Mining Hierarchies of Models: From Abstract Views to Concrete Specifications

Gianluigi Greco[1], Antonella Guzzo[2], and Luigi Pontieri[2]

Dept. of Mathematics[1], UNICAL, Via P. Bucci 30B, 87036, Rende, Italy
ICAR-CNR[2], Via P. Bucci 41C, 87036 Rende, Italy
{ggreco}@mat.unical.it, {guzzo,pontieri}@icar.cnr.it

Abstract. Process mining techniques have been receiving great attention in the literature for their ability to automatically support process (re)design. The output of these techniques is a concrete workflow schema that models all the possible execution scenarios registered in the logs, and that can be profitably used to support further-coming enactments. In this paper, we face process mining in a slightly different perspective. Indeed, we propose an approach to process mining that combines novel discovery strategies with abstraction methods, with the aim of producing hierarchical views of the process that satisfactorily capture its behavior at different level of details. Therefore, at the highest level of detail, the mined model can support the design of concrete workflows; at lower levels of detail, the views can be used in advanced business process platforms to support monitoring and analysis. Our approach consists of several algorithms which have been integrated into a systems architecture whose description is accounted for in the paper as well.

1 Introduction

The difficulties encountered in the design of complex workflows have recently stimulated the development of process mining techniques [1–8], whose aim is to automatically derive a model for the process at hand, based on log data collected during its past enactments. Notably, when a large number of activities and complex behavioral patterns are involved in the analysis, process mining may be a rather trickish task, and the discovered model might fail in representing the process in a clear and concise manner. Indeed, process mining algorithms are generally designed to maximize the accuracy of the mined model, i.e., they equip the model with as many variants as they are required to support all the registered logs; therefore, the resultant schema is well-suited for supporting the enactment, but is less useful for a business user who wants to monitor and analyze the business operation at some appropriate level of abstraction.

To overcome this limitation, we propose an approach to process mining that produces a hierarchical process model which satisfactorily captures the behavior of the process at hand, by providing different views at different level of details. Roughly speaking, the model is essentially a tree such that the root encodes the most abstract view, which has no pretension of being an executable workflow, whereas any level of internal nodes encodes a refinement of such an abstract model, in which some specific details are introduced.

The capability of discovering a modular and expressive description for a process can be a valid help in designing, monitoring, and analyzing process models, and can pave the way for effectively reusing, customizing and semantically consolidating process knowledge. And, in fact, the need and the usefulness of process hierarchies/taxonomies has already emerged in several applicative contexts, and process abstraction is currently supported in some advanced platforms for business management (e.g, iBOM [9], ARIS [10]), in which the designer can manually define the relationships among the abstract and the actual process.

In the literature, the definition of process hierarchies was first considered in [11], envisaging a repository of process descriptions for supporting both design and sharing of process models. The notion of process specialization/generalization (w.r.t some suitable behavioral semantics) has been investigated for different modelling formalisms, such as Object Behavior Diagrams [12], UML diagrams [13], process-algebra specifications and Petri-nets [14, 3], DataFlow diagrams [15]. Recently, some abstraction techniques aiming at summarizing complex processes have been proposed in [9, 16].

The main distinguishing feature of our approach with respect to the proposals cited above is the combination of mining and abstraction methods for automatically producing a hierarchical process model. This entails that no substantial human intervention is required while abstracting process schemas, so that software modules implementing the algorithms described in the paper represent a valuable add on for advanced process management platforms. In more details, the contribution of the paper is as follows:

- In Section 3, we introduce a top-down clustering algorithm that generates a hierarchy of workflow schemas, by inducing each of them from a homogeneous cluster of traces. Since, at each step, the algorithm greedily splits the cluster equipped with the least sound schema, schemas at the leaves of the hierarchy effectively model different usage-scenarios for the process.
- The whole hierarchy build by means of clustering is of great value in structuring different execution classes into an effective taxonomical view. In Section 4, we propose an algorithm for obtaining a taxonomy of schemas, by producing, for each non-leaf node of the hierarchy, an abstract schema generalizing all those associated with the children.
- In Section 5, we present an abstraction algorithm and some associated metrics, which are meant to support the above generalization algorithm by properly replacing groups of "specific" activities with "higher-level" activities.
- Finally, in Section 6, we sketch the architecture of a system implementing the whole approach, and discuss some concluding remarks and future works.

