Mining most specific Workflow Models from Event-based Data

Guido Schimm

OFFIS, Escherweg 2,
26121 Oldenburg, Germany
schimm@offis.de

Abstract. This paper presents an approach on mining most specific workflow models from event-based data. The approach is embedded in the context of data mining and knowledge discovery in databases. It consists of two parts. The first one is an introduction of a block-structured workflow model representation and the second one is an extraction procedure for workflow models based on that model representation. This paper describes both parts in detail and also outlines preceding and subsequent steps.

1 Introduction

Today, workflow management systems are applied in many organizations. Setting up and maintaining a workflow management system require workflow models which prescribe how business processes should be managed by the system. Typically, the user is required to provide these models. Constructing process models from scratch is a difficult and error-prone task that often requires the use of an expert. An alternative way to construct workflow models from scratch is to extract them from event-based data captured in form of logs from running business processes. The goal of workflow mining is to extract workflow models for business processes from such logs.

In this paper we present an approach on mining most specific workflow models. A model is considered to be most specific according a given log if it is complete and minimal. Completeness describes that a model should preserve all the dependencies between activities that are present in the log. Minimality assures that the model should neither introduce paths of execution nor spurious dependencies between activities which are not present in the log.

Let us assume that the execution of process instances from which we have captured a log is controlled by knowledge that only the actors have in mind. This process-related knowledge is called the implicit model. In case that a log does not cover all possible ways of executing a process, the mined workflow model may be different from the implicit model due to the fact that it contains dependencies between activities which are not part of the implicit model. The obvious limitation of mining workflow models is that the quality of a mined model, with respect to its implicit model, depends on how much the log covers

the implicit model. Also, its characteristic to be most specific can only be related to the log, not to the implicit model itself.

The rest of the paper is organized as follows. The following section outlines the context in which our approach is embedded. In section 3 we define the input. A workflow model representation is outlined in section 4. In section 5 we describe our mining process in detail, followed by a short description of engineering the output in section 6. An experimental evaluation is outlined in section 7. Subsequently, we discuss related work and conclude the paper with a summary.

2 Mining Framework

We consider the mining of workflow models from event based data to be a process of *knowledge discovery in databases* (KDD)[1]. A KDD process roughly consists of the following steps: data consolidation, selection and preprocessing, data mining, interpretation and evaluation. In this paper, we focus on the data mining step and its interfaces to the preceding and subsequent steps.

Generally, data mining consists of two tasks. The first task is to define a model representation. Based on this the second task is the extraction of models from data using appropriate algorithms. Workflow mining can be considered a data mining method that uses a particular workflow meta-model as model representation and specialized algorithms for extracting workflow models from logs. The interfaces of the workflow mining step to its preceding and subsequent steps are given by a definition of its input and a transformation of its output.

3 Defining the Input

Monitoring process instances provides a large amount of data of different events that occur while an instance is being executed. Because we are interested in extracting models that describe control flows between activities within processes we consider events which are related to the life cycle of an activity.

Let us use the finite state machine depicted in Figure 1 to describe the life cycle of an activity. In this paper, we focus on two kinds of events: *Start* and *Complete*. An event of kind *Start* marks the transition from the state *Scheduled* into the the state *Active*. An event of kind *Complete* marks the transition from the state *Running* into the state *Completed*.

For each activity captured in form of a pair of *Start* and *Complete* events we distinguish its type, occurrence, and instance. Two activities of the same type, i.e. the same kind of activity, are differentiated into multiple occurrences if they are embedded in different contexts. The context of an activity is defined by the set of other activities for which a precedence relation to that activity exists. Often, it is enforced that each activity within the same process is named differently. In this case we capture exactly one occurrence per activity. We capture an activity occurrence that is executed at least ones, i.e. we get one activity instance. We

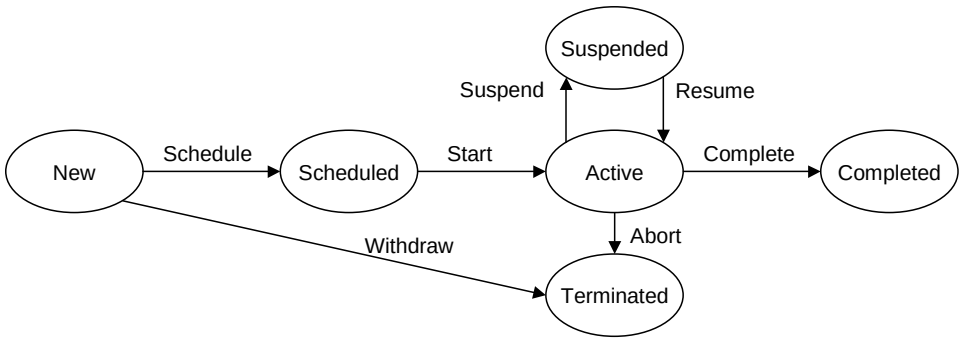

Fig. 1. Finite state machine for the life cycle of an activity

read multiple instances of an activity occurrence if it is performed repeatedly, e.g. inside a loop.

We log an instance of a business process executed under the control of a monitoring tool in form of a trace. A trace consists of a set of events of the life cycle of each activity performed inside a particular process instance. It is defined as follows.

Definition 1. *Let* $e = (c, o, i, s, t)$ *denote an event of starting or completing the instance* $i (i \in \mathbf{N})$ *of an activity occurrence* o *at the logical point in time* $t (t \geq 1)$ *as part of executing a process instance* c*, where* $s \in \{Start, Complete\}$ *is called stereotype of* e*. A trace* $E(c) = (\{e = (c', o, i, s, t) \mid c' = c\}, <_t)$ *is the set of all events of an instance* c *ordered by time, where every event has a unique* t*.*

Mining a model for a business process is based on a set of traces, called the log of the business process. A log is defined as follows.

