. (Even though the a-priori model is not known in most real-life applications.) Moreover, given some a-priori model we will not assume that we are able to observe all possible execution sequences, instead we use *Occam's razor*, i.e., "one should not increase, beyond what is necessary, the number of entities required to explain anything" (William of Ockham, 14th century). Note that it would be unrealistic to assume that all possible firing sequences are present in the log. First of all, the number of possible sequences may be infinite (in case of loops). Second, parallel processes typically have an exponential number of states and, therefore, the number of possible firing sequences may be enormous. Finally, even if there is no parallelism and no loops but just N binary choices, the number of possible sequences may be 2^N. Therefore, we will use a weaker notion of completeness. We will define such a notion in Section 6 but in the meanwhile we assume some informal notion of correctness.

After discussing different notions of "correctness" in the context of process mining, we return to the WF-nets shown in Figure 3. N_1, N_2, N_3 are the WF-nets on the left-hand side, each representing some a-priori model. $N_1^{'}$, $N_2^{'}$ and $N_3^{'}$ are the corresponding models on the right-hand side, each representing the discovered model by applying classical algorithms such as the α or α^+algorithm

Let us consider the first WF-net shown in Figure 3. There is a non-free-choice between D and E in N_1, i.e., the choice is not made by D or E themselves, but is decided by the choice made between A and B. First, there is a free choice between A and B. After one of them is chosen to execute, C is executed. Finally, whether D or E is chosen to execute depends on which one of A and B has been executed. The minimal and complete event log of N_1 can be represented as $\{ACD,BCE\}$. Although the mining result $N_1^{'}$ is a sound WF-net, it is not behaviorally equivalent with N_1. The join places connecting A and D as well as B and E are missing in $N_1^{'}$. Thus the minimal and complete event log of $N_1^{'}$ is $\{ACD, ACE, BCD, BCE\}$. Compared to N_1, $N_1^{'}$ can generate two additional event traces, i.e., ACE and BCD. Obviously, this is a "mining failure" that should be avoided. In order to mine WF-nets with such feature, the mining algorithm must remember all the choices made in the net and investigate the relations between each pair of them later. The difficulty focuses on how to find all the choices between tasks from event log efficiently and how to use the relations between each pair of these choices correctly.

Let us now consider the second WF-net shown in Figure 3. There is a non-free-choice between B and D as well as C and D in N_2. After A is executed, there are two tasks (i.e., E and C) enabled concurrently. On the one hand, if E is chosen to execute first, B, C and D will be enabled concurrently. In the next step, either D will be executed or C and B will be executed concurrently. On the other hand, if C is chosen to execute first, there will be no choice to be made later. One of the minimal and complete event logs of N_2 can be represented as $\{AEDFG, AECBFG, ACEFBG, AEBCFG, ACFEBG\}$. Although N_2 is a sound WF-net, its mining result $N_2^{'}$ is not a sound one. Two join arcs (i.e., the arc from the place connecting A and C to D and the arc from D to the place connecting B and G) are missing in $N_2^{'}$. If D is executed at some time, there will be a deadlock at G finally. Here another "mining failure" occurs. The causal relation between A and D as well as D and G is not detected from the event log by the α-algorithm. The difficulty is how to detect similar causal relations between tasks from event logs to make the mining result sound and behaviorally equivalent with the original net.

Finally, we consider the third WF-net shown in Figure 3. There is a non-free-choice between C and D in N_3. After A is executed, B and D can be executed concurrently. Before C is executed, the sequence DE can be executed any number of times. C is only enabled when B has been executed and D is just enabled. One of the minimal and complete event logs of N_3 can be represented as $\{ABC, ABDEC, ADBEC, ADEDEBC\}$. Here N_3 is a sound WF-net, its mining result $N_3^{'}$ is not sound either. One causal relation (i.e., from A to C) is missing in $N_3^{'}$. After A and B are executed successively, the net blocks. Here again a "mining failure" surfaces. Such undetected causal relations should be mined correctly by the newcome mining algorithms.

In summary, the essence of such mining failure is that some causal relations between tasks are not detected from event logs by current process mining algorithms. Almost all the process mining algorithms today determine the possible causal relations between two tasks, e.g., a and b, if and only if the subsequence ab occurs in some event trace at least once. Causal relations detected from event logs using similar idea can be considered to have a *dependency distance* of one. In WF-nets, such as N_1, N_2 and N_3 shown in Figure 3, there are causal relations between tasks with longer (i.e., more than one) dependency distance. In this paper, we try to tackle such issues listed above in some extent.

5 Dependency classification

To distill a process model with non-free-choice constructs from event logs correctly, there must be a way to mine all the dependencies (i.e., causal relations) between tasks without mistakes. As research results show, not all dependencies can be mined from event logs directly by current process mining algorithms [40].

In fact, there are two kinds of dependencies between tasks in WF-nets, i.e., explicit and implicit ones. An *explicit dependency*, which is also called *direct dependency*, reflects direct causal relationships between tasks. An *implicit dependency*, which is also called *indirect dependency*, reflects indirect causal relationships between tasks. To clarify the differences between both classes of relationships, the corresponding formal definitions are given below.[4]

Definition 1 (Explicit Dependency). *Let $N = (P, T, F)$ be a sound WF-net with input place i and output place o. For any $a, b \in T$, there is an explicit dependency between a and b iff:*

1. *connective: $a \bullet \cap \bullet b \neq \emptyset$, and*

[4] We assume the reader to be familiar with the formal definition of Petri nets in terms of a three tuple (P, T, F) and related notations. For the reader not familiar with these notation we refer to [1, 3].

2. *successive: there is some reachable marking $s \in [N, [i]\rangle$ such that $(N, s)[a\rangle$ and $(N, s - \bullet a + a\bullet)[b\rangle$.*

Definition 2 (Implicit Dependency). *Let $N = (P, T, F)$ be a sound WF-net with input place i and output place o. For any $a, b \in T$, there is an implicit dependency between a and b iff:*

1. *connective: $a \bullet \cap \bullet b \neq \emptyset$,*
2. *disjunctive: there is no reachable marking $s \in [N, [i]\rangle$ such that $(N, s)[a\rangle$ and $(N, s - \bullet a + a\bullet)[b\rangle$, and*
3. *reachable: there is some reachable marking $s \in [N, [i]\rangle$ such that $(N, s)[a\rangle$ and there is some reachable marking $s' \in [N, s - \bullet a + a\bullet\rangle$ such that $(N, s')[b\rangle$.*

As Figure 2 shows, P_2 together with its surrounding arcs reflects explicit dependencies between T_1 and T_3 as well as T_2 and T_3. While P_3 together with its surrounding arcs reflects an implicit dependency between T_1 and T_4. If there are only explicit dependencies between tasks in a process model with non-free-choice constructs, most process mining algorithms, such as the α algorithm etc., can mine it correctly. Otherwise, existing process mining algorithms have problems "discovering" implicit dependencies.

Now we investigate what characteristics a process model with implicit dependencies may have. Assume that there is an implicit dependency between A and B. Once A is executed, there must be some other tasks before B to be executed. After that, B is to be executed. There is never any chance that B can directly follow A in some trace, because the "dependence distance" is at least two. So the implicit dependency between A and B has no chance to be detected directly, using classical approaches such as the $>$ relation in the α algorithm. A typical fragment of a process model with an implicit dependency is shown in Figure 4.

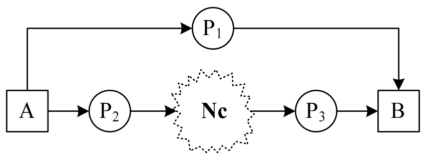


Fig. 4. Characteristics of a process model with an implicit dependency

Let us assume that in Figure 4 subnet N_c contains at least one task. It takes tokens from P_2 and puts tokens into P_3. In a general case, there may be more complicated relationships between N_c and the rest of the process model. However, only the simplest case is considered while other cases can be converted to this case easily (simply extending N_c). Therefore, we need not consider the cases where some tasks outside of N_c take P_2 as their input place or P_3 as their output place. Furthermore, if there are no other tasks connected to P_1, P_2 and P_3, P_1 becomes an implicit place. Implicit places do not influence the behavior of a process model, i.e., they can be removed without changing the set of reachable states. Clearly no mining algorithm is able to detect these places. Although their addition is harmless, we prefer mining algorithms that avoid constructing implicit places. Note that not all implicit dependencies correspond to implicit places. Therefore, we consider extensions of the basic case shown in Figure 4. These extensions add arcs to P_1, P_2, and P_3. In total we will consider seven extensions. These are shown in Figure 5. For example, Figure 5(a) extends Figure 4 by adding input arcs to P_2 and output arcs to P_3. Note that each of the "patterns" depicted in Figure 5 may appear in a sound WF-net.