2 Formal Framework

In this section, we introduce the basic notions and notation for formally representing workflow models, which will be exploited in the rest of the paper. The

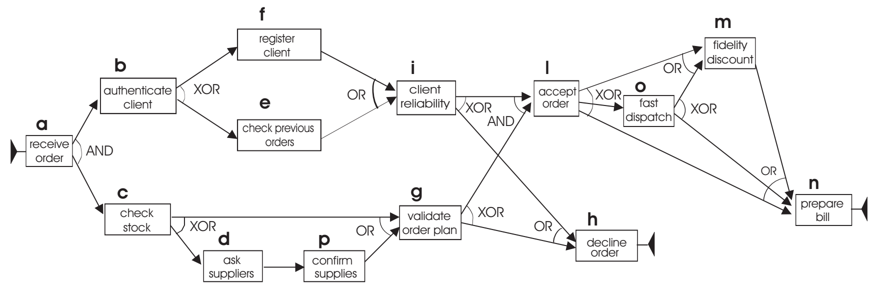


Fig. 1. Workflow schema for the sample *OrderManagement* process.

control flow graph of a process P is a tuple $\langle A, E, a^0, F\rangle$, where: A is a finite set of *activities*, $E \subseteq (A - F) \times (A - \{a^0\})$ is a relation of precedences among activities, $a^0 \in A$ is the starting activity, $F \subseteq A$ is the set of final activities. A control flow graph defines the potential orderings according to which the activities of P can be executed; it is often enriched with some kind of constraints imposing further restrictions on the executions.[1] For any a activity of the workflow schema, the split constraint for a is: *(S.i) AND-split* if a activates all of its successor activities, once completed; *(S.ii) OR-split*, if a may activate any number (non-deterministically chosen) of its successor activities, once completed; *(S.iii) XOR-split* if a activates exactly one out of all its successor activities, once completed. The join constraint for a is: *(J.i) AND-join* if a can be executed only after all of its predecessors have notified a to start; *(J.ii) OR-join*, if a can be executed as soon as one of its predecessors notifies a to start.

Let P be a process. A *workflow schema* for P, denoted by $\mathcal{W}(P)$, is a tuple $\langle A, E, a^0, F, \mathcal{C}\rangle$, where $\langle A, E, a^0, F\rangle$ is a control flow graph for P, and $\mathcal{C}$ is a set of constraints for the activities in A. Fig. 1 shows a possible workflow schema for the *OrderManagement* process of handling customers' orders in a business company. Constraints are drawn by means of labels beside the tasks – e.g., `accept order` is an *and-join* activity as it must be notified by its predecessors that both the client is reliable and the order can be supplied correctly.

Each time a workflow is enacted in a workflow management system, it produces an *instance*, i.e., a suitable subgraph of the schema, containing both initial and final activity, that satisfies all the constraints. Actually, many process-oriented commercial systems store partial information about the various instances of a process, by tracing some events related to the execution of its activities. In particular, the *logs* kept by most of such systems simply consist of sequences of event occurrences, which, in general, cannot allow to reconstruct the structure of all workflow instances. Let A_P be the set of task identifiers for the process P; then, a *workflow trace s over A_P* is a string in A_P^*, representing a task sequence. For instance, in our running example, a trace can be encoded

[1] We do not refer to any specific syntax proposed for expressing constraints; rather, we deal with some basic features occurring in the most typical workflow systems.

by the string `acbgih`. A *workflow log for* P, denoted by $\mathcal{L}_P$, is a bag of traces over A_P, i.e., $\mathcal{L}_P = [\ s \mid s \in A_P^* \ ]$.

We next formalize the relationship between traces and instances. Let I be an instance of a workflow schema $\mathcal{W}$, and s be a trace in $\mathcal{L}_P$. Then, s is *compliant with* $\mathcal{W}$ *through* I, denoted by $s \models^I \mathcal{W}$, if the last activity of s is a final activity w.r.t. to $\mathcal{W}$ and there exists a topological sort s' of I such that s is a prefix of s'. Furthermore, s is simply said to be *compliant with* $\mathcal{W}$, denoted by $s \models \mathcal{W}$, if there exists an instance I such that $s \models^I \mathcal{W}$.

