Rediscovering Workflow Models from Event-Based Data using Little Thumb

A.J.M.M. Weijters
a.j.m.m.weijters@tm.tue.nl

W.M.P van der Aalst
w.m.p.v.d.aalst@tm.tue.nl

Department of Technology Management, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands, +31 40 2473857/2290

Abstract

Contemporary workflow management systems are driven by explicit process models, i.e., a completely specified workflow design is required in order to enact a given workflow process. Creating a workflow design is a complicated time-consuming process and typically, there are discrepancies between the actual workflow processes and the processes as perceived by the management. Therefore, we propose a technique for rediscovering workflow models. This technique uses workflow logs to discover the workflow process as it is actually being executed. The workflow log contains information about events taking place. We assume that these events are totally ordered and each event refers to one task being executed for a single case. This information can easily be extracted from transactional information systems (e.g., Enterprise Resource Planning systems such as SAP and Baan). The rediscovering technique proposed in this paper can deal with noise and can also be used to validate workflow processes by uncovering and measuring the discrepancies between prescriptive models and actual process executions.

Keywords: Process mining, Workflow mining, Knowledge discovery, Petri nets

1. Introduction

During the last decade workflow management concepts and technology [2,19,23] have been applied in many enterprise information systems. Workflow management systems such as Staffware, IBM MQSeries, COSA, etc. offer generic modeling and enactment capabilities for structured business processes. By making graphical process definitions, i.e., models describing the life-cycle of a

typical case (workflow instance) in isolation, one can configure these systems to support business processes. Besides pure workflow management systems, many other software systems have adopted workflow technology. Consider for example ERP (Enterprise Resource Planning) systems such as SAP, PeopleSoft, Baan and Oracle, CRM (Customer Relationship Management) software, etc. Another example is the field of web services. All web services composition languages such BPEL4WS, BPML, XLANG, WSFL, WSCI, etc. have adopted most workflow concepts. Despite its promise, many problems are encountered when applying workflow technology. As indicated by many authors, workflow management systems are too restrictive and have problems dealing with change [2,4,7,12,20]. Many workshops and special issues of journals have been devoted to techniques to make workflow management more flexible (e.g., [2,4,20]). Most of the research in this area is devoted to techniques to increase the flexibility either by allowing for ad-hoc changes (as reflected by workflow management systems such as InConcert [18]) or by providing mechanisms to migrate cases and evolve workflows [7,12].

In this paper we take a different perspective with respect to the problems related to flexibility. We argue that many problems are resulting from a discrepancy between workflow *design* (i.e., the construction of predefined workflow models) and workflow *enactment* (the actual execution of workflows). Workflow designs are typically made by a small group of consultants, managers and specialists. As a result, the initial design of a workflow is often incomplete, subjective, and at a too high level. During the implementation of the workflow, i.e., configuring the workflow management system and training the workers involved, these issues cause many problems ("The devil is in the details"). Therefore, we propose to "reverse the process". Instead of starting with a workflow design, we start by gathering information about the workflow 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 workflow), (ii) each event refers to a case (i.e., a workflow instance), and (iii) events are totally ordered. Any information system using transactional systems such as ERP, CRM, or workflow management systems will offer this information in some form. Note that we do not assume the presence of a workflow management system. The only assumption we make, is that it is possible to construct workflow logs with event data. These workflow logs are used to construct a process specification, which adequately models the behavior registered. We use the term *process mining* for the method of distilling a structured process description from a set of real executions. In this paper, we propose a new technique for process mining.

The remainder of this paper is as follows. First we discuss related work. Section 3 introduces some preliminaries including a modeling language for workflow processes and the definition of a workflow log. Then we present a new technique for process mining (Section 4) and the implementation of this technique in the workflow-mining tool Little Thumb (Section 5). In Section 6 we present our experimental results. Finally, we conclude the paper by summarizing the main results and pointing out future work.

2. Related work

The idea of process mining is not new [6,8,9,10,14,15,16,17,21,22,24,25,26]. Cook and Wolf have investigated similar issues in the context of software engineering processes. In [8] 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 [8] are limited to sequential behavior. Cook and Wolf extend their work to concurrent processes in [9]. They also propose specific metrics (entropy, event type counts, periodicity, and causality) and use these metrics to discover models out of event streams. This approach is similar to the one presented in this paper. However, our metrics are quite different and our final goal is to find explicit representations for a broad range of process models, i.e., we generate a concrete Petri net rather than a set of dependency relations between events. In [10], 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 [6]. 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 the approach presented in this paper. Given the nature of workflow graphs there is no need to identify the nature (AND or OR) of joins and splits. Moreover, workflow graphs are acyclic. The only way to deal with

iteration is to enumerate all occurrences of a given activity. In [22], a tool based on these algorithms is presented.

Schimm [24] has developed a tool to discover hierarchically structured workflows. His approach requires all splits and joins to be balanced.

Herbst and Karagiannis also address the issue of process mining in the context of workflow management [14,15,16,17]. The approach uses the ADONIS modeling language and is based on hidden Markov models where models are merged and split in order to discover the underlying process. The work presented in [14,16,17] is limited to sequential models. A notable difference with other approaches is that the same activity can appear multiple times in the workflow model. The result in [15] incorporates concurrency but also assumes that workflow logs contain explicit causal information. The latter technique is similar to [6,22] and suffers from the drawback that the nature of splits and joins (i.e., AND or OR) is not discovered.

Compared to existing work we focus on workflow processes with concurrent behavior, i.e., detecting concurrency is one of our prime concerns [25]. Therefore, we want to distinguish AND/OR splits/joins explicitly. To reach this goal we combine techniques from machine learning with WorkFlow nets (WF-nets, [1]). WF-nets are a subset of Petri nets. Note that Petri nets provide a graphical but formal language designed for modeling concurrency. Moreover, the correspondence between commercial workflow management systems and WF-nets is well understood [1,2,13,19,23].