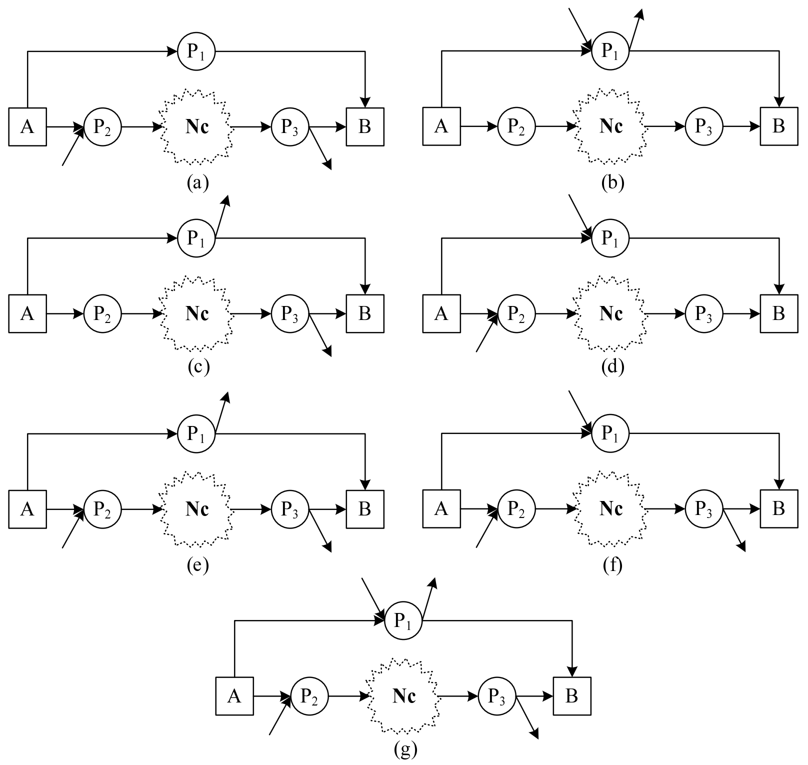


Fig. 5. Sound sub-WF-nets with implicit dependencies: (1) patterns (b) and (g) will be mined incorrectly because the α algorithm will create two places for P_1, (2) patterns (c), (d), (e) and (f) will be mined incorrectly because the α algorithm will miss some arcs, and (3) pattern (a) will be mined incorrectly because the α algorithm will not find place P_1.

In the remainder, we will show that it is possible to successfully mine processes embedding one or more of the patterns shown in Figure 5. Using existing algorithms such as the α algorithm [12, 41], the WF-net (a) shown in Figure 5 cannot be discovered, i.e., place P_1 and its surrounding arcs will not be mined correctly. For (b) and (g), place P_1 may be replaced by two or more places. For (c) and (e), the arc (P_1,B) will be omitted. For (d) and (f), the arc (A,P_1) will be omitted.

In this paper, we will consider three cases:

1. The situation described by patterns (b) and (g) in Figure 5, where the α algorithm incorrectly replaces place P_1 by two or more places.

2. The situation described by patterns (c), (d), (e) and (f), where the α algorithm misses the arc between A and P_1 (A, P_1) or P_1 and B (P_1, B).
3. The situation described by pattern (a) where place P_1 is not discovered at all.

In the next section, we will show how these three cases can be detected.

6 Detecting implicit dependencies

From the previous sections, it is obvious that the detection of implicit dependencies is the most important factor for mining process models with non-free-choice constructs correctly. In this section, we will introduce three methods to tackle the three problems illustrated by Figure 5 in detail. There exists a one-to-one relationship between the three methods and the above three cases of implicit dependencies.

To detect explicit dependencies between tasks, we adopt the α algorithm [12, 41]. Some definitions, such as $>_W$, $\rightarrow_W$, $\#_W$, $\|_W$, etc., are also borrowed from there with some modifications. Based on these basic ordering relations, we provide some additional new definitions for advanced ordering relations. The definition of one-loop-free workflow net directly adopts the definition with the same name presented in [41].

Definition 3 (Ordering relations). *Let N=(P,T,F) be a one-loop-free workflow net and W be an event log over T. Let a,b$\in$T:*

- *$a \triangle_W b$ iff there is a trace $\sigma = t_1t_2t_3 \ldots t_n$ and $i \in \{1, \ldots, n-2\}$ such that $\sigma \in W$ and $t_i = t_{i+2} = a$ and $t_{i+1} = b$,*
- *$a >_W b$ iff there is a trace $\sigma = t_1t_2t_3 \ldots t_n$ and $i \in \{1, \ldots, n-1\}$ such that $\sigma \in W$ and $t_i = a$ and $t_{i+1} = b$,*
- *$a \rightarrow_W b$ iff $a >_W b$ and ($b \not>_W a$ or $a \triangle_W b$ or $b \triangle_W a$),*
- *$a \#_W b$ iff $a \not>_W b$ and $b \not>_W a$,*
- *$a \|_W b$ iff $a >_W b$ and $b >_W a$ and $\neg(a \triangle_W b$ or $b \triangle_W a)$,*
- *$a \lhd_W b$ iff $a \#_W b$ and there is a task c such that $c \in T$ and $c \rightarrow_W a$ and $c \rightarrow_W b$,*
- *$a \rhd_W b$ iff $a \#_W b$ and there is a task c such that $c \in T$ and $a \rightarrow_W c$ and $b \rightarrow_W c$,*
- *$a \gg_W b$ iff $a \not>_W b$ and there is a trace $\sigma = t_1t_2t_3 \ldots t_n$ and $i, j \in \{1, \ldots, n\}$ such that $\sigma \in W$ and $i < j$ and $t_i = a$ and $t_j = b$ and for all $k \in \{i+1, \ldots, j-1\}$ satisfying $t_k \neq a$ and $t_k \neq b$ and $\neg(t_k \lhd_W a$ or $t_k \rhd_W a)$, and*
- *$a \succ_W b$ iff $a \rightarrow_W b$ or $a \gg_W b$.*

The definitions of $\triangle_W$, $>_W$ and $\#_W$ are the same as those defined in [41]. Definitions of $\rightarrow_W$ and $\|_W$ are a little different. Given a complete event log of a sound SWF-net [12, 41] and two tasks a and b, $a \triangle_W b$ and $b \triangle_W a$ must both come into existence. But for a one-loop-free event log of a sound WF-net, it is not always true. Now we will turn to the last five new definitions. Relation $\lhd_W$

corresponds to XOR-Split while relation $\rhd_W$ corresponds to XOR-Join. Relation $\gg_W$ represents that one task can only be indirectly followed by another task. Relation $\succ_W$ represents that one task can be followed by another task directly or indirectly. Consider the event log shown in Table 1, it can be represented as string sets, i.e., $\{T_1T_3T_4, T_2T_3T_5\}$. From this log, the following advanced ordering relations between tasks can be detected: $T_1 \rhd_W T_2$, $T_4 \lhd_W T_5$, $T_1 \gg_W T_4$ and $T_2 \gg_W T_5$.

To improve the correctness of a mining result, the quality of its corresponding event log is especially significant. Although other ordering relations can be derived from $>_W$, $\gg_W$ is a little special. $>_W$ reflects relations with the length of one, while $\gg_W$ is a relation whose length is two or more. They are both used in the following definition.

Definition 4 (Complete event log). *Let $N = (P, T, F)$ be a sound WF-net and W be an event log of N. W is complete iff:*

- *for any event log W' of N: $>_{W'} \subseteq >_W$, $\triangle_{W'} \subseteq \triangle_W$ and $\gg_{W'} \subseteq \gg_W$, and*
- *for any $t \in T$: there is a $\sigma \in W$ such that $t \in \sigma$.*

In this paper we assume perfect information: (i) the log must be complete (as defined above) and (ii) the log is noise free (i.e., each event registered in the log is correct and representative for the model that needs to be discovered). Some techniques to deal with incompleteness and noise will be discussed later.

Based on the above ordering relations defined in Definition 3. Some implicit ordering relations reflecting implicit dependencies can be derived.