Definition 2. *Let* $C(p) = \{c_1, \ldots, c_n\}$ *be a set of instances for a particular business process* p*. A log* $L(p) = \{E(c) \mid c \in C(p)\}$ *is a set of traces captured for* p*.*

For a particular business process we select the traces into a log. Before we can start mining we check the log in order to detect inconsistencies. We expect each activity instance in each trace to have exactly one event of stereotype *Start* and one event of stereotype *Complete*, such that each *Start* event comes always before the corresponding *Complete* event. Inconsistent traces are eliminated.

4 Model Representation

The model representation, i.e. the meta-model the extracted workflow models are based on, is a block-oriented model. It defines that each workflow model consists of an arbitrary number of nested building blocks. Building blocks are differentiated into operators and terms. Operators combine building blocks and

define the control flow of a workflow model. Basic terms are references for activity occurrences or sub-workflows that are embedded into a control flow. We use references to activity occurrences instead of activity occurrences because this allows us to have partial synchronization for an activity occurrence.

We build a block-structured model top-down by setting one operator as starting point of the workflow model and nest other operators until we get the desired control flow structure. At the bottom of this structure we embed basic terms into operators and terminate the nesting process. Therefore, each process model can be represented in form of a tree whose leafs are always operators. Beside the tree representation of block-structured models we can write them in form of terms. Furthermore, block-structured models can be represented in form of diagrams, too. Block-structured models have some advantages: They are well-formed and always sound. Therefore, using this kind of meta-model we are sure that the extracted workflow models do not contain any deadlocks or other anomalies.

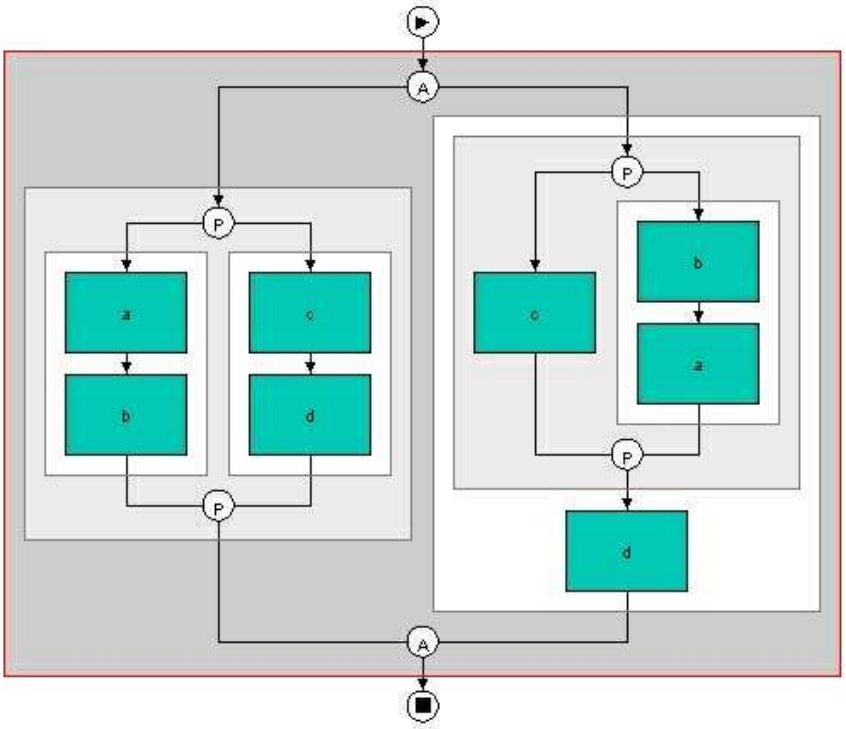

Fig. 2. A simple process model

In this paper we use a basic block-oriented meta-model that consists of activity occurrence references as basic terms and n-ary operators for sequences, parallels, and alternatives. The *Sequence* operator defines that all embedded

blocks are performed in a sequential order. Let $\mathcal{S}$ denote the *Sequence* operator and let a and b denote two activity occurrence references, then $\mathcal{S}(a, b)$ is a workflow model defining that activity a is performed before activity b. In contrast, the *Parallel* operator defines that all embedded blocks can be executed in parallel, i.e. that no precedence relation exists between the embedded blocks. Let $\mathcal{P}$ denote the *Parallel* operator, and let a and b denote two activity occurrence references, then $\mathcal{P}(a, b)$ is a workflow model defining that activity a and activity b can be performed independently, i.e. in any order or in parallel. The *Alternative* operator defines a choice of exactly one block out of all its embedded blocks. This operator is supplemented by a set of rules determining the choice. Let $\mathcal{A}$ denote the *Alternative* operator, and let a and b denote two activity occurrence references, then $\mathcal{A}(a, b)$ is a workflow model defining that either activity a or activity b is executed. Additionally, we define a *Loop* operator $\mathcal{L}$. The *Loop* operator contains only one block that is executed repeatedly until its loop condition holds.

For example, Figure 2 shows the process model $\mathcal{A}(\mathcal{P}(\mathcal{S}(a, b), (c, d)), \mathcal{S}(\mathcal{P}(c, \mathcal{S}(b, a)), d))$ in form of a diagram. The root of this model is a alternative operator that contains all other blocks in a hierarchical manner, such that all activity occurrence references are embedded in a nested control flow.