3. Preliminaries: Workflow nets and event logs

Workflows are by definition *case-based*, i.e., every piece of work is executed for a specific *case*. Examples of cases are a mortgage, an insurance claim, a tax declaration, an order, or a request for information. The goal of workflow management is to handle cases as efficient and effective as possible. A workflow process is designed to handle similar cases. Cases are handled by executing *tasks* in a specific order. The workflow process model specifies which tasks need to be executed and in what order. Alternative terms for such a model are: ‘procedure’, ‘workflow graph’, ‘flow diagram’ and ‘routing definition’. In the workflow process model, routing elements are used to describe sequential, conditional, parallel and iterative routing thus specifying the appropriate route of a case [19,23]. Many cases can be handled by following the same workflow process definition. As a result, the same task has to be executed for many cases.

Petri nets [11] have been proposed for modeling workflow process definitions long before the term ”workflow management'” was coined and workflow management systems became readily available. Consider for example Information Control Nets, a variant of the classical Petri nets, already introduced in the late seventies [13]. Petri nets constitute a good starting point for a solid theoretical foundation of workflow management. Clearly, a Petri net can be used to specify the routing of cases (workflow instances). *Tasks* are modeled by *transitions*, and *places* and *arcs* model *causal dependencies*. As a working example we use the Petri net shown in Figure 1.

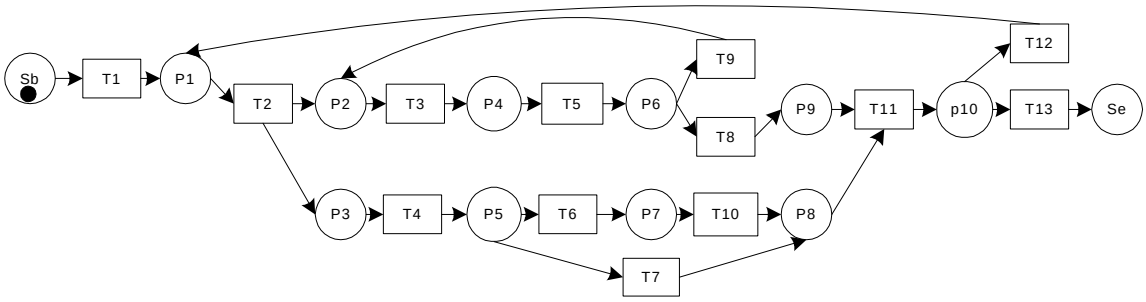

Figure 1: Example of a workflow process modeled as a Petri net.

The transitions *T1, T2, …, T13* represent tasks, The places *Sb, P1, …, P10, Se* represent the causal dependencies. In fact, a place corresponds to a condition that can be used as pre- and/or post-condition for tasks. An AND-split corresponds to a transition with two or more output places (from *T2* to *P2* and *P3*), and an AND-join corresponds to a transition with two or more input places (from *P8* and *P9* to *T11*). OR-splits/OR-joins correspond to places with multiple outgoing/ingoing arcs (from *P5* to *T6* and *T7*, and from *T7* and *T10* to *P8*). At any time, a place contains zero or more *tokens*, drawn as black dots. Transitions are the active components in a Petri net: they change the state of the net according to the following *firing rule*:

(1) A transition *t* is said to be *enabled* if and only if each input place of *t* contains at least one token.

(2) An enabled transition may fire. If transition *t* fires, then *t* consumes one token from each input place *p* of *t* and produces one token for each output place *p* of *t.*

A Petri net that models the control-flow dimension of a workflow is called a *WorkFlow net* (WF-net) [1]. A WF-net has one source place (*Sb*) and one sink place (*Se*) because any case (workflow instance) handled by the procedure represented by the WF-net is created when it enters the workflow management system and is deleted once it is completely handled, i.e., the WF-net specifies the life-cycle of a case. An additional requirement is that there should be no "dangling tasks and/or conditions", i.e., tasks and conditions which do not contribute to the processing of cases. Therefore, all the nodes of the workflow should be on some path from source to sink.

The WF-net focuses on the process perspective and abstracts from the functional, organization, information and operation perspectives [19]. These perspectives can be added using for example high-level Petri nets, i.e., nets extended with color (data) and hierarchy. Although WF-nets are very simple, their expressive power is impressive. In this paper we restrict our self to so-called *sound* WF-nets [1]. A workflow net is sound if the following requirements are satisfied: (i) termination is guaranteed, (ii) upon termination, no dangling references (tokens) are left behind, and (iii) there are no dead tasks, i.e., it should be possible to execute an arbitrary task by following the appropriate route. Soundness is the minimal property any workflow net should satisfy. Note that soundness implies the absence of livelocks and deadlocks. Sound WF-nets can be used to model the basic constructs identified by the Workflow Management Coalition [23] and used in contemporary workflow management systems. Experience with leading workflow management systems such as Staffware, COSA, MQSeries Workflow, etc. show that it is fairly straightforward to express these tool-specific languages in terms of the tool-independent language of WF-nets. For an introduction to WF-nets the reader is referred to [1].

In this paper, we use *workflow logs* to discover workflow models expressed in terms of WF-nets. A workflow log is a sequence of *events*. An event is described by a *case identifier* and a *task identifier*. An event $e=(c,t)$ corresponds to the execution of task t for a given case c. For reasons of simplicity, we assume that there is just one workflow process. Note that this is not a limitation since the case identifiers can be used to split the workflow log into separate workflow logs for each process. One workflow log may contain information about thousands of cases. Since there are no causal dependencies between events corresponding to different cases, we can project the workflow log onto a separate event sequence for each case without loosing any information. Therefore, we can consider a workflow log as a set of event sequences where each event sequence is simply a

sequence of task identifiers. An example of an event sequence of the Petri net of Figure 1 is given below:

T1, T2, T4, T3, T5, T9, T6, T3, T5, T10, T8, T11, T12, T2, T4, T7, T3, T5, T8, T11, T13

Using the definitions for WF-nets and event logs we can easily describe the problem addressed in this paper: Given a workflow log we want to discover a WF-net that (i) potentially generates as many event sequences appearing in the log as possible, (ii) generates as few event sequences not appearing in the log as possible, (iii) captures concurrent behavior, and (iv) is as simple and compact as possible. Moreover, to make our technique practical applicable we want to be able to deal with noise.