Definition 5 (Implicit ordering relations). *Let W be a complete event log and $N = (P, T, F) = \alpha^+(W)$ be a mined WF-net from W using the α^+ algorithm. Let $a, b \in T$:*

- *$a \mapsto_{W^1} b$ iff $a \not\succ_W b$ and there is a task $c \in T$ such that there are two different places $p_1, p_2 \in P$ such that $p_1, p_2 \in \bullet c$ and $a \in \bullet p_1$ and $a \notin \bullet p_2$ and $b \in p_2 \bullet$ and there is no task $t \in \bullet p_2$ such that $t \succ_W a$ or $t \parallel_W a$,*
- *$a \mapsto_{W^{21}} b$ iff $a \gg_W b$ and $|a \bullet| > 1$ and there is a task $b' \in T$ such that $b \lhd_W b'$ and there is a place $p \in a\bullet$ such that there is no task $t \in p\bullet$ such that $t \succ_W b$ or $t \parallel_W b$ and there is a task $t' \in p\bullet$ such that $t' \succ_W b'$ or $t' \parallel_W b'$,*
- *$a \mapsto_{W^{22}} b$ iff $a \gg_W b$ and $|\bullet b| > 1$ and there is a task $a' \in T$ such that $a \rhd_W a'$ and there is a place $p \in \bullet b$ such that there is no task $t \in \bullet p$ such that $a \succ_W t$ or $a \parallel_W t$ and there is a task $t' \in \bullet p$ such that $a' \succ_W t'$ or $a' \parallel_W t'$,*
- *$a \mapsto_{W^2}$ iff $a \mapsto_{W^{21}} b$ or $a \mapsto_{W^{22}} b$, and*
- *$a \mapsto_{W^3} b$ iff there are two tasks $a', b' \in T$ such that $a \bullet \cap a' \bullet \neq \phi$ and $\bullet b \cap \bullet b' \neq \phi$ and $a \gg_W b$ and $a \not\gg_W b'$ and $a' \not\gg_W b$ and $a' \gg_W b'$ and $\bullet b \subseteq \bullet b' \cup \{\bullet t | a \not\gg_W t \wedge a' \gg_W t \wedge (b' \parallel_W t \vee b' \succ_W t) \wedge \bullet b \cap \bullet t \neq \phi\}$.*

First of all, we try to detect implicit dependencies from an event log of a process model with a sub-WF-net similar to Figure 5(b) and (g). $\mapsto_{W^1}$ insures that once there is a place connecting two successive tasks in the mined model and the latter task has more than one input place, the latter task can always have chance to be executed directly following the former task.

Secondly, we try to detect implicit dependencies from an event log of a process model with a sub-WF-net similar to Figure 5(c) to (f). $\mapsto_{W^2}$ insures that once a task takes tokens from one of multiple parallel branches, it together with its parallel tasks must consume tokens from other branches too.

Finally, we try to detect implicit dependencies from an event log of a process model with a sub-WF-net similar to Figure 5(a). $\mapsto_{W^3}$ insures that if two exclusive tasks (i.e., involved in an XOR-Join) lead to different sets of parallel branches and these two sets together with their tasks satisfy certain conditions, the mined WF-net is still sound.

7 Mining algorithm

In this section, we first analyze the interrelationships among the three implicit ordering relations proposed in the previous section. Then, we introduce two reduction rules for eliminating those implicit ordering relations leading to redundant implicit dependencies. Then, we give the mining algorithm named α^{++} for constructing process models with non-free-choice constructs involving implicit dependencies. Finally, we briefly discuss the complexity of the α^{++} algorithm.

7.1 Interrelationships among the three implicit ordering relations

The three implicit ordering relations proposed in the previous section are not independent of each other. There are a total of $3! = 6$ kinds of interrelationships among them. With the help of these interrelationships, the correct sequence of detecting these relations can be identified naturally.

First, we will clarify the influence of $\mapsto_{W^2}$ and $\mapsto_{W^3}$ on $\mapsto_{W^1}$. See Figure 6, we assume that $\mapsto_{W^2}$ will be treated as $\rightarrow_W$ before detecting $\mapsto_{W^1}$. Here a has two input places, i.e., p_2 and p_3. After detecting $\mapsto_{W^1}$, $t \mapsto_{W^1} b$ will be detected and there will be an arc connecting t and p_3. Clearly, the sub-WF-net is not sound in this case. Here we get a conclusion that detecting $\mapsto_{W^2}$ should not be executed before detecting $\mapsto_{W^1}$. Similarly, detecting $\mapsto_{W^3}$ should not be executed before detecting $\mapsto_{W^1}$ either.

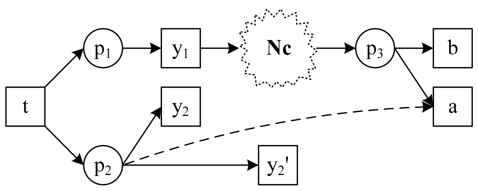


Fig. 6. Example for the influence of $\mapsto_{W^2}$ on $\mapsto_{W^1}$

Secondly, we will clarify the influence of $\mapsto_{W^1}$ and $\mapsto_{W^2}$ on $\mapsto_{W^3}$. From the essence of $\mapsto_{W^1}$, we can see that it does not produce any new $\rhd_W$ or $\lhd_W$ ordering relation. While $\mapsto_{W^3}$ fully involves all these two kinds of ordering relations between tasks. Here we get a conclusion that $\mapsto_{W^1}$ has no influence on $\mapsto_{W^3}$. On the contrary, $\mapsto_{W^2}$ always produces new $\rhd_W$ or $\lhd_W$ ordering relations. So detecting $\mapsto_{W^3}$ should not be executed before detecting $\mapsto_{W^2}$.

Finally, we will clarify the influence of $\mapsto_{W^1}$ and $\mapsto_{W^3}$ on $\mapsto_{W^2}$. The detection of $\mapsto_{W^2}$ depends on the parallel branches of some task and four kinds of advanced ordering relations (i.e., $\lhd_W$, $\rhd_W$, $\|_W$ and $\succ_W$). From the essence of $\mapsto_{W^1}$ and $\mapsto_{W^3}$, we can see that they do not affect any task's parallel branches and do not produce any new $\lhd_W$, $\rhd_W$, $\|_W$ or $\succ_W$ ordering relation. Here we get a conclusion that the detection of $\mapsto_{W^1}$ or $\mapsto_{W^3}$ does not influence that of $\mapsto_{W^2}$.

By now, we can identify the correct sequence of detecting the three kind of implicit ordering relations, i.e., $\mapsto_{W^1} > \mapsto_{W^2} > \mapsto_{W^3}$. When detecting these relations successively, the only important thing to remember is that all $\mapsto_{W^2}$ relations must be treated as $\rightarrow_W$ before detecting $\mapsto_{W^3}$.

7.2 Eliminating redundant implicit dependencies

Not all the implicit dependencies derived from the implicit ordering relations are meaningful to the mined process model. There may exist some implicit dependencies leading to implicit places, which are called *redundant implicit dependencies*. We will give two reduction rules to eliminate these implicit dependencies (i.e., eliminating the corresponding implicit ordering relations) in this subsection.

Figure 7 shows the first kind of mined WF-net involving redundant implicit dependencies. Here p_2 is an implicit place caused by $A \mapsto_{W^2} E$ that needs to be eliminated.

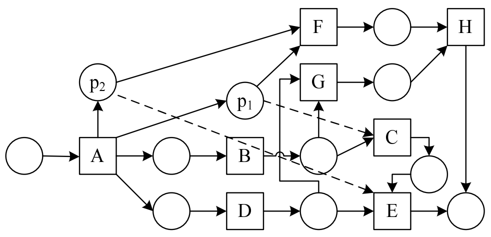


Fig. 7. The first kind of mined WF-net involving redundant implicit dependencies

Therefore, we need a reduction rule to eliminate this kind of redundant implicit dependencies after detecting $\mapsto_{W^2}$. We do this by eliminating the corresponding implicit ordering relations. This reduction rule named Rule 1 is formalized as follows.

$$\begin{array}{l} \forall_{a,b,c \in T_W} a \mapsto_{W^2} b \wedge a \mapsto_{W^2} c \wedge b \succ_W c \Rightarrow a \not\mapsto_{W^2} c \\ \forall_{a,b,c \in T_W} a \mapsto_{W^2} c \wedge b \mapsto_{W^2} c \wedge a \succ_W b \Rightarrow a \not\mapsto_{W^2} c \end{array} \tag{1}$$

Figure 8 shows the second kind of mined WF-net involving redundant implicit dependencies. Here either p_2 or p_3 is an implicit place caused by $A \mapsto_{W^3} H$ or $D \mapsto_{W^3} H$ respectively.