For our meta-model we define an algebra that consists of a set of axioms covering distributivity, associativity, commutativity, and idempotency. These axioms are the basis of term rewriting systems that can be used in order to transform workflow models. A set of basic axioms used by our workflow mining approach is shown in Table 1. Associativity is defined for all operators (A2, A5, A8). Distributivity is defined in form of left distributivity of the *Sequence* operator over the *Alternative* operator (A4), and in form of full distributivity of the *Parallel* operator over the *Alternative* operator (A7). Note that this algebra does not contain an axiom for the right or full distributivity of the *Sequence* operator over the *Alternative* operator because this would lead to different models depending on the time a decision is made. Commutativity of embedded blocks is defined for *Alternative* and *Parallel* operators (A1, A6). Idempotency is only defined for the *Alternative* operator (A3).

A1: $\mathcal{A}_1(x_1, \dots, x_n) = \dots = \mathcal{A}_{n!}(x_n, \dots, x_1)$
A2: $\mathcal{A}(\dots, x_1, \dots, x_n, \dots) = \mathcal{A}(\dots, A(x_1, \dots, x_n), \dots)$
A3: $\mathcal{A}(x, x, \dots) = x$
A4: $\mathcal{S}(\mathcal{A}(x_1, \dots, x_n), y_1, \dots, y_m) =$
$\quad \mathcal{A}(\mathcal{S}_1(x_1, y_1, \dots, y_m), \dots, \mathcal{S}_n(x_n, y_1, \dots, y_n))$
A5: $\mathcal{S}(\dots, x_1, \dots, x_n, \dots) = \mathcal{S}(\dots, \mathcal{S}(x_1, \dots, x_n), \dots)$
A6: $\mathcal{P}_1(x_1, \dots, x_n) = \dots = \mathcal{P}_{n!}(x_n, \dots, x_1)$
A7: $\mathcal{P}(\mathcal{A}(x_1, \dots, x_n), y_1, \dots, y_m) =$
$\quad \mathcal{A}(\mathcal{P}_1(x_1, y_1, \dots, y_m), \dots, \mathcal{P}_n(x_n, y_1, \dots, y_n))$
A8: $\mathcal{P}(\dots, x_1, \dots, x_n, \dots) = \mathcal{P}(\dots, P(x_1, \dots, x_n), \dots)$

Table 1. Basic algebra

The basic meta-model can be supplemented with further operators and basic terms. For example, one can define a *Parallel* operator that controls the order of starting its embedded blocks as well as a non-exclusive *Alternative* operator. Also, an basic term that represents sub-processes may be defined. In this paper we use only the basic meta-model outlined above.

5 Mining Workflow Models

In the previous sections we have defined the input of the mining process and a model representation. Now, we can describe how to mine most specific workflow models from input. Our mining process consists of five steps which are performed on a log $L(p)$ for a process p in sequential order. Following, each step is described in the order of application.

5.1 Labeling Multiple Activity Instances

A trace may contain events for multiple instances of the same activity occurrence. These instances result from executing an activity occurrence repeatedly within a process instance. Due to the fact that a workflow model only contains references for activity occurrences, we now substitute each instance of a set of multiple instances by a single instance of an activity occurrence. For this purpose we define the following labeling operator.

Definition 3. *Let $E(c)$ denote a trace, $E(c) \in L(p)$. The operator $\zeta : E \rightarrow E$,*

$$\zeta(e) = \begin{cases} e : & |\{e' \mid e, e' \in E(c) \wedge e'.o = e.o \wedge e'.s = e.s\}| = 1 \\ (c, k, 1, s, t) : & \text{otherwise, where } k = o \circ i \text{ represents a concatination} \end{cases}$$

is called labeling operator.

The labeling operator is applied to each event in each trace of the log $L(p)$. It transforms events, such that each instance of a set of multiple instances of an activity occurrence within a particular $E(c)$ is represented by a single instance of an activity occurrence that is artificially differentiated by the label $k = o \circ i$.

5.2 Grouping Traces into Classes

In this step, we group traces of a log into trace classes. The way of grouping depends on the meta-model. For the basic meta-model the building of trace classes is defined as follows.

Let us consider a trace. Its sequence of events can be splitted into alternating subsequences which only contain events of the same stereotype. Each subsequence is called a cluster of the trace.

Definition 4. *Let* $\gamma(E(c)) = \{E(c)_1^+, E(c)_1^-, E(c)_2^+, \ldots, E(c)_q^-\}$ *denote the clustering of* $E(c)$*, where* $E(c)_1^+$ *is the first set of subsequent events of stereotype* Start*,* $E(c)_1^-$ *is the first set of subsequent events of stereotype* Complete*, and so on.*

The type of an event is a tuple containing the activity occurrence and the stereotype. For the basic meta-model we define the equivalence of traces as follows.

Definition 5. *Let* $\kappa : \{(C, O, \mathbf{N}, S, T)\} \rightarrow \{(O, S)\}$*, with* $\kappa(e) = (e.o, e.s)$*, denote a projection on event* e*, where* o *denotes the activity occurrence of* e *and* s *denotes its stereotype.* $\kappa(e)$ *is called the type of* e*. Two events* e *and* e' *are equivalent,* $e \equiv e'$*, iff they have the same type,* $\kappa(e) = \kappa(e')$*. Two traces* $E(c)$ *and* $E(c')$ *are equivalent,* $E(c) \equiv E(c')$*, iff clusters at the same position contain the same subset of equivalent events,* $\forall E(c)_i^{+/-} \in \gamma(E(c)), \forall E(c')_i^{+/-} \in \gamma(E(c')) : \{\kappa(e) \mid e \in E(c)_i^{+/-}\} = \{\kappa(e') \mid e' \in E(c')_i^{+/-}\} \Leftrightarrow E(c) \equiv E(c')$*.*

Note, that the order of events within a cluster is not relevant. A trace class can now be defined as an ordered set of clusters that contains the event types of its member traces.