4. A Heuristic Process Mining Technique

In this section, we present the details of our *heuristic* process mining technique. In another paper [5], we describe a more *formal* approach and the so-called α-algorithm for which it is proven that it can successfully rediscover a large class of practically relevant WF-nets. The α-algorithm is based on four ordering relation that can be easily derived from a workflow log. Let A and B be events and W a workflow log, then

(1) $A>B$ if and only if there is a trace line in W in which event A is directly followed by B,
(2) $A \rightarrow B$ if and only if $A>B$ and not $B>A$,
(3) $A\#B$ if and only if not $A>B$ and not $B>A$,
(4) $A||B$ if and only if both $A>B$ and $B>A$.

The $A \rightarrow B$ relation is the so-called *dependency relation* (B depends (directly) on A), $A\#B$ relation is the so-called *non-parallel relation* (i.e. there is no direct dependency between them and parallelism is unlikely), and $A||B$ relation is the so-called *parallel relation* (it indicates potential parallelism). In the α-algorithm the dependency relation is used to connect events, the non-parallel relation is used to detect the kinds of splits and joins.

However, the formal approach presupposes perfect information: (i) the log must be complete (i.e. if a task can follow another task directly, the log should contain an example of this behavior) and (ii) we assume that there is no noise in the log (i.e. everything that is registered in the log is correct). In practical situations, logs are rarely complete and/or noise free. Therefore, in practice it becomes more difficult to decide if two events say A and B are in the $A \rightarrow B$ or $A \# B$ relation. For instance the causality relation ($A \rightarrow B$) between two tasks A and B only holds if in the log there is a trace in which A is directly followed by B (i.e. the relation A>B holds) and there is no trace in which B is directly followed by A (i.e. not B>A). However, in a noisy situation one abusive example can completely mess up the derivation of a right conclusion. For this reason we try to developed heuristic mining techniques which are less sensitive for noise and the incompleteness of logs. Moreover, we try to conquer some other limitations of the α-algorithm (short loops and some Petri-net constructs). The remainder of this section is used to present our heuristic mining techniques. In the next section, we describe Little Thumb, a workflow-mining tool based on the rules of thumb presented here.

In our heuristic mining approach we distinguish three mining steps: Step (i) the construction of a dependency/frequency table (D/F-table), Step (ii) the induction of a D/F-graph out of a D/F-table, and Step (iii) the reconstruction of the WF-net out of the D/F-table and the D/F graph.

4.1 Construction of the dependency/frequency table

The starting point of our workflow mining technique is the construction of a D/F-table. For each task *A* the following information is abstracted out of the workflow log: (i) the overall frequency of task *A* (notation *#A*), (ii) the frequency of task *A* directly preceded by another task *B* (notation $\#B<A$), (iii) the frequency of *A* directly followed by another task *B* (notation $\#A>B$), (iv) a local metric that indicates the strength of the dependency relation between task *A* and another task *B* (notation $\$A \rightarrow^{L} B$) and finally (v) a more global metric that indicates the strength of the dependency relation (notation $\$A \rightarrow B$).

Metric (i) through (iii) seems clear without extra explanation. The definition of the local metric (iv) is as follows: $\$A \rightarrow^{L} B = (\#A>B - \#B>A) / (\#A>B + \#B>A + 1)$. Remark that in this definition only local information is used (i.e. the $A>B$ relation). The effect of this definition is that if for instance event A is 5 times directly followed by event B but the other way around never accurse, the value of

$A \overset{L}{\rightarrow} B = 5/6 = 0.833$ indicating that we are not completely sure of the dependency relation (noise can have effected the result). However if A is 50 times followed by B but the other way around never occurs, the value of $A \overset{L}{\rightarrow} B = 50/51 = 0.980$ indication that we are pretty sure of the dependency relation. Even if A is 50 times directly followed by B and noise has effected that B is ones followed by A the value of $A \overset{L}{\rightarrow} B = 49/52 = 0.942$.

The last metric, metric (v), is more global than the other measurements because not only direct following events are involved. The underlying intuition is as follows. If it is always the case that, when task *A* occurs, shortly later task *B* also occurs, then it is plausible that task *A* causes the occurrence of task *B*. On the other hand, if task *B* occurs (shortly) before task *A*, it is implausible that task *A* is the cause of task *B*. Bellow we define the formalization of this intuition. If, in an event stream, task *A* occurs before task *B* and *n* is the number of intermediary events between them, the $A \rightarrow B$-dependency counter is incremented with a factor $(\delta)^n$. δ is a dependency fall factor (δ *in [0.0…1.0]).* In our experiments δ is set to 0.8. The effect is that the contribution to the dependency metric is maximal 1 (if task *B* appears directly after task *A* then *n*=0) and decreases if the distance increases. The process of looking forward from task *A* to the occurrence of task *B* stops after the first next occurrence of task *A* or task *B*. The other way around, if task *B* occurs before task *A* and *n* is again the number of intermediary events between them, the $A \rightarrow B$-dependency counter is decreased with a factor $(\delta)^n$. After processing the whole workflow log the $A \rightarrow B$-dependency counter is divided by the minimum overall frequency of task *A* and *B (min (#A, #B)).*

B	#B	#B<A	#A>B	$A→^{L}B	$A→B
T10	1035	0	581	0.998	0.803
T5	3949	80	168	0.353	0.267
T11	1994	0	0	0	0.193
T13	1000	0	0	0	0.162
T9	1955	50	46	-0.041	0.161
T8	1994	68	31	-0.370	0.119
T3	3949	146	209	0.177	0.019
T6	1035	0	0	0	0.000
T7	959	0	0	0	-0.011
T12	994	0	0	0	-0.093
T1	1000	0	0	0	-0.246
T2	1994	0	0	0	-0.487
T4	1994	691	0	-0.999	-0.825

Table 1: D/F-table for event *T6 (i.e., A=T6).*

Given the process model of Figure 1 a workflow log with 1000 event sequences (23573 events) is generated. As an example, Table 1 shows the above-defined metrics for task *T6*. Notice that the task *T6* belongs to one of two concurrent event streams (the AND-split in *T2*). It can be seen from Table 1 that (i) *T6* is never directly preceded by *T10 (#B<A=0),* (ii) *T6* is often directly followed by *T10 (#A>B=581),* and (iii) both dependency measurements from *T6 to T10 are relatively high (0.998 and 0.803)*. In the next section, we will use the D/F-table in combination with a relatively simple heuristic to construct a D/F-graph.