Therefore, we need a reduction rule to eliminate this kind of redundant implicit dependencies after detecting $\mapsto_{W^3}$. As Figure 8 shows, either p_2 or p_3 can

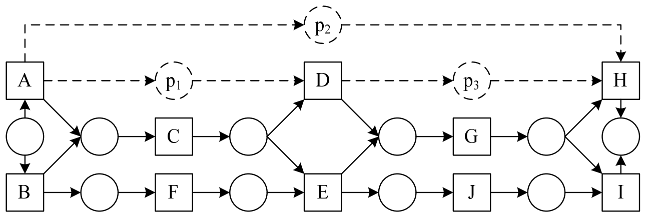


Fig. 8. The second kind of mined WF-net involving redundant implicit dependencies

be eliminated but they should not be both eliminated. Here we prefer to eliminate the implicit dependencies with longer distances according to the transitive closure of all the basic implicit dependencies. This reduction rule named Rule 2 is formalized as follows.

$$\begin{aligned}&\forall_{a,b\in T_W}(a \mapsto_{W^3} b \wedge \exists_{t_1,\cdots,t_n\in T_W}(n \geq 1 \wedge a \mapsto_{W^3} t_1 \wedge \cdots \wedge t_n \mapsto_{W^3} b))\\ &\Rightarrow a \not\mapsto_{W^3} b\end{aligned} \tag{2}$$

Consider again the WF-net shown in Figure 8, p_2 will be eliminated after applying Rule 2. Here $A \mapsto_{W^3} H$ is not a basic implicit dependency. It can be decomposed into $A \mapsto_{W^3} D$ and $D \mapsto_{W^3} H$. In this example, $A \mapsto_{W^3} D$ and $D \mapsto_{W^3} H$ are both basic implicit dependencies. The goal of this reduction rule is just to eliminate all non-basic implicit dependencies while keep the basic ones.

7.3 Constructing process models

By now, all the explicit and implicit dependencies can be detected correctly. It is necessary to give an algorithm that constructs the final mined process model. The solution to tackle length-one loops in sound WF-nets and some mining steps are borrowed from [41] with some modification. All the related ordering relations come from Definition 3.

The algorithm - called α^{++} - to mine sound WF-nets with non-free-choice constructs is formalized as follows. Note that the function *eliminateTask*(σ, t) maps any event trace σ to a new one σ' without the occurrence of a certain transition t [41]. Also note that *eliminateRDByRule1* and *eliminateRDByRule2* eliminate redundant implicit dependencies in any dependency set by applying Rule 1 and 2 respectively.

Definition 6 (Mining algorithm α^{++}). *Let W be a loop-complete event log over T. The $\alpha^{++}(W)$ is defined as follows.*

1. $T_{log} = \{t \in T | \exists_{\sigma\in W} t \in \sigma\}$
2. $L1L = \{t \in T_{log} | \exists_{\sigma=t_1t_2\cdots t_n\in W; i\in\{1,2,\cdots,n\}} t = t_{i-1} \wedge t = t_i\}$
3. $T' = T_{log} - L1L$
4. $X_W = \{(A,B,C) | A \subseteq T' \wedge B \subseteq T' \wedge C \subseteq L1L \wedge \forall_{a\in A}\forall_{c\in C}(a >_W c \wedge \neg(c \triangle_W a)) \wedge \forall_{b\in B}\forall_{c\in C}(c >_W b \wedge \neg(c \triangle_W b)) \wedge \forall_{a\in A}\forall_{b\in B} a \not\parallel_W b \wedge \forall_{a_1,a_2\in A} a_1 \#_W a_2 \wedge \forall_{b_1,b_2\in B} b_1 \#_W b_2\}$
5. $L_W = \{(A,B,C) \in X_W | \forall_{(A',B',C')\in X_W} A \subseteq A' \wedge B \subseteq B' \wedge C \subseteq C' \Rightarrow (A,B,C) = (A',B',C')\}$

```
6. $W^{-L1L} = \emptyset$
7. For each $\sigma \in W$ do:
   (a) $\sigma' = \sigma$
   (b) For each $t \in L1L$ do:
       i. $\sigma' := eliminateTask(\sigma', t)$
   (c) $W^{-L1L} := W^{-L1L} \cup \sigma'$
8. $ID_{W^1} = \{(a,b) | a \in T' \wedge b \in T' \wedge a \mapsto_{W^1} b\}$
9. $(P_{W^{-L1L}}, T_{W^{-L1L}}, F_{W^{-L1L}}) = \alpha(W^{-L1L})$
10. Treat each $a \mapsto_{W^1} b \in ID_{W^1}$ as $a \rightarrow_W b$ and $ID_{W^2} = \{(a,b) | a \in T' \wedge b \in T' \wedge a \mapsto_{W^2} b\}$
11. $ID_{W^2} := eliminateRDByRule1(ID_{W^2})$
12. $X_W = \{(A \cup A_2, B \cup B_2) | p_{(A,B)} \in P_{W^{-L1L}} \wedge A_2 \cup B_2 \neq \emptyset \wedge A \cap A_2 = \emptyset \wedge B \cap B_2 = \emptyset \wedge \forall_{a \in A} \forall_{b \in B_2} (a \mapsto_{W^1} b \vee a \mapsto_{W^2} b) \wedge \forall_{a \in A_2} \forall_{b \in B \cup B_2} (a \mapsto_{W^1} b \vee a \mapsto_{W^2} b) \wedge \forall_{a_1 \in A} \forall_{a_2 \in A_2} (a_2 \#_W a_1 \wedge a_2 \not\gg_W a_1) \wedge \forall_{b_1 \in B} \forall_{b_2 \in B_2} (b_1 \#_W b_2 \wedge b_1 \not\gg_W b_2)\}$
13. $Y_W = \{(A,B) | ((A,B) \in X_W \vee p_{(A,B)} \in P_{W^{-L1L}}) \wedge \forall_{(A',B') \in X_W \vee p_{(A',B')} \in P_{W^{-L1L}}} (A \subseteq A' \wedge B \subseteq B' \Rightarrow (A,B) = (A',B'))\}$
14. Treat each $a \mapsto_{W^2} b \in ID_{W^2}$ as $a \rightarrow_W b$ and $ID_{W^3} = \{(a,b) | a \in T' \wedge b \in T' \wedge a \mapsto_{W^3} b\}$
15. $ID_{W^3} := eliminateRDByRule2(ID_{W^3})$
16. $X_W = \{(A,B) | A \subseteq T' \wedge B \subseteq T' \wedge \forall_{a \in A} \forall_{b \in B} a \mapsto_{W^3} b \wedge \forall_{a_1, a_2 \in A} a_1 \#_W a_2 \wedge \forall_{b_1, b_2 \in B} b_1 \#_W b_2\}$
17. $Z_W = \{(A,B) \in X_W | \forall_{(A',B') \in X_W} A \subseteq A' \wedge B \subseteq B' \Rightarrow (A,B) = (A',B')\}$
18. $P_W = \{p_{(A,B)} | (A,B) \in Y_W \cup Z_W\} - \{p_{(A,B)} | \exists_{(A',B',C') \in L_W} A' = A \wedge B' = B\} \cup \{p_{(A \cup C, B \cup C)} | (A,B,C) \in L_W\}$
19. $T_W = T_{W^{-L1L}} \cup L1L$
20. $F_W = \{(a, p_{(A,B)}) | (A,B) \in P_W \wedge a \in A\} \cup \{(p_{(A,B)}, b) | (A,B) \in P_W \wedge b \in B\}$
21. $\alpha^{++}(W) = (P_W, T_W, F_W)$
```

The α^{++} works as follows. Steps 1 to 3 are directly borrowed from [41]. In steps 4 and 5, the places connecting length-one-loop transitions are identified and included in L_W. Then all length-one-loop transitions are removed from the input log W and the new input log W^{-L1L} to be processed by the α algorithm is derived (steps 6 and 7). In Step 8, all the implicit ordering relations $\mapsto_{W^1}$ in W^{-LlL} are detected. In Step 9, the α algorithm discovers a WF-net based on W^{-L1L} and the ordering relations as defined in Definition 3. In steps 10 to 13, all the places involving $\mapsto_{W^1}$ and $\mapsto_{W^2}$ relations are derived and included in Y_W. First, all the implicit dependencies $\mapsto_{W^2}$ in W^{-LlL} are detected once all the $\mapsto_{W^1}$ in ID_{W^1} have been treated as $\rightarrow_W$ (Step 10). Then Rule 1 is applied to reduce the redundant implicit dependencies in ID_{W^2} (Step 11). At last, all the places involving the first two kinds of implicit dependencies are derived from $P_{W^{-L1L}}$ while the other places in $P_{W^{-L1L}}$ are retained (steps 12 and 13). In steps 14 to 17, all the places involving $\mapsto_{W^3}$ relations are derived and included in Z_W. First, all the $\mapsto_{W^2}$ in ID_{W^2} are treated as $\rightarrow_W$ and all the implicit dependencies $\mapsto_{W^3}$ in W^{-LlL} are detected (Step 14). Then Rule 2 is applied to reduce the redundant implicit dependencies in ID_{W^3} (Step 15). At last, all the places involving the third kind of implicit dependency are derived based on these

$\mapsto_{W^3}$ relations (steps 16 and 17). In steps 18 to 20, all the places in the mined WF-net are gathered and the length-one-loop transitions are added to the net and all the arcs of the net are derived too. The WF-net with non-free-choice constructs as well as length-one-loops and implicit dependencies is returned in Step 21.