Definition 6. *Let* $\rho(p)_h$ *denote a part* $\{E(c_1), \ldots E(c_m) | E(c_1) \equiv \cdots \equiv E(c_m)\}$ *of equivalent traces of a partition of* $L(p)$ *based on the equivalence relation* $E(c) \equiv E(c')$*.* $D_h = \{U_1^+ = \{\kappa(e) \mid e \in \cup E(c_i)_1^+\}, U_1^- = \{\kappa(e) \mid e \in \cup E(c_i)_1^-\}, \ldots, U_q^- = \{\kappa(e) \mid e \in \cup E(c_i)_q^-\} \mid \forall E(c_i) : E(c_i) \in \rho(p)_h, 1 \leq i \leq |\rho(p)|\}$ *is the trace class of* $\rho(p)$*, where* $U_l^{+/-}, 1 \leq l \leq q$*, denote the clusters of event types.*

For a log $L(p)$ we can now determine the partition $D(L(p))$ whose parts are trace classes $(D_h)_{1 \leq h \leq l}$ consisting of equivalent traces. The number of trace classes for $D(L(p))$ depends on the variability of the process instances captured in $L(p)$. It ranges from 1 to the number of traces.

5.3 Extracting Precedence Relations

For each trace class D_h, $D_h \in D(L(p))$, we now extract a precedence relation. It relates two event types for which we detect a dependency.

Algorithm 1. *Let* $R_h = \{(o, o') \mid o \in O \cup \alpha, o' \in O\}$ *denote the precedence relation for* D_h*, where* α *is an initiating activity occurrence that marks the start of a process. Let* $R(L(p)) = (R_h)_{1 \leq h \leq k}$ *denote the family of all precedence relations for* $D(L(p))$*. Let* $\mu : U \rightarrow \{o \mid (o, s) \in U\}$ *be a projection function that returns all activity occurrences of the event types of a cluster* U*. Let* $\varphi : U \rightarrow \{Start, Complete\}$ *denote a function that returns the stereotype of a cluster* U*.* S *is a set. Input:* $D(L(p))$*.*

For each $D_h, D_h \in D(L(p))$ {
  $R_h := \emptyset$
  $S := \alpha$
    For each $U_i^{+/-}, U_i^{+/-} \in D_h, i = 1, \ldots, q$ {
      If $\varphi(U_i^{+/-}) = Start$ {
        $R_h := R_h \cup \{(o, o') | o \in S, o' \in \mu(U_i^+)\}$
      }
      Else { $S := S \cup \mu(U_i^-)$ }
    }
  }
  $R(L(p)) = R(L(p)) \cup R_h$
}
Return $R(L(p))$

Algorithm 1 shows that we use transitive dependencies between activity occurrences, e.g. $\forall o, o', o'' \in \bigcup \mu(U_i), U_i \in D_h : (o, o') \in R_h \wedge (o', o'') \in R_h \Rightarrow (o, o'') \in R_h$. Also note that we extract the precedence relation for each trace class separately.

Example 1. Let $D_1 = \{\{(a, Start), (c, Start)\}, \{(a, Complete)\}, \{(b, Start)\}, \{(c, Complete)\}, \{(d, Start)\}, \{(b, Complete), (d, Complete)\}\}$, $D_2 = \{\{(a, Start), (c, Start)\}, \{(c, Complete)\}, \{(d, Start)\}, \{(a, Complete)\}, \{(b, Start)\}, \{(b, Complete), (d, Complete)\}\}$, and $D_3 = \{\{(c, Start), (b, Start)\}, \{(c, Complete), (b, Complete)\}, \{(a, Start)\}, \{(a, Complete)\}, \{(d, Start)\}, \{(d, Complete)\}\}$ be the trace classes for a log. Figure 3 shows the precedence relations R_i that is extracted from D_i by applying Algorithm 1.

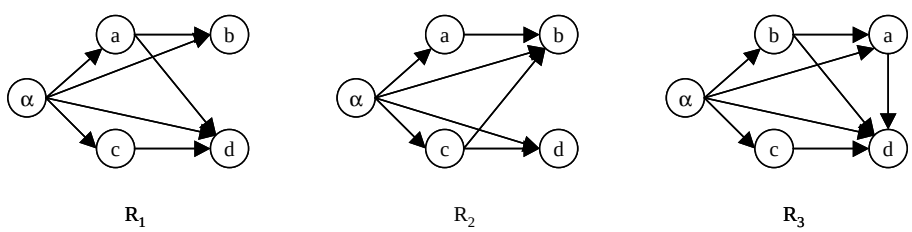

Fig. 3. Graphs of example precedence relations

At this point the relations in $R(L(p))$ may contain pseudo dependencies. Pseudo dependencies are dependencies which actually do not exist in the implicit model. They occur randomly and are caused by random execution times or delays. For example, activities which are embedded in parallel sequences are often performed in a pseudo sequential order due to the fact that they are embedded in opposite ends of their sequences. In order to detect pseudo dependencies in relations we group together all relations which have the same set of activity

occurrences. The main indicator to decide whether a dependency is real or spurious is that a pseudo dependency is a dependency that is not found in every relation of such a group. We use the following algorithm for pseudo dependency detection and elimination.