4.2 Induction of dependency/frequency graphs

In the previous section, we observed that the information in the *T6*-D/F-table (Table 1) strongly suggests that task *T10* depends on task *T6* because the dependency measurements between *T6* and *T10* are high, and *T6* is never directly preceded by *T10* and frequently directly followed by *T10*. In earlier approaches, cf. [25,26], we developed heuristic rules in line with this observation. As an illustration, consider the following rule:

IF $((\#A>B > \sigma)$ AND $(\#B<A \leq \sigma)$ AND $(\$A \overset{L}{\rightarrow} B \geq N_1)$ AND $(\$A \rightarrow B \geq N_2))$ THEN $A \rightarrow B$

$A \rightarrow B$ is the dependency relation (B depends (directly) on A) as introduced before. The four conditions demand that specific values of the D/F graph $(\#A>B, \#B<A, \$A \overset{L}{\rightarrow} B, \$A \rightarrow B)$ are higher or lower than a certain threshold value (σ, N_1, N_2). In the perfect situation where we know that we have a workflow log that is totally free of noise, every task-pattern-occurrence is informative and for instance, the value of σ can be set to zero. However, if it is not clear if a workflow log is noise free, we must protect the induction process against inferences based on noise; only task-pattern-occurrences above a threshold frequency are reliable enough for our induction process. The situation for the N_1 and N_2 parameters is less straightforward. Especially the N_2 parameter appears sensitive for parallelism in the workflow. The high number of parameters and the sensitivity of these parameters for noise are clear disadvantages of this kind of heuristic rules. In [21] we have attempted to use machine-learning techniques for threshold tuning.

However, it seems not necessary to formulate a rule that for each pair of events A and B takes the decision if they are in the dependency relation or not. Because we know that each not-first event must have at least one cause event, and each not-last event must have at least one dependent event. Using this trivial information in our heuristic rule we can limit the search for "the best" candidate. This best helps us enormously in finding dependency relations. The above-formulated idea is implemented in the following first version of our heuristic rule. But first the definition of the *dependency score* (DS) between event two events say X and Y (notation $DS(X,Y)$), is given because we will use the DS in the formulation of the heuristic rule.

- Suppose X and Y are events, then the dependency score $DS(X,Y) = (((\$X \overset{L}{\rightarrow} Y)^2 + (\$X \rightarrow Y)^2)/2$.
- First (temporally) version of mining rule 1. Given a task A:
  $A \rightarrow X$ if and only if X is the event for which DS(A,X) is maximal,
  $Y \rightarrow A$ if and only if Y is the event for which DS(Y,A) is maximal.

Remark that the new heuristic rule does not contain any parameter. Applying this simple heuristic on the D/F-table (Table 1) results in the D/F-graph of Figure 2. If we compare the D/F-graph of Figure 2 with the Petri net of Figure 1 (the Petri-net used for generating the workflow log and D/F-table in Table 1), it can be seen that all the connections between the nodes are in accordance with underlying workflow model (all connections are correct and there are no missing connections). For

each arc the dependency score (DS) is given and for each task the number of event occurrences in the log.

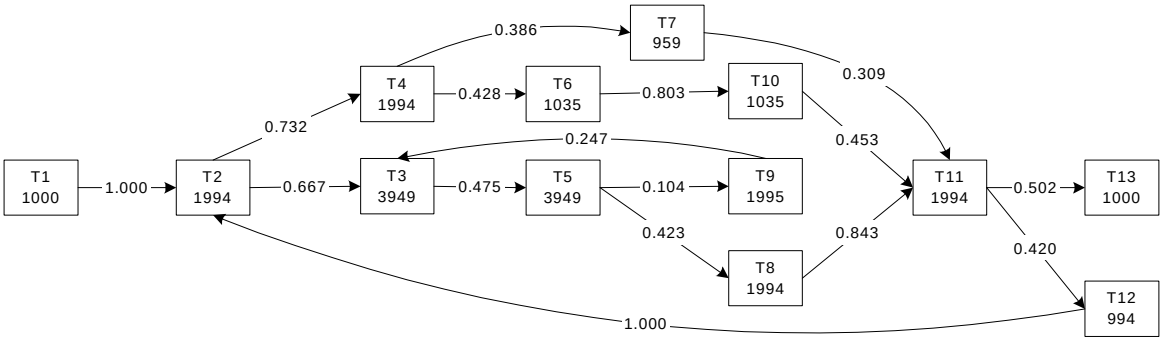

Figure 2: The extracted D/F-graph based on heuristic rule 1.

However, the heuristic rule formulated above will not recognize all possible dependency relations. Figure 3 gives an overview of the types of relations that will not be recognized: (i) some complex connection structures, (ii) recursion, and (iii) short loops. Below we try to update the will discuss them.

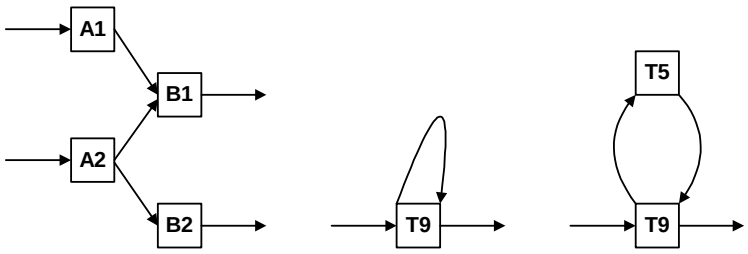

Figure 3: An overview of the types of dependency relations not recognized by the first version of mining rule 1: (i) complex interconnected structures, (ii) recursion, and (iii) short loops.

Applying the first version of mining rule 1 on the first type of structure of Figure 3 can result in missing dependency relations. For instance if the result of applying the rule on *A1* gives *A1→B1*, on

A2 gives *A2→B2*, on *B1* gives again *A1→B1*, and finally on *B2* gives again *A2→B2*, then the dependency relation *A2→B1* will not be recognized. In line with these observations, the first version of our mining rule 1 is updated.

- Mining rule 1 (definite version). Given a task *A*
  Suppose *X* is the event for which *DS(A,X) = M* is maximal. Then A→Y if and only if *DS(A,Y)< 0.95*M*.
  Suppose *X* is the event for which *DS(X,A) = M* is maximal. Then Y→A if and only if *DS(Y,A)< 0.95*M*.