7.4 Complexity of the α^{++} algorithm

To conclude this section, we consider the complexity of the α^{++} algorithm. For a complex process model, its complete event log may be huge containing millions of events. Fortunately, the α^{++} algorithm is driven by relations $>_W$, $\triangle_W$ and $\gg_W$. The time it takes to build relations $>_W$, $\triangle_W$ and $\gg_W$ is linear in the size of the log. Moreover, we only require the log to be complete with respect to these relations, i.e., we do not need logs that capture all possible traces. The complexity of the remaining steps in the α^{++} algorithm is exponential in the number of tasks. However, note that the number of tasks is typically less than 100 and does not depend on the size of the log. Therefore, the complexity is not a bottleneck for large-scale application [12].

Practical experiences show that process models of up to 25 tasks based on logs of about half a million events can be analyzed within one minute on a standard computer. In the next section we will give an example of this size. Based on real-life logs we also experienced that the α^{++} algorithm is typically *not* the limiting factor. The real limiting factor is the visualization of the model and the interpretation of the model by the analyst. Although the α^{++} algorithm is able to construct much larger models, people have problems comprehending such models (especially when the layout is machine generated). Therefore, ProM offers an extensive set of filtering mechanisms to collapse part of the process into a single node or to abstract from less frequent paths and tasks.

8 Experimental evaluation

The α^{++} algorithm has been implemented as a ProM plug-in and can be downloaded from www.processmining.org. As shown in Section 1, ProM is a general process mining framework [25]. It takes an event log in the standard XML format (MXML) as input and uses a process mining plug-in to mine a process model from that log. A screenshot of ProM is shown in Figure 9. The screenshot shows the Petri net constructed by the α^{++} algorithm. The screenshot also shows that the result can be automatically translated to an Event-driven Process Chain (EPC, [36, 49]) and that the result can be analyzed for soundness. In fact the result can be exported to tools such as CPN Tools, ARIS, YAWL, ARIS PPM, Yasper, EPC Tools, Woflan, etc.

A lot of experiments have been done to evaluate the proposed methods together with the implemented algorithm in the previous sections. The α^{++} plug-in of ProM has been applied to several real-life logs and smaller artificial logs.

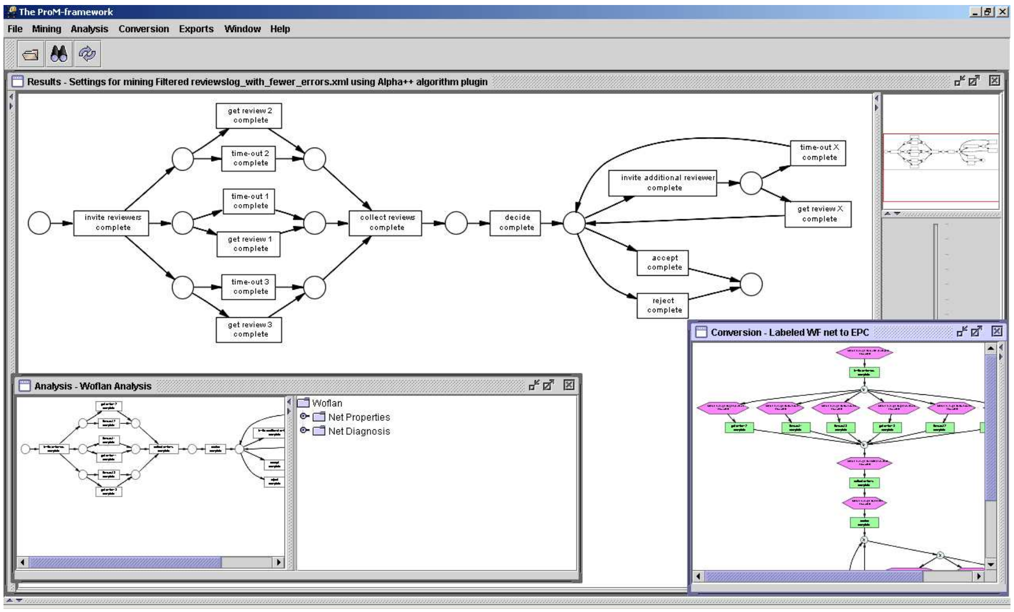


Fig. 9. A screenshot of ProM showing the result of applying the α^{++} algorithm

This section shows the application of the α^{++} algorithm to smaller artificial logs and discusses the application of large more realistic logs. Moreover, we also discuss the limitation of the α^{++} algorithm.

8.1 Evaluation based on smaller artificial logs

Instead of showing large real-life models, we first focus on smaller artificial examples that demonstrate the fact that α^{++} significantly improves existing approaches such as the classic α algorithm.

To illustrate the capabilities of the α^{++} plug-in, we first show some experimental results for models with implicit dependencies.

Figure 10(a) shows an original WF-net. One of its complete workflow log is $\{ABC, ABDEC, ADBEC, ADEBC, ABDEDEC\}$. After applying α^{+} algorithm on this log, the mined model is similar to Figure 10(b) except for the two dotted arcs. Based on this net and the corresponding log, $A \mapsto_{W^1} C$ is detected. Thus p_1 and p_2 should be merged together. The resulting mined model will be the same as the original one, i.e., the α^{++} algorithm is able to correctly discover processes such as the one shown in Figure 10(a).

Figure 11 shows the effect of detecting $\mapsto_{W^2}$. The WF-nets excluding the dotted arcs are mined by α^{+} algorithm. The dotted arcs correspond to the detected implicit dependency relation $\mapsto_{W^2}$. Thus the WF-nets in Figure 11 can all be discovered correctly by the α^{++} algorithm. For Figure 11(a), the corresponding complete workflow log is $\{ABCE, ACBE, ADE\}$. From this log, no implicit dependency is detected. For Figure 11(b), the corresponding complete workflow log is $\{ACFBGE, AFCBGE, AFBCGE, AFBGCE, AFDGE\}$. From this log,

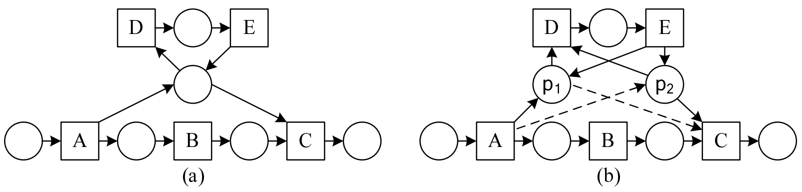


Fig. 10. Detecting implicit dependency relation $\mapsto_{W^1}$

implicit dependencies $A \mapsto_{W^2} D$ and $D \mapsto_{W^2} E$ are detected. For Figure 11(c), the corresponding complete workflow log is $\{ACD, BCE, AFCE, ACFE\}$. From this log, implicit dependencies $A \mapsto_{W^2} D$ and $B \mapsto_{W^2} E$ are detected. For Figure 11(d), the corresponding complete workflow log is $\{ABC, ABDE, ADBE\}$. From this log, implicit dependency $A \mapsto_{W^2} C$ is detected.