Algorithm 2. *Let* $\beta : R \rightarrow \{o \mid (o, o') \in R \vee (o', o) \in R\}$ *denote a function which returns the set of activity occurrences of a relation* R*. Let* B_j *denote a part of a partition of* $(R(L(p))$ *that is defined by the equivalence relation* $\beta(R) = \beta(R') \Leftrightarrow R \equiv R'$*. Let* $\phi : R \rightarrow R'$ *denote a function that computes the transitive reduction of a precedence relation.* U, V *are sets of relations;* u, v *are sets. Input:* $R(L(p))$*.*

For each $B_j\{$
  $U := B_j$
  While $|U| > 0$ $\{$
    $v := R_1, V := R_1, R_1 \in U$
    $U := U \setminus R_1$
    For each $R_i, R_i \in U$ $\{$
      $u := R_1 \cap R_i$
      If $\beta(u) = \beta(R_1)$ $\{$
        $V := V \cup R_i$
        $U := U \setminus R_i$
        $v := v \cap R_i$
        $\}$
    $\}$
  $R(L(p)) := R(L(p)) \setminus V$
  $R(L(p)) := R(L(p)) \cup \phi(v)$
  $\}$
$\}$
Return $R(L(p))$

Algorithm 2 returns the adjusted precedence relations $R'(L(p))$ describing each alternative path of executing the process p as it was captured in $L(p)$.

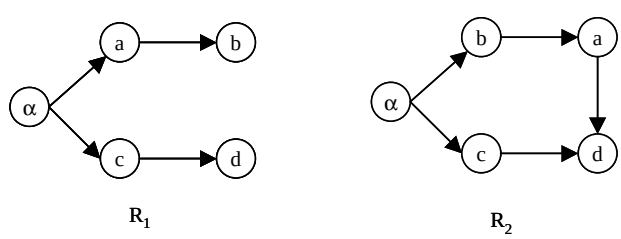

Fig. 4. Graphs of adjusted precedence relations

Example 2. Applying Algorithm 2 on $R(L(p)) = \{R_1, R_2, R_3\}$ from Example 1 results in $R'(L(p)) = \{R_1, R_2\}$ which is depicted in Figure 4. Note that R_1 and R_2 from $R(L(p))$ are merged into R_1 of $R'(L(p))$.

5.4 Model Synthesis

In this step we generate an initial model for p from $R'(L(p))$. First, we generate a sub-model for each path represented by a precedence relation $R_h \in R(L(p))$. Then, a model for the overall workflow is assembled from all sub-models. We use algorithm 3 that handles both task.

Algorithm 3. *Let* $f(R_h) = \{\alpha o_1, o_1 o_2, \ldots, o_{k-1} o_k\}, o_k \in \{o \mid (o, o') \notin R_h \wedge (o', o) \in R_h\}$*, denote a path in* R_h *starting at* α *and ending at an activity occurrence* o_k *that do not have any successor. Let* $F(R) = \{f(R_h)\}$ *denote the set of paths for* $R(L(p))$*. Let* $\eta : f(R_h) \rightarrow \{o \mid ((o, o') \in f(R_h) \vee (o', o) \in f(R_h)) \wedge o \neq \alpha\}$ *denote a function that returns all activity occurrences of a path in ascending order. Let* $\psi : O \rightarrow A$ *be a function that returns an unique activity occurrence reference for an activity occurrence.* sb_n *and* b_m *are variables of type* block*. Input:* $R(L(p))$*.*

For each $R_h, R_h \in R(L(p))\{$
  For each $f_i(R_h), f_i(R_h) \in F(R)$ $\{$
    $sb_i := \mathcal{S}(\psi(o_1), \psi(o_2), \ldots), o_j \in \eta(f_i(R_h)), 1 \leq j \leq |\eta(f_i(R_h))|$
  $\}$
  $b_h := \mathcal{P}(sb_1, \ldots, sb_n)$
$\}$
Return $\mathcal{A}(b_1, \ldots, b_m)$

We apply algorithm 3 on $R(L(p))$. It results an initial model $M(L(p))$ in the form $\mathcal{A}(\mathcal{P}(\mathcal{S}(\ldots), \ldots), \ldots)$.

Example 3. Applying Algorithm 3 on $R'(L(p))$ from Example 2 results in the model $\mathcal{A}(\mathcal{P}(\mathcal{S}(a, b), (c, d)), \mathcal{S}(\mathcal{P}(c, \mathcal{S}(b, a)), d))$ depicted in Figure 2.

5.5 Model Transformation

In this step we transform the initial model into a consolidated and anticipative form. For this purpose we use three different kinds of transformations. First, we detect loops and complete the model with appropriate loop operators. Second, we merge parallel paths in the model by applying a term rewriting system. Third, we change the decision structure of the model in order to make it an anticipatory model.

In section 5.1 we used a labeling operator to artificially differentiate multiple instances of an activity occurrence within a case. Actually, we want to have loop operators containing activity occurrences for those occurrences. At this point we construct loop operators for the labeled occurrences.

Algorithm 4. *Let $\Pi(\mathcal{S})$ denote a minimal partition of the activity occurrence references in a sequence $\mathcal{S}(a, a', \dots)$, so that for each part π the following holds: $\forall a, a', \psi^{-1}(a) = o \circ i, \psi^{-1}(a') = o \circ j : a \in \pi \Rightarrow a' \in \pi \wedge \forall a, a', a'', a''' \in \pi, \psi^{-1}(a) = o \circ i, \psi^{-1}(a') = o \circ j, \psi^{-1}(a'') = o \circ (i+1), \psi^{-1}(a''') = o \circ (j+1) : a < a' < a'' < a'''$. Let $\theta : \pi \rightarrow (o, o', \dots)$ denote a function that returns the sequence of all activity occurrences in the order in which their first occurrence is found in π, and let $\Psi : (o, o', \dots) \rightarrow (\psi(o), \psi(o'), \dots)$ be a function that returns a sequence of activity occurrence references for a sequence of activity occurrences. Let $\xi : A \rightarrow \{0, 1\}$ denote a function that returns 1 iff a is a reference for a labeled activity occurrence, and 0 otherwise. Input: $M(p_j)$.*