Remark that again we have introduced a threshold value of 0.95. However, it is only one parameter, and the parameter seems robust for noise and concurrent processes. For this reason parameter tuning appears not necessary; the default value of 0.95 is appropriate.

The remaining two type of relations of Figure 3 not covered by rule (1) are recursion (ii) and short loops (iii). Recursion in for instance event *T9* will result in patterns like *T5, T4, T9, T9, T6, T9, T8*. Recursion can be recognized by observing a high frequency of *#T9>T9* in combination with a DS(T9,T9) value of about zero (normally DS(A,A) is only near to zero if #A>A is also near to zero).

A short loop from for instance *T9* to *T5* will result in patterns like *T5, T4, T9, T5, T9, T6, T5, T8*. At first sight short loops can be recognized by observing that both a high and exactly equal frequency of *T5>T9* and *T9<T5* in combination with dependency measurements *DS(T5,T9)* and *DS(T9,T5)* both near to zero. However, the same behavior can be observed when *T5* and *T9* both depend on event *X* with *X* is an AND-split.

In line with these observations, heuristic rule 1 is extended with two simple heuristic rules for recursion (rule 2) and for short loops (rule 3). Details are omitted.

4.3 Generating WF-nets from D/F-graphs

By applying the heuristic rules of subsection 4.1 on a workflow log it appears relatively simple to find the corresponding D/F-graphs. But the types of the splits and joins are not yet represented in the D/F-graph. However (i) information in the D/F-table, and (ii) the frequency of the nodes in the D/F-graph contain useful information to indicate the type of a join or a split.

For instance, if we have to detect the type of a split from *A* to *B* AND/OR *C*, we can look in the D/F-table to the values of $B>C$ and $B<C$. If *A* is an AND-split, then pattern *B, C* and the pattern *C, B* can both appear, and we expect a positive value for both $B>C$ and $C>B$ (or in other words the $B||C$-relation holds). If it is an OR-split, the patterns *B,C* and *C,B* will not appear.

As an example of (ii) we look to the frequency of the nodes in the D/F-graph of Figure 2. *T2* is an AND-split (because *#T4 = #T2* and there are no other incoming arcs for *T4*. *T5* is an OR-split (because *#T5 = #T8 + #T9*), and analogue *T4* and *T11*. The join in node *T11* is a little bit more complex: it appears a combination of a OR-join between *T7* and *T10*, combined with a AND-join with *T8 (#T7+#T10=#T8=#T11).*

In our mining tool the second observation (a frequency check) is used for the validation of the induced workflow model (see Paragraph 5), a heuristic based on the first observation is used to determine the kinds of splits and joins. Remark that for two events A and B exactly one of the following possible relations holds: $A \rightarrow B$, $B \rightarrow A$, $A \# B$, or $A||B$. Based on the D/F-graph we already know if $A \rightarrow B$ or $B \rightarrow A$ holds. If that is not the case we only have to take the decision if $A \# B$, or $A||B$ holds. If this decision is taken, we can apply the α-algorithm of our formal approach [5] to translate this information into a WF-net. Using the α-algorithm in this way we were able to reconstruct the types of the splits and joins appearing in our working example and to reconstruct the complete underlying WF-net (exactly the same one as the WF-net of Figure 1). In the next section, (Section 5) we introduce the workflow mining tool Little Thumb. The tool is based on the workflow mining heuristics as presented in this section. In Section 6 we will report our experimental results of applying the above-defined workflow mining heuristics to other workflow logs, with and without noise.

5. Little Thumb

Little Thumb[1] is a tool that attempts to induce a workflow model from a workflow log. The workflow log may contain errors (noise) and can be incomplete. In Figure 4 a screenshot of Little

[1] For two reasons we chose the name Little Thumb for our process mining tool: (i) it is based on heuristic rules also know as “rules of thumb”, and (ii) the analogy with the fairy tale. In the fairy tale Little Thumb and his brothers are left in the forest by their parents. Fortunately, Little Thumb left a trail of white pebbles to find his way back. The second time, Little Thumb uses breadcrumbs instead of white pebbles to mark the way back. Unfortunately, the breadcrumbs are partly eaten away by the birds, thus making it impossible to find the way back. The goal of the tool Little

Thumb is given. In this example session, Little Thumb is used to analyze a workflow log. For this example we use a log with information about the processing of complaints. However, 5% of the cases in the workflow log contain errors (i.e. part of the event registration is missing or two events are interchanged). In the left upper corner, we see the D/F-table as discussed in Section 4.2.

In the right side of the screenshot we see 5 tabs; (i) Generate WF-log, (ii) Select events, (iii) Load WF-log, (iv) Analyse WF-log, and finally (v) Check WF-net. With the Generate-WF-log tab (i) it is possible to load a WF-net and to generate workflow logs, with and without noise. In an experimental setting, these workflow-logs can be used as mining material to test the mining performance of our tool. With the Select-events tab (ii) we can concentrate our mining process on the most frequent events and neglect low frequent events. The function of Load-WF-log tab (iii) is trivial. The Analyze-WF-log function (tab (iv)) is of more importance. Below we will first illustrate the Analyse-WF-log-function followed by a short illustration of the Check-WF-net function (tab (v)).

Thumb is to deal with situations where the information in the log is incomplete and partly incorrect. This is analogous to finding the way back home on the basis of few breadcrumbs being left by the birds.