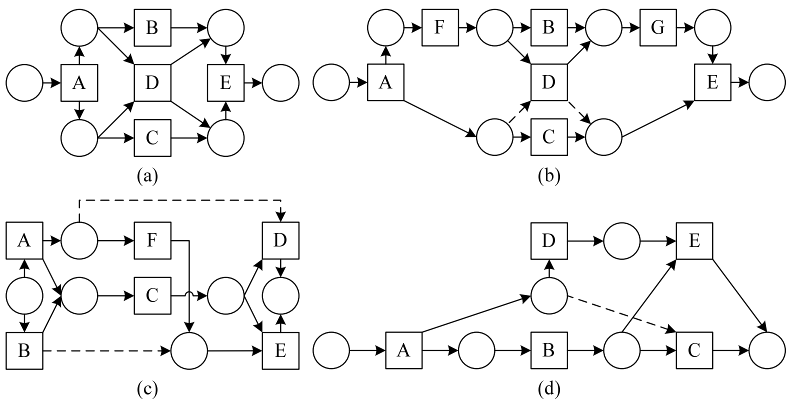


Fig. 11. Detecting implicit dependency relation $\mapsto_{W^2}$

Figure 12 shows the effect of detecting $\mapsto_{W^3}$. All the implicit dependencies in the WF-nets are detected successfully from the corresponding logs. For Figure 12(a), the corresponding complete workflow log is $\{ACD, ACE, BCD, BCE\}$. From this log, no implicit dependency is detected. If the log changes to $\{ACD, BCE\}$, implicit dependencies $A \mapsto_{W^3} D$ and $B \mapsto_{W^3} E$ can be detected. For Figure 12(b), the corresponding complete workflow log is $\{ACFD, AFCD, BCGE, BGCE\}$. From this log, no implicit dependency is detected either. For Figure 12(c), the corresponding complete workflow log is $\{ACD, BCFE, BFCE\}$. From this log, implicit dependency $A \mapsto_{W^3} D$ is detected. For Figure 12(d), the corresponding complete workflow log is $\{ACEBCD\}$. From this log, implicit dependencies $A \mapsto_{W^3} E$ and $B \mapsto_{W^3} D$ are detected.

Figure 13(a) shows the effect of detecting $\mapsto_{W^2}$ and $\mapsto_{W^3}$ successively. Figure 13(b) shows the effect of detecting $\mapsto_{W^1}$ and $\mapsto_{W^3}$ successively. The mined WF-nets with implicit dependencies are the same as the original ones, i.e., the α^{++} algorithm is able to correctly discover processes such as the ones shown in Figure 13. For Figure 13(a), the corresponding complete workflow log is $\{ACDEGH, ACDGEH, ACGDEH, BCDFH\}$. From this log, implicit dependencies $C \mapsto_{W^2} F$,$A \mapsto_{W^3} E$,$A \mapsto_{W^3} G$ and $B \mapsto_{W^3} F$ are detected. For Fig-

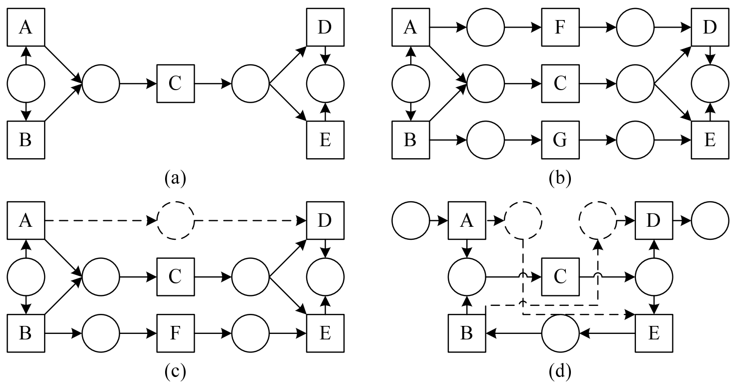


Fig. 12. Detecting implicit dependency relation $\mapsto_{W^3}$

ure 13(b), the corresponding complete workflow log is $\{FBG, ABC, FDBEG, FBDEG, FDEBG, ADEDEBG, ABDEC\}$. From this log, implicit dependencies $A \mapsto_{W^1} C$,$F \mapsto_{W^1} G$,$A \mapsto_{W^3} C$ and $F \mapsto_{W^3} G$ are detected.

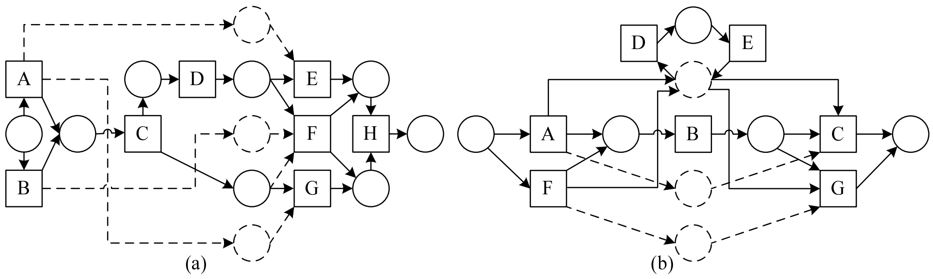


Fig. 13. Detecting implicit dependency relations $\mapsto_{W^2}$ and $\mapsto_{W^3}$ as well as $\mapsto_{W^1}$ and $\mapsto_{W^3}$

8.2 Evaluation based on real-life logs

The above experimental results show that our algorithm is powerful enough to detect implicit dependencies between tasks. Now we use a more realistic example given in Figure 14 to show the applicability of the α^{++} algorithm. This process model was discovered based on a log containing 29502 event traces (i.e., cases) and 416586 events. In the resulting model there are 26 different tasks. It takes about one minute for the α^{++} algorithm to discover the model shown in Figure 14. The dotted arcs reflect the implicit dependencies between tasks detected by the α^{++} algorithm, i.e., *Contact outsource organization*$\mapsto_{W^2}$*Send bill*, *Repair car on the spot RSM*$\mapsto_{W^2}$*Send bill* and *Repair car on the spot ASM*$\mapsto_{W^2}$*Send bill*.

The process model shown in Figure 14 is taken from a set of 25 realistic process models. These models were constructed by students in group projects where they had to model real life business processes. Each of the models has a size and complexity comparable to Figure 14. The processes were modeled using the

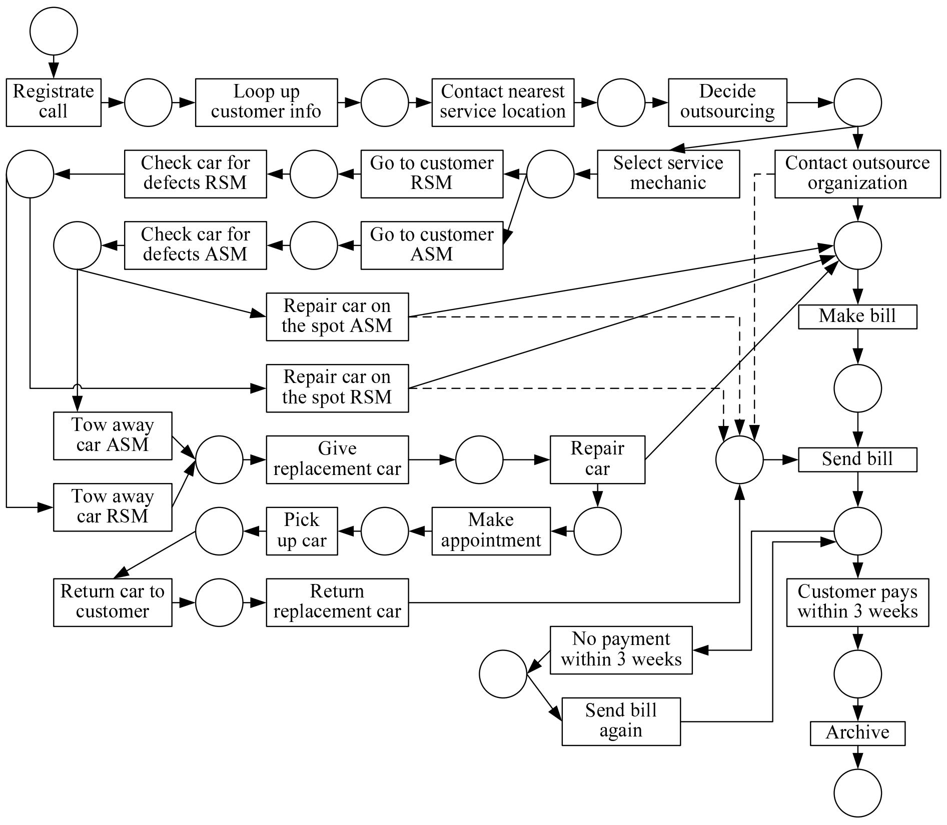


Fig. 14. A realistic example of repairing car including implicit dependencies

tool Protos [43]. Protos is the most widely used business process modeling tool in the Netherlands. Currently it is used by about 1500 organizations in more than 20 countries, e.g., more than half of all municipalities within the Netherlands use Protos for the specification of their in-house business processes. In each of the group projects a different real-life case was selected and the students modeled the corresponding processes. It is important to note that none of the authors was involved in the modeling of these processes. These models were automatically transferred to the simulation tool CPN Tools [21]. By using the ProMimport tool (promimport.sourceforge.net), the simulation logs of CPN Tools were converted into MXML logs. All of the steps were carried out automatically without passing any explicit process information into the logs. We used the logs of these 25 realistic process models to evaluate the α^{++} algorithm. Each of these logs was complete and *for each of the 25 logs the α^{++} algorithm was able to discover the corresponding process model correctly.* All the implicit dependencies between tasks hidden in the logs are detected successfully by the α^{++} algorithm.