$\forall S_i \in M$ {
  $a_\diamond :=$ *first labeled a in S_i*
  $a_\triangle :=$ *last labeled a in S_i*
  Split S into S_p, S_b, S_p where
    $S_p := \mathcal{S}(a, \dots, a'), a' < a_\diamond$
    $S_b := \mathcal{S}(a'', \dots), \forall a : a_\diamond < a < a_\triangle \wedge \xi(a) = 0$
    $S_l := \mathcal{S}(a_\diamond, a''', \dots, a_\triangle), \forall a : \xi(a) = 1$
    $S_s := \mathcal{S}(a'''', \dots), a'''' > a_\triangle$
  Determine $\Pi(S_l)$
    For each $\pi_k, \pi_k \in \Pi(S_l)${
      $b_k = \mathcal{L}(\mathcal{S}_k(\Psi(\theta(\pi_k))))$
    }
  $\mathcal{S} := \mathcal{S}(S_p, \mathcal{P}(S_b, b_1, \dots, b_l), S_s)$
}

After constructing loops we merge sequences and loops, respectively, embedded in parallel operators by applying a term rewriting system (TRS_1) that consists of the following rewritings:

$$\begin{aligned}
&\mathcal{P}(\mathcal{S}(b_1, \dots, b_u, b_{u+1}, \dots, b_v), \dots, \mathcal{S}(b_1, \dots, b_u, b'_{u+1}, \dots, b_w)) \rightarrow \\
&\mathcal{S}(b_1, \dots, b_u, \mathcal{P}(\mathcal{S}(b_{u+1}, \dots, b_v), \dots, \mathcal{S}(b'_{u+1}, \dots, b'_w)) \\
&\mathcal{P}(\mathcal{S}(b'_1, \dots, b'_u, b_{u+1}, \dots, b_v), \dots, \mathcal{S}(b'_1, \dots, b'_u, b_{u+1}, \dots, b_w)) \rightarrow \\
&\mathcal{S}(\mathcal{P}(\mathcal{S}(b_1, \dots, b_u), \dots, \mathcal{S}(b'_1, \dots, b'_u), b_{u+1}, \dots, b_w)
\end{aligned}$$

Note, that b denotes a block, i.e. an operator or a term. Because TRS_1 is not confluent we have to specify the order of application. We always apply the first rewriting before the second one if both rewritings are applicable.

Example 4. Let $b_1, b'_1, \dots, b_n, b'_n$ denote references for activity occurrences $b_1, \dots, b_n$. Given the model: $\mathcal{P}(\mathcal{S}(b_1, b_2, b_3, b_4), \mathcal{S}(b'_1, b'_2, b_5, b_6), \mathcal{S}(b_7, b'_5, b'_6))$ Applying TRS_1 produces: $\mathcal{P}(\mathcal{S}(b_1, b_2, \mathcal{P}(\mathcal{S}(b_3, b_4), \mathcal{S}(b_5, b_6))), \mathcal{S}(b_7, b'_5, b'_6))$

Up to now we have mined models from a retrospective perspective. Actually, we want workflow models in order to prescribe the order of executing activities. Therefore, we make a shift from the retrospective perspective to an anticipatory perspective at this point. We do so by splitting the overall *Alternative* operator

of $M(L(p))$ into partial *Alternative* operators, and moving any of these operators to its very latest possible position in its model. For this purpose only, we use the left distributivity of the *Alternative* operator over the *Sequence* operator.

We use the following term rewriting system TRS_2 based on the left and right distributivity of the *Alternative* operator over the *Sequence* operator and the *Parallel* operator.

$$\mathcal{A}(\mathcal{S}(b_1,\ldots,b_u,b_{u+1},\ldots,b_v),\ldots,\mathcal{S}(b_1,\ldots,b_u,b'_{u+1},\ldots,b_w)) \rightarrow$$
$$\mathcal{S}(b_1,\ldots,b_u,\mathcal{A}(\mathcal{S}(b_{u+1},\ldots,b_v),\ldots,\mathcal{S}(b'_{u+1},\ldots,b'_w))$$
$$\mathcal{A}(\mathcal{S}(b'_1,\ldots,b'_u,b_{u+1},\ldots,b_v),\ldots,\mathcal{S}(b'_1,\ldots,b'_u,b_{u+1},\ldots,b_w)) \rightarrow$$
$$\mathcal{S}(\mathcal{A}(\mathcal{S}(b_1,\ldots,b_u),\ldots,\mathcal{S}(b'_1,\ldots,b'_u),b_{u+1},\ldots,b_w)$$
$$\mathcal{A}(\mathcal{P}(b_1,\ldots,b_u),\ldots,\mathcal{P}(b'_1,\ldots,b'_u),\mathcal{P}(b_v,\ldots,b_w),\ldots,\mathcal{P}(b_x,\ldots,b_y)) \rightarrow$$
$$\mathcal{P}(b_1,\ldots,b_u,\mathcal{A}(\mathcal{P}(b_v,\ldots,b_w),\ldots,\mathcal{P}(b_x,\ldots,b_y)))$$

Because TRS_2 is not confluent we have to specify the order of application of its rewritings. Again, we apply the left distribution before right distribution. We use this order because the left distribution has the desired affect of changing points in time decisions have to be made.