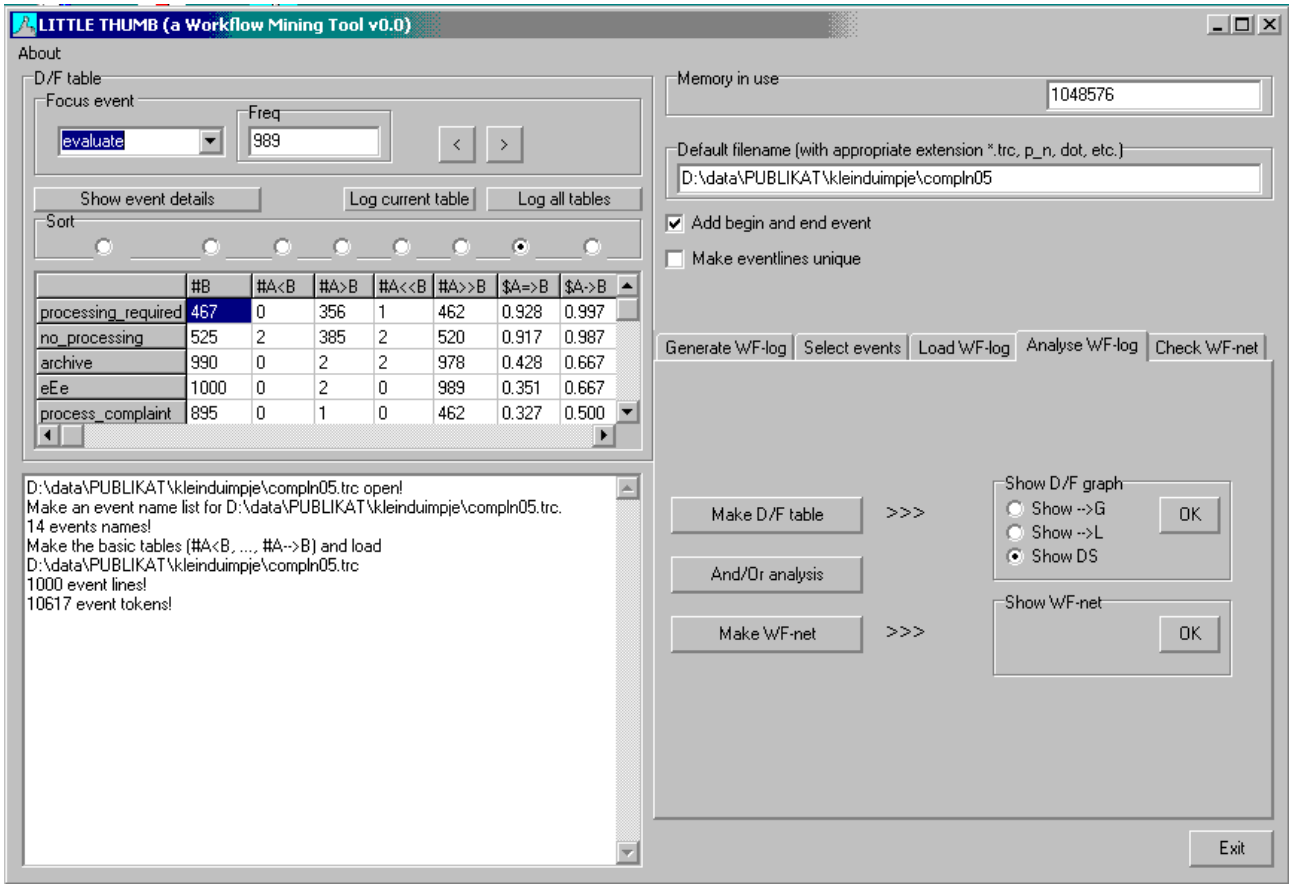

Figure 3: A screenshot of Little Thumb.

We try to follow Little Thumb during his attempt to analyze the loaded workflow log. The same steps as mentioned in Paragraph 4 can be distinguished. Step (i) the construction of a dependency/frequency table (D/F-table). Step (ii) the induction of a D/F-graph out of a D/F-table. Step (iii) the induction of the AND/OR information for the different splits and joins and the reconstruction of the WF-net out of the D/F-table and the D/F graph.

The first step is already executed (see the D/F-table in the screenshot of Figure 4). The ‘evaluate’ event (frequency 989) is in focus. The ‘evaluate’-event is 356 times followed by the ‘processing_required’-event, and 385 times by the ‘no_processing’-event. It easily to see that the workflow log contains noise: the 2 value for the #A<B-counter for the ‘no-processing’-event is caused by an error in the log (i.e. an unusual order). The result of step (ii) the induction of a D/F-graph out of a D/F-table is will results in extra information in the D/F-table: the extra information indicates if events are in the ->, #, or || relation. With the Show-D/F-graph option, it is possible to

display the D/F-graph as illustrated in Figure 5. For each connection the dependency score (DS) is given and for each event the number of occurrences in the log. It is also possible to show the $A \rightarrow^{L} B$-value or the $A \rightarrow B$-value for each connection. The D/F-graph can be used for a first validation with a domain expert and possible changes can be easily implemented by changing the values in the extended D/F-table.

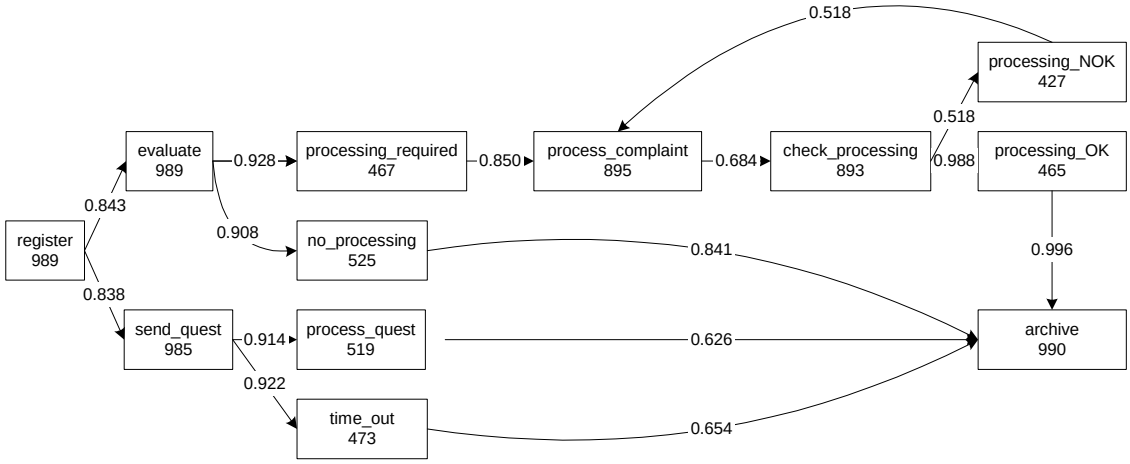

Figure 5: The resulting D/F-graph of Little Thumb.