We also applied the α^{++} algorithm to several other real-life logs. We have MXML logs from various organizations (ranging from hospitals and governmental organizations to a manufacturer of wafer steppers). These experiences show that as long as the log is complete, the α^{++} algorithm is able to discover a suitable model.

8.3 Limitations

Despite the successful application of the α^{++} algorithm to many artificial and real-life event logs, not all sound WF-nets can be successfully derived from their corresponding event logs. In the remainder of this section, we discuss some exceptional situations in which the α^{++} algorithm fails.

Consider the two sound WF-nets N_4 and N_5 shown in Figure 15. Their derived nets N_4' and N_5' shown in the shadow are not the same as the original. For N_4, one of its complete event logs is $\{ABCE, ACBE, ABDDCE\}$. After applying α^{++} algorithm on that log, N_4' is derived. There is an implicit dependency between A and D as well as between D and E in N_4. The task D is involved in both a length-one-loop and two implicit dependencies. In Definition 6, it is assumed that none task in length-one-loop is involved in any implicit dependency. Thus the places connected to D can not be detected correctly in steps 4 and 5 of the definition. The mined net N_4' is not a WF-net because D is not connected. The behavior of N_4' is not the same as that of N_4 either. For N_5, similar thing happens. There are two implicit dependencies between A and D as well as between H and E in N_5. Although N_5' is a sound WF-net, it is not behavioral equivalent with N_5. Although mining such WF-nets is difficult, it is possible to correctly mine them using the α^{++} algorithm after a minor modification. According to Definition 5, $A \mapsto_{W^2} D$ and $D \mapsto_{W^2} E$ can be detected from the log of N_4 as well as $A \mapsto_{W^2} D$ and $H \mapsto_{W^2} E$ from that of N_5. With this minor modification, the α^{++} algorithm is still powerful enough to mine such WF-nets correctly based on definitions 3 and 5.

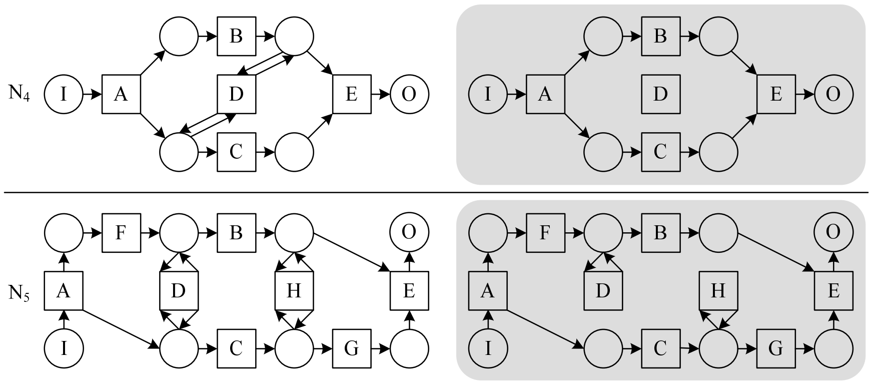


Fig. 15. WF-nets with length-one-loops involving implicit dependencies

There are also a few WF-nets which could not be derived from their complete event logs correctly even after the α^{++} algorithm is modified as discussed before. Maybe some more advanced ordering relations introduced in the future can handle these cases. See N_6 and N_7 in Figure 16. Their derived nets using the α^{++} algorithm are not sound any more. For N_6, one of its complete event logs is $\{ABDEHFI,\ ADBEHFI,\ ACDFGEI,\ ADCFGEI\}$. Although there are many choices to make in N_6, only the choice between B and C is free-choice. In one event trace, once B or C is chosen to execute, the remaining execution sequence is determined. The ordering relations between H and E, G and F, E and G, and F and H are too difficult for any of today's mining algorithm. For N_7, one of its complete event logs is $\{ABCE, ACDF, ADBG\}$. All generated event traces are based upon the choice between B, C and D. The ordering relations between B and B, C and C, D and D, A and E, A and F, and A and G are even more difficult to mine.

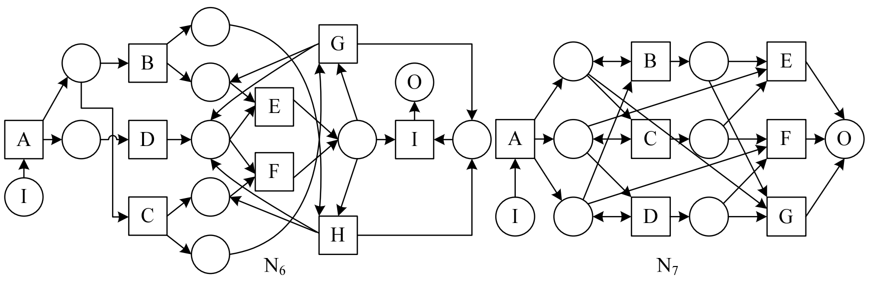


Fig. 16. Sample WF-nets leading to mining failure

The above examples refer to rather complex structures that are difficult to mine but at the same time are rather rare. *More important problems from a practical point of view are issues related to noise and completeness.* The α^{++} heavily relies on a particular notion of completeness, i.e., if two tasks can follow one another they should follow one another at least once in the entire log. Note that this is a much weaker notion of completeness than used by the classical approaches (which typically require completeness in terms of sequences). Nevertheless, even our weaker form of completeness is not realistic in case there is a lot of possible concurrency. This is not so much a problem of the α^{++} algorithm, i.e., it reveals a fundamental problem related to process mining. This problem is that it is impossible to discover behavior that did not yet happen because the observation period was too short. The other problem is the problem of noise. Noise may refer to exceptions of incorrectly logged events. The only way to address this is to filter away less frequent behavior. ProM offer a wide variety of filters and plug-ins able to deal with noise. Nevertheless, the problem is similar to the problem of completeness. How to distinguish regular from irregular behavior? Therefore, both issues suggest a more interactive form of process mining where an analyst is guiding the process mining algorithms to deal with incompleteness and noise.

9 Conclusion and future work

Process mining offers a new and exciting way to extract valuable information from event logs. In this paper, we have focused on process discovery, i.e., deriving a process model able to explain the observed behavior. This is interesting in many domains, e.g., discovering careflows in hospitals, monitoring web services, following maintenance processes, analyzing software development processes, etc. Although several process discovery techniques have been developed, implemented and applied successfully, they are unable to correctly mine certain processes. All of the existing techniques have problems dealing with implicit dependencies that may result from processes exhibiting non-free-choice behavior. Since real-life processes have such implicit dependencies, it is a highly relevant problem.

This paper describes an approach that is able to successfully mine a certain class of implicit dependencies, i.e., some non-free-choice Petri nets can be discovered correctly. Hence, it is a considerable improvement over existing approaches. The resulting α^{++} algorithm has been implemented and tested on a wide variety of logs, i.e., real-life logs and artificial logs. These experimental evaluations show that the approach is indeed able to detect implicit dependencies between tasks.

Our future work will focus on the application of the α^{++} algorithm to more real-life processes. Some techniques to deal with incompleteness should be explored. Moreover, we also want to address other open problems in the process-mining domain, e.g., invisible tasks (e.g., the skipping of tasks that is not recorded), duplicate tasks (i.e., different tasks in the model cannot be distinguished in the log), noise (e.g., dealing with exceptional behavior or incorrect logs), etc. In fact, we invite other researchers and tool developers to join us in this endeavor. The ProM framework provides a plugable open-source environment which makes it easy to develop alternative process mining algorithms.

Acknowledgements

The authors would like to thank Ton Weijters, Boudewijn van Dongen, Ana Karla Alves de Medeiros, Anne Rozinat, Christian Günter, Eric Verbeek, Minseok Song, Ronny Mans, Laura Maruster, Monique Jansen-Vullers, Hajo Reijers, Michael Rosemann, Huub de Beer, Peter van den Brand, Andriy Nikolov, Wouter Kunst, Martijn van Giessel et al. for their on-going work on process mining techniques. We also thank EIT, STW, and NWO for supporting the development of the ProM framework, cf. www.processmining.org.

This work is supported by the 973 Project of China (No. 2002CB312006) and the Project of National Natural Science Foundation of China (No. 60373011).