Example 5. Let $b_1, b'_1, \ldots, b_n, b'_n$ denote references for activity occurrences $b_1, \ldots, b_n$. Given the model: $\mathcal{A}(\mathcal{S}(b_1,b_2,b_3,b_4,b_5),\mathcal{S}(b'_1,b'_2,b_6,b_6),\mathcal{P}(b_8,b_9,b_{10}),\mathcal{P}(b'_9,b'_{10},b_{11}))$ Applying TRS_2 produces: $\mathcal{A}(\mathcal{S}(b_1,b_2,\mathcal{A}(\mathcal{S}(b_3,b_4,b_5),\mathcal{S}(b_6,b_7))),\mathcal{P}(b_9,b_{10},\mathcal{A}(b_8,b_{11})))$

At this point we have mined a workflow model from a workflow log.

5.6 Complete and Minimal Models

We expect our mining procedure to extract models wich are complete and minimal. Let us summarize how this is achieved regarding the entire process.

Before we start we eliminate inconsistent traces, i.e. the handling of noise is considered to be completed before we run the mining procedure. Based on this, each trace goes into a trace class. In doing so, we ensure that every behavior presented in the log is taken into account and the resulting model is complete. From each trace class exactly one precedence relation is separately extracted. Then, only such precedence relations are merged which differ in eliminated pseudo dependencies. After that, a separate sub-model is extracted from each precedence relation. The term rewriting operating on this model only permits the merging of blocks which are embedded in equal contexts.

We stress on separate handling because it is key in order to mine minimal models. For example, let us consider two simple precedence relations $R_1 = \{(\alpha,a),(a,c),(c,e)\}$ and $R_2 = \{(\alpha,b),(b,c),(c,e)\}$ as extract from a log. In case that we process $R_1 \cup R_2$ instead of processing both relations separately, this results in a model that introduces the possibility of executing e after a and c were executed. But this execution sequence is never found in the log. With our

approach the model $\mathcal{A}(\mathcal{S}(a,c,e),\mathcal{S}(b,c,d))$ is constructed. Note that there is no term rewriting or any other transformation that adulterates the sequences.

In case that there are non-desired paths in a model these can be eliminated in the subsequent evaluation stage of the KDD process. For example, such paths may base on very few traces and therefore considered to be irrelevant exceptions. Also, such paths can represent valuable process knowledge. However, the decision about that is not subject to a mining procedure. It is subject to a subsequent model evaluation performed by a user.

6 Engineering the Output

The process mining output comes in form of workflow models based on the meta-model described above. At this point we want to transform these models into models applicable in different workflow systems. A common format for exchanging models between different workflow systems is the WfMC *Interface 1: Process Definition Interchange*[2]. This interface includes a common meta-model for describing process definitions and a textual grammar for the interchange of process definitions, called Workflow Process Definition Language (WPDL). Also, there is a XML version of this language, called XPDL. In order to deploy our models to different workflow systems we can transform them into WPDL-models.

There is a structural difference between our models and WPDL-models in the sense that is WPDL does not support activity occurrence references. Therefore, it is necessary to first insert an additional synchronization of multiple references of the same activity occurrence which are embedded in multiple parallel operators. For this purpose, we expand each reference of an activity occurrence for which there are more than one references embedded in a parallel operator, such that each of these parallel operators contains all such references together. After then, we apply the term rewriting system TRS_1 to the expanded model.

Example 6. Let $b_1, b'_1, \ldots, b_n, b'_n$ denote references for activity occurrences $b_1, \ldots,$ b_n. Given the model: $\mathcal{P}(\mathcal{S}(b_1, b_2, \mathcal{P}(b_3, b_4)), \mathcal{S}(b_5, b'_4), \mathcal{S}(b_6, \mathcal{P}(b'_4, b_7)))$ Its expanded form is: $\mathcal{P}(\mathcal{S}(b_1, b_2, \mathcal{P}(b_3, b_4, b_7)), \mathcal{S}(b_5, \mathcal{P}(b'_3, b'_4, b_7), \mathcal{S}(b_6, \mathcal{P}(b'_3, b'_4, b'_7)))$ Applying TRS_1 results in: $\mathcal{S}(\mathcal{P}(\mathcal{S}(b_1, b_2), b_5, b_6), \mathcal{P}(b_3, b_4, b_7))$ Note, that there is now an additional synchronization of b_4 with b_3 and b_7.

After this we can transform a model into WPDL. Algorithm 5 performs a simple transformation.

Algorithm 5. *Let b denote a block of a workflow model. Let $\phi : B \rightarrow T$, where $T = \{\mathcal{S}, \mathcal{P}, \mathcal{A}, \mathcal{L}, \mathcal{O}\}$, denote a function that returns the type of a Block b. Let* createActivity(T) *be a subroutine that creates and writes a* WPDL Route Activity*, let* createLoopActivity(T) *be a subroutine that creates and writes a* WPDL Loop Activity*, and let* writeActivity(B) *denote a subroutine that writes a* WPDL Activity *for an activity instance b, where $\phi(b) = \mathcal{O}$. Let P, P', D denote sets of* WPDL Activities*. Let* writeTransitions(P, P') *denote a subroutine that writes* WPDL Transitions *from all $p \in P$ to all $p' \in P'$, and let* callSelf(B, D) *be a recursive execution of the algorithm. Input: $b := M$, $P := \emptyset$.*

```
If φ(b) = S {
   For each b′ embedded in b ordered by S {
      P := callSelf(b′, P)
      }
   Return P
}
If φ(b) = P ∨ φ(b) = A {
   D := createActivity(φ(b))
   writeTransitions(P, D)
   For each b′ embedded in b {
      P′ := P′ ∪ callSelf(b′, D)
   }
   D := createActivity(φ(b))
   writeTransitions(P′, D)
   Return D
}
If φ(b) = L {
   D := createLoopActivity(L)
   writeTransitions(P, D)
   b′ := the block embedded inside the loop
   P := callSelf(b′, D)
   writeTransitions(P, D)
   Return D
}
If φ(b) = O {
   writeActivity(ϕ(b))
   writeTransitions(P, ϕ(b))
   Return ϕ(b)
}
```

Besides a transformation of models into WPDL/XPDL, one can provide further algorithms for transforming models into proprietary workflow description languages for particular workflow systems, for example, IBMs *Flow Description Language* (FDL) used by *IBM MQSeries Workflow*.