The types of the splits and joins are not yet represented in the D/F-graph. In Step (iii) we use the information in the extended D/F-table to indicate the type of a joins and splits. Little Thumb will use this information for the induction of the complete WF-net (represented as a Petri-net). In the WF-net we can see that the split from ‘register’ to ‘send_quest’ and ‘evaluate’ is an AND-split. Again it is possible to evaluate the mined WF-net with a domain expert how knows the process that we try to model very well; possible changes can easily be implemented by editing the values in the extended D/F-table.

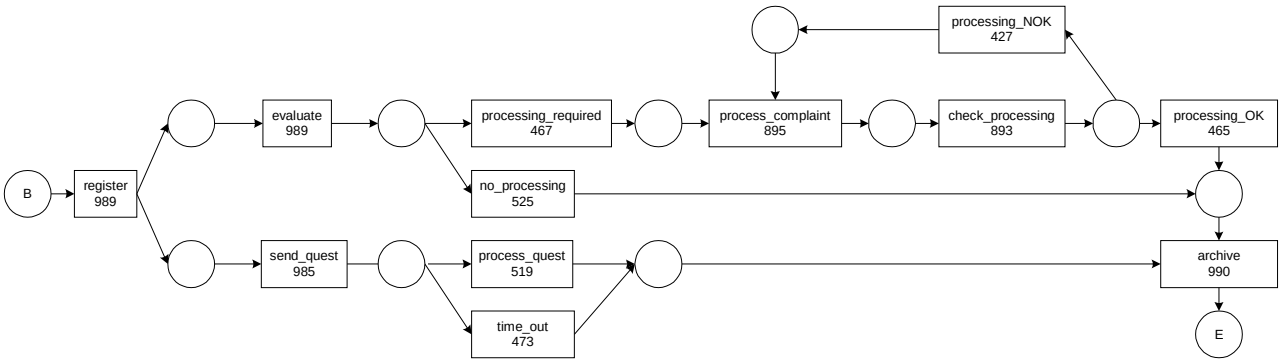

Figure 6: The WF-net as induced by Little Thumb.

Finally we will shortly discuss the last tab (v); the so-called Check WF-net tab. This tab gives us the possibility to validate the WF-net. During the first check, we present all the traces in the workflow log of the WF-net; the WF-net checks if the trace can be parsed by the WF-net. If not, the position in the trace were the parsing stops is shown to the user. Traces in the workflow log with noise will cause problems during tracing. However, if a high number of traces run in problems around the same event, this is a indication that there is a problem in the WF-net around that event. In the second check, we test out if the frequency information of the events (#A) is in accordance with the structure of the mined WF-net. For instance, the frequency information around 'evaluate' (989), 'processing_required' (467) and 'no_processing' (525) points out a OR-split. Small differences can be explained by noise, clear differences indicate an error in the mined WF-net. This concludes our introduction to Little Thumb.

6. Experiments

6.1 First Experiments

To test our approach we use the Petri-net representations of six different free-choice workflow models. The complexity of these models range from comparable with the complexity of our working model of Figure 1 (13 tasks) to models with 16 tasks. All models contain concurrent processes and loops. For each model we generated three random workflow logs with 1000 event sequences: (i) a workflow log without noise, (ii) one with 5% noise, and (iii) a log with 10% noise.

To incorporate noise in our workflow logs we define four different types of noise generating operations: (i) delete the head of a event sequence, (ii) delete the tail of a sequence, (iii) delete a part of the body, and finally (iv) interchange two random chosen events. During the deletion-operations at least one event, and no more than one third of the sequence is deleted. The first step in generating a workflow log with 5% noise is to randomly generate a workflow log for the workflow model without errors. The next step is the random selection of 5% of the original event sequences and applying one of the four above described noise generating operations on it (each noise generation operation with an equal probability of 1/4). Table 2 is the D/F-table for event T6 of the WF-net of Figure 1 but now with some noise added. Comparing this table with Table 1 shows small differences between the values of the tables.

B	**#B**	**#B<A**	**#A>B**	**$A→$^L B**	**$A→B**
T10	1004	2	556	0.991	0.790
T5	3817	77	162	0.354	0.267
T11	1901	0	2	0.667	0.182
T13	923	0	0	0.000	0.161
T9	1902	50	46	-0.041	0.161
T8	1908	66	26	-0.430	0.108
T3	3814	141	203	0.180	0.030
T6	1007	0	0	0.000	0.000
T7	920	0	0	0.000	-0.011
T12	972	0	0	0.000	-0.098
T1	926	3	2	-0.167	-0.254
T2	1904	0	1	0.500	-0.473
T4	1921	664	4	-0.987	-0.808

Table 2: D/F-table for event *T6 (i.e., A=T6)* but now from a workflow log with noise.

Applying the above method on the six noise free workflow logs results in six perfect D/F-graphs (i.e. all the connections are correct and there are no missing connections), and exact copies of the underlying WF-nets. If we add 5% or 10% noise to the workflow logs, the resulting D/F-graphs and WF-nets are still perfect.

6.2 Second Experiments

In a second series of experiments, we try to use workflow logs that resemble real workflows. We observe that at least four elements strongly influence the behavior of a WF-net and/or the workflow mining process:

- *The number of event types* (i.e. tasks) in the WF-net: four different workflow models are used with 12, 22, 32 and 42 event types.
- *The amount of material* in the workflow log: we generated logs with 100, 200, 600, 1000, 1400, and 2000 trace lines.
- *The amount of noise*: we generated workflow log without noise, with 5% noise, 10% noise, 20% noise, and 50% noise.
- *The unbalance* refers to the (relative) probability that enabled event will fire. We consider four different settings: all events have a equal probability to fire, the probabilities vary a little bit (i.e. between [0.9...1.1]), a little bit more (i.e. between [0.5...1.5]), and the probability vary strongly (i.e. between [0.1...1.9]). If for instance task X is an OR-split to A and B and A has a probability or weight of 0.2 and B as weight 1.8, about in 10% of the cases A is taken and in 90% of the cases B is taken. However, if task X is an AND-split to A and B, then both A and B will be executed but in 10% of the cases A precedes B and in 90% of the cases B precedes A. This kind of unbalance can influence the behavior of WF-net and the material in the workflow log dramatically.