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. Business Alignment: Using Process Mining as a Tool for Delta Analysis. In J. Grundspenkis and M. Kirikova, editors, *Proceedings of the 5th Workshop on Business Process Modeling, Development and Support (BPMDS'04)*, volume 2 of *Caise'04 Workshops*, pages 138–145. Riga Technical University, Latvia, 2004.
3. W.M.P. van der Aalst. Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 1–65. Springer-Verlag, Berlin, 2004.
4. W.M.P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and H.M.W. Verbeek. Choreography Conformance Checking: An Approach based on BPEL and Petri Nets (extended version). BPM Center Report BPM-05-25, BPMcenter.org, 2005.
5. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems.* MIT press, Cambridge, MA, 2002.
6. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.
7. W.M.P. van der Aalst, A.K. Alves de Medeiros, and A.J.M.M. Weijters. Genetic Process Mining. In G. Ciardo and P. Darondeau, editors, *Applications and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 48–69. Springer-Verlag, Berlin, 2005.
8. W.M.P. van der Aalst and A.K.A. de Medeiros. Process Mining and Security: Detecting Anomalous Process Executions and Checking Process Conformance. In N. Busi, R. Gorrieri, and F. Martinelli, editors, *Second International Workshop on Security Issues with Petri Nets and other Computational Models (WISP 2004)*, pages 69–84. STAR, Servizio Tipografico Area della Ricerca, CNR Pisa, Italy, 2004.
9. W.M.P. van der Aalst and M. Song. Mining Social Networks: Uncovering Interaction Patterns in Business Processes. In J. Desel, B. Pernici, and M. Weske, editors, *International Conference on Business Process Management (BPM 2004)*, volume 3080 of *Lecture Notes in Computer Science*, pages 244–260. Springer-Verlag, Berlin, 2004.
10. 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.
11. 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.

12. 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.
13. 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.
14. 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.
15. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1. Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation, 2003.
16. A.W. Biermann and J.A. Feldman. A Survey of Results in Grammatical Inference. In S. Watanabe, editor, *Frontiers of Pattern Recognition*, pages 31–54. Academic Press, 1972.
17. A.W. Biermann and J.A. Feldman. On the Synthesis of Finite-State Machines from Samples of their Behavior. *IEEE Transaction on Computers*, 21:592–597, 1972.
18. J.E. Cook and Z. Du. Discovering thread interactions in a concurrent system. *Journal of Systems and Software*, 77(3):285–297, 2005.
19. J.E. Cook, Z. Du, C. Liu, and A.L. Wolf. Discovering models of behavior for concurrent workflows. *Computers in Industry*, 53(3):297–319, 2004.
20. 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.
21. CPN Group, University of Aarhus, Denmark. CPN Tools Home Page. http://wiki.daimi.au.dk/cpntools/.
22. A. Datta. Automating the Discovery of As-Is Business Process Models: Probabilistic and Algorithmic Approaches. *Information Systems Research*, 9(3):275–301, 1998.
23. J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1995.
24. J. Desel, W. Reisig, and G. Rozenberg, editors. *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2004.
25. B. 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, *Application and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 444–454. Springer-Verlag, Berlin, 2005.
26. B.F. van Dongen and W.M.P. van der Aalst. A Meta Model for Process Mining Data. In J. Casto and E. Teniente, editors, *Proceedings of the CAiSE'05 Workshops (EMOI-INTEROP Workshop)*, volume 2, pages 309–320. FEUP, Porto, Portugal, 2005.
27. M. Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede. *Process-Aware Information Systems: Bridging People and Software through Process Technology*. Wiley & Sons, 2005.
28. A. Ehrenfeucht and G. Rozenberg. Partial (Set) 2-Structures - Part 1 and Part 2. *Acta Informatica*, 27(4):315–368, 1989.

29. G. Greco, A. Guzzo, L. Pontieri, and D. Saccá. Mining Expressive Process Models by Clustering Workflow Traces. In *Proc of Advances in Kowledge Discovery and Data Mining, 8th Pacific-Asia Conference (PAKDD 2004)*, pages 52–62, 2004.
30. D. Grigori, F. Casati, M. Castellanos, U. Dayal, M. Sayal, and M.C. Shan. Business process intelligence. *Computers in Industry*, 53(3):321–343, 2004.
31. D. Grigori, F. Casati, U. Dayal, and M.C. Shan. Improving Business Process Quality through Exception Understanding, Prediction, and Prevention. In P. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. Snodgrass, editors, *Proceedings of 27th International Conference on Very Large Data Bases (VLDB'01)*, pages 159–168. Morgan Kaufmann, 2001.
32. D. Harel, H. Kugler, and A. Pnueli. Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. In *Formal Methods in Software and Systems Modeling*, volume 3393 of *Lecture Notes in Computer Science*, pages 309–324. Springer-Verlag, Berlin, 2005.
33. J. Herbst. A Machine Learning Approach to Workflow Management. In *Proceedings 11th European Conference on Machine Learning*, volume 1810 of *Lecture Notes in Computer Science*, pages 183–194. Springer-Verlag, Berlin, 2000.
34. IDS Scheer. ARIS Process Performance Manager (ARIS PPM): Measure, Analyze and Optimize Your Business Process Performance (whitepaper). IDS Scheer, Saarbruecken, Gemany, http://www.ids-scheer.com, 2002.
35. S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation.* International Thomson Computer Press, London, UK, 1996.
36. G. Keller, M. Nüttgens, and A.W. Scheer. Semantische Processmodellierung auf der Grundlage Ereignisgesteuerter Processketten (EPK). Veröffentlichungen des Instituts für Wirtschaftsinformatik, Heft 89 (in German), University of Saarland, Saarbrücken, 1992.
37. F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques.* Prentice-Hall PTR, Upper Saddle River, New Jersey, USA, 1999.
38. H. Liang, J. Dingel, and Z. Diskin. A Comparative Survey of Scenario-Based to State-Based Model Synthesis Approaches. In *Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM06)*, pages 5–12, New York, NY, USA, 2006. ACM Press.
39. A.K. Alves de Medeiros and C.W. Guenther. Process Mining: Using CPN Tools to Create Test Logs for Mining Algorithms. In K. Jensen, editor, *Proceedings of the Sixth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2005)*, volume 576 of *DAIMI*, pages 177–190, Aarhus, Denmark, October 2005. University of Aarhus.
40. 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.
41. 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 (UMICS 2004)*, volume 3272 of *Lecture Notes in Computer Science*, pages 154–168. Springer-Verlag, Berlin, 2004.
42. M. zur Mühlen and M. Rosemann. Workflow-based Process Monitoring and Controlling - Technical and Organizational Issues. In R. Sprague, editor, *Proceedings*

of the 33rd Hawaii International Conference on System Science (HICSS-33), pages 1–10. IEEE Computer Society Press, Los Alamitos, California, 2000.

43. Pallas Athena. *Protos User Manual.* Pallas Athena BV, Plasmolen, The Netherlands, 2004.
44. R. Parekh and V. Honavar. An Incremental Interactive Algorithm for Regular Grammar Inference. In *International Colloquium on Grammatical Inference: Learning Syntax from Sentences (ICGI 1996)*, volume 1147 of *Lecture Notes in Computer Science*, pages 238–249. Springer-Verlag, Berlin, 1996.
45. R. Parekh and V.G. Honavar. Learning DFA from Simple Examples. *Machine Learning*, 44(1-2):9–35, 2001.
46. 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.
47. P. Sarbanes, G. Oxley, and et al. Sarbanes-Oxley Act of 2002, 2002.
48. M. Sayal, F. Casati, U. Dayal, and M.C. Shan. Business Process Cockpit. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB'02)*, pages 880–883. Morgan Kaufmann, 2002.
49. A.W. Scheer. *ARIS: Business Process Modelling.* Springer-Verlag, Berlin, 2000.
50. J. Scott. *Social Network Analysis.* Sage, Newbury Park CA, 1992.
51. TIBCO. TIBCO Staffware Process Monitor (SPM). http://www.tibco.com, 2005.
52. S. Wasserman and K. Faust. *Social Network Analysis: Methods and Applications.* Cambridge University Press, Cambridge, 1994.
53. A.J.M.M. Weijters and W.M.P. van der Aalst. Workflow Mining: Discovering Workflow Models from Event-Based Data. In C. Dousson, F. Höppner, and R. Quiniou, editors, *Proceedings of the ECAI Workshop on Knowledge Discovery and Spatial Data*, pages 78–84, 2002.
54. 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.
55. L. Wen, J. Wang, W.M.P. van der Aalst, Z. Wang, and J. Sun. A Novel Approach for Process Mining Based on Event Types. BETA Working Paper Series, WP 118, Eindhoven University of Technology, Eindhoven, 2004.
56. L. Wen, J. Wang, and J. 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.