7 Experimental Evaluation

Our approach is implemented by a tool named *Process Miner*. It is able to read traces of a particular process stored in files of a common XML format or databases and to extract a workflow model based on these traces. The mined workflow models are represented in a graphical editor. With this editor an user can edit the model and export it to a workflow management system. Also, models can be simulated by the tool in order to analyze their performance before deploying them in a workflow management application.

Using a meta-model that consist of *Sequence*, *Parallel*, *Loop* and *Alternative* operators as well as activity occurrence references we have tested the approach on data from different sources. At the one side we used synthetic data. At the other side we used event based data produced by IBM MQSeries Workflow. While executing workflow instances this workflow system logs the events concerning the start and the completion of an activity instance within a particular process instance. The mined models covered the ones which underlies the process instances producing the input data.

8 Related Work

Our approach is close related to the work in [3–8].

Mining workflow models was first considered by Agrawal et al. [3, 4]. Their approach defines a workflow model as a graph supplemented by conditions for transitions between graph nodes. They divide the mining of workflow models into two problems. The first one is called *graph mining*. The approach presents a solution for this problem in form of an algorithm for extracting a graph from event-based data. The algorithm is based on the key concepts of defining transitive relations between activities and building a graph from it that is transformed into its transitive reduction. In contrast to our approach, they consider an activity to be atomic and mix different different paths of execution. The second problem is about supplementing the graph with conditions in order to distinguish alternative and parallel splits within a workflow model. This problem is called *condition mining*. It is not treated by the approach.

Herbst and Karagiannis deal in their approach with mining workflow models with non-unique tasks names [5, 6]. The approach consists of two steps. In the first step, a stochastic task graph is induced from a workflow log. This is done by using a search procedure which embeds a graph generation algorithm in order to find a mapping from activity instances to activity nodes in the graph. The second step transforms the graph into an *ADONIS* workflow model. This step is more extensive than the transformation in our approach because stochastic task graphs do not explicitly outline a synchronization structure, so that this structure has to be extracted in the second step.

In [7, 8] van der Aalst and Weijters present an approach on mining workflow nets which is based on counting frequencies of dependencies between activities. The nodes of a workflow net represent activities found in the workflow log. Dependencies between the activities are represented by arcs between the appropriate nodes. In order to decide whether a dependency is represented by the workflow net they use heuristic rules in combination with threshold values. In addition, their approach deals with noisy logs.

There are many similarities between the approaches above and our approach. We consider our approach to be different by the following facts. First, it aims to extract the most specific model for a given workflow log. Second, it explicitly considers time consuming activities instead of atomic activities. Third, it is based

on a block-structured meta-model that can be supplemented with application specific basic terms and operators.

9 Summary

In this paper we have presented an approach on mining most specific workflow models from event-based data. We considered process mining a special data mining that requires the development of an appropriate process meta-model and the development of algorithms extracting models based on this meta-model from event based data. According to this we have outlined a block-structured process meta-model consisting of a set of basic operators and supplemented by an basic algebra. Based on this meta-model we described our mining process in detail. In order to deploy mined models we have outlined a simple transformation of workflow models into a common exchange format. Also, we have outlined an overview over the experimental evaluation of our approach and related work.

References

1. Fayyad, U., Piatesky-Shapiro, G., Smyth, P.: From data mining to knowledge discovery in databases. AI Magazine (1996) 37–54
2. WorkflowManagementCoalition: Interface 1: Process definition interchange process model. http://www.wfmc.org/standards/docs/TC-1016-P-v11-IF1-Process-definition-Interchange.pdf (1999) Document Number WfMC TC-1016-P.
3. Agrawal, R., Gunopulos, D., Leymann, F.: Mining process models from workflow logs. In: Proceedings of the 6. International Conference on Extending Database Technology (EDBT). (1998) 469–483
4. Agrawal, R., Gunopulos, D., Leymann, F.: Mining process models from workflow logs. IBM Research Report RJ 10100, IBM Almaden Research Center / IBM German Software Development Lab (1998) www.almaden.ibm.com/cs/quest.
5. Herbst, J., Karagiannis, D.: Integrating machine learning and workflow management to support acquisition and adaption of workflow models. In: Proceedings of the Ninth International Workshop on Database and Expert Systems Applications, IEEE (1998) 745–752 IEEE.
6. Herbst, J.: Ein induktiver Ansatz zur Akquisition und Adaption von Workflow-Modellen. Dissertation, Universität Ulm (2001)
7. van der Aalst, W.: Process design by discovery: Harvesting workflow knowledge from ad-hoc executions. (In Jarke, M., O'Leary, D., Studer, R., eds.: Knowledge Management: An Interdisciplinary Approach, Dagstuhl Seminar Report Nr. 281) http://aifbhermes.aifb.uni-karlsruhe.de/dagstuhl-km-2000.
8. van der Aalst, W., Weijters, A.: Workflow mining - discovering workflow models from event-based data. In: Proceedings ECAI Workshop "Knowledge Discovery from Temporal and Spatial Data", ECAI 2002, Lyon (22.06.2002) 78–84 http://tasda.elibel.tm.fr/ecai02/w12.pdf.