We generated 480 different workflow logs by varying each of the above enumerated elements (i.e. 4 x 6 x 5 x 4 = 480). Based on the experiments it was possible to conclude under all circumstances most dependency relations, the type of splits and the type of joins are correctly found. The mining technique appears especially robust for the number of trace lines and the amount of unbalance. Only

if the amount of noise is increased to 50%, the mining technique runs into serious problems. A lower unbalance or the maximum number of trace lines (2000) did not help.

In about a quarter of the cases, exactly the right WF-net was found. In about 75% of the experiments, the check facility of Little Thumb found one or two errors in combination with an indication where in the WF-net the error seems to appear. We observed that most errors have to do with complex interconnected structures (Figure 3) in combination with short loops. An improvement of the heuristic rules for short loops seems necessary.

7. Conclusion

In this paper, we (i) introduced the context of workflow processes and process mining, (ii) some preliminaries including a modeling language for workflow processes, and (iii) a definition of a workflow log. Hereafter, we presented the details of the three steps of our process mining technique: Step (i) the construction of the D/F-table, Step (ii) the induction of a D/F-graph out of a D/F-table, and Step (iii) the reconstruction of the WF-net out of the D/F-table and the D/F graph.

After this, we introduced Little Thumb, a workflow mining tool based on our process mining techniques.

In Section 6, we describe two series of experiments. In the first experiment, we applied our technique on six different workflow models with about 15 tasks. All models contain concurrent processes and loops. For each workflow model, we generated three random workflow logs with 1000 event sequences: (i) without noise, (ii) with 5% noise, and (iii) with 10% noise. Using the proposed technique, we were able to reconstruct the correct D/F-graphs and WF-nets. The experimental results on the workflow logs with noise indicate that our technique seems robust in case of noise.

In the second experiment, not only the amount of noise was varied, but also the amount of material in the log, the complexity of the WF-net (with 12, 22, 32 and 42 event types), and amount of unbalance in the WF-net. Again, the experimental results indicate that our technique is robust against these factors. However, there are still problems with our mining techniques; errors are made around complex interconnected structures in combination with short loops. An improvement of the heuristic rules for short loops seems necessary.

Notwithstanding the reported results and improvements, there is a lot of future work to do. More experimental work must be done especially on real workflow logs. In [3] we present a common

XML-format for workflow logs. Experience shows that it is fairly simple to extract information out of enterprise-specific information systems and translate this to XML format. Little Thumb can read the XML format. However, Little Thumb is only an experimental tool; if our mining technique becomes stable, we can put more effort in developing really interactive, and user friendly mining tool.

Finally, we will extend our mining technique in order to enlarge the set of underlying WF-nets that can be successfully mined.

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, J. Desel, and A. Oberweis, editors. Business Process Management: Models, Techniques, and Empirical Studies, volume 1806 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2000.

3 W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm, A.J.M.M. Weijters. Workflow Mining: A Survey of Issues and Approaches (working paper). http://www.tm.tue.nl/it/staff/wvdaalst/workflow/mining/.

4 W.M.P. van der Aalst and S. Jablonski, editors. Flexible Workflow Technology Driving the Networked Economy, Special Issue of the International Journal of Computer Systems, Science, and Engineering, volume 15, number 5, 2000.

5 W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow mining: which processes can be rediscovered, Working Paper WP74, Beta Research School for Operations Management and Logistic, Eindhoven Technical University, pp 1-25, 2002.

6 R. Agrawal, D. Gunopulos, and F. Leymann. Mining process models from workflow logs. In the proceedings of the Sixth International Conference on Extending Database Technology, pages 469-483, 1998.

7 F. Casati, C. Ceri, B. Pernici, and G. Pozzi. Workflow Evolution. Data and Knowledge Engineering, 24(3): 211-238, 1998.

8 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.

9 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), Orlando, FL, November 1998, pages 35-45.

10 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.

11 J. Desel and J. Esparza. Free Choice Petri Nets, volume 40 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, 1995.

12 C.A. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In N. Comstock and C.A. Ellis, editors, Conf. on Organizational Computing Systems, pages 10 - 21. ACM SIGOIS, ACM. Milpitas, California, 1995.

13 C.A. Ellis and G.J. Nutt. Modelling and Enactment of Workflow Systems. In M. Ajmone Marsan, editor, Application and Theory of Petri Nets 1993, volume 691 of Lecture Notes in Computer Science, pages 1-16. Springer, Berlin, Germany, 1993.

14 J. Herbst. A Machine Learning Approach to Workflow Management. In 11th European Conference on Machine Learning, volume 1810 of Lecture Notes in Computer Science, pages 183-194, Springer, Berlin, Germany, 2000.

15 J. Herbst. Dealing with Concurrency in Workflow Induction In U. Baake, R. Zobel and M. Al-Akaidi, European Concurrent Engineering Conference, SCS Europe, Ghent, Belgium, 2000.

16 J. Herbst and D. Karagiannis. An Inductive Approach to the Acquisition and Adaptation of Workflow Models. In M. Ibrahim and B. Drabble, editors, Proceedings of the IJCAI'99 Workshop on Intelligent Workflow and Process Management: The New Frontier for AI in Business, pages 52-57, 1999.

17 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.

18 InConcert. InConcert Process Designer's Guide. InConcert Inc, Cambridge, Massachusetts, 1998.

19 S. Jablonski and C. Bussler. Workflow Management: Modeling Concepts, Architecture, and Implementation. International Thomson Computer Press, 1996.

20 M. Klein, C. Dellarocas, and A. Bernstein, editors. Adaptive Workflow Systems, special issue of the journal of Computer Supported Cooperative Work, volume 9, numbers 3-4, 2000.

21 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. 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.

22 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.

23 P. Lawrence (editor). Workflow Handbook 1997, Workflow Management Coalition. John Wiley and Sons, New York, 1997.

24 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, Springer-Verlag, Berlin, 2002.

25 A.J.M.M. Weijters and W.M.P. van der Aalst. Process Mining: Discovering Workflow Models from Event-Based Data. In B. Kröse, M. de Rijke, G. Schreiber, and M. van Someren, editors, Proceedings of the 13th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2001), pages 283-290, 2001.

26 A.J.M.M. Weijters and W.M.P. van der Aalst. Rediscovering Work ow Models from Event-Based Data. In V. Hoste and G. de Pauw, editors, Proceedings of the 11th Dutch-Belgian Conference on Machine Learning (Benelearn 2001), pages 93-100, 2001.