Finally, the following functions allow to evaluate the degree of conformance of $\mathcal{W}$ w.r.t. a given log $\mathcal{L}_P$: *(i)* $soundness(\mathcal{W}, \mathcal{L}_P)$, expressing the percentage of instances of $\mathcal{W}$ which have some corresponding traces in $\mathcal{L}_P$, and *(ii)* $completeness(\mathcal{W}, \mathcal{L}_P)$, which measures the percentage of traces in $\mathcal{L}_P$ that are compliant with $\mathcal{W}$. It is worth noticing that both soundness and completeness should be considered during the process mining task, in order to discover a schema that satisfactorily model the input traces.

3 Mining Hierarchies of Workflow Schemas

Our approach to discover expressive process models at different level of details is articulated in two phases. First, we mine a hierarchy of workflow schemas, by means of a hierarchical top-down clustering algorithm, called `HierarchyDiscovery`. Then, we visit the mined model in a bottom-up way, i.e., from the leaves to the root, and we restructure it at several levels of abstraction, by means of the algorithm `BuildTaxonomy`. Details on the former phase are reported in this section, whereas details on the latter are reported in Section 4.

3.1 Algorithm `HierarchyDiscovery`

A process mining technique that is specifically tailored for complex process, involving lots of activities and exhibiting different variants has been presented in [8]. It relies on the idea of explicitly representing all the possible usage scenarios by means of a collection of different, specific, workflow schemas, in order to obtain a modular representation of the process itself, which is yet sounder than a single workflow schema mixing all of them. We here propose a new algorithm that extends the one presented in [8] by allowing the computation of hierarchical process models rather than simple collections of workflow schemas. The mined model is now meant to be a hierarchy of workflow schemas that collectively represent the process at different levels of granularity and abstraction: the set of schemas corresponding to children of any node v represents the same set of execution as v, but in a more detailed and sounder way, as different subclass of executions are separately described. We next formalize the notion of hierarchical model.

Definition 1. Let $\mathcal{L}_P$ be a set of log traces for a process P. Then, a *schema hierarchy* for P is a tuple $\mathcal{H} = \langle \mathcal{WS}, T, \lambda \rangle$, such that:

```
Input: A set of log traces $\mathcal{L}_P$, two natural numbers $maxSize$ and $k$, a threshold $\gamma$.
Output: A schema hierarchy for $P$.
Method: Perform the following steps:
$W_0$ :=mineWFschema($\mathcal{L}_P$);
$\mathcal{WS} := \{W_0\}$;
$Traces[W_0] := \mathcal{L}_P$;    // $Traces[W_i]$ refers to the log traces modelled by $W_i$, $\forall W_i \in \mathcal{WS}$
$T := \langle \{ v_0 \}, \emptyset, v_0 \rangle$;
$\lambda(v_0) := W_0$;
while $|\mathcal{WS}| \leq maxSize$ and $soundness(\langle \mathcal{WS}, T, \lambda \rangle, \mathcal{L}^P) < \gamma$ do
  let $W_q$ be the least sound "leaf" schema [a] and $v_q{=}\lambda^{-1}(W_q)$ be its associated node in $T$;
  let $n{=}|\mathcal{WS}|$ be the number of schemas currently stored in $\mathcal{WS}$;
  $\langle L_{n+1}, ..., L_{n+k} \rangle$ := partition-FB($Traces[W_q]$);
  if $k > 1$ then
    for $h = 1..k$ do
      $W_{n+h}$ := mineWFschema($L_{n+h}$);
      $\mathcal{WS} := \mathcal{WS} \cup \{W_{n+h}\}$;
      $Traces[W_{n+h}] := L_{n+h}$;
      $T.V := T.V \cup \{v_{n+h}\}$;    $T.E := T.E \cup \{(v_q, v_{n+h})\}$;
      $\lambda(v_{n+h}) := W_{n+h}$;
    end for
  end if
end while
return $\langle \mathcal{WS}, T, \lambda \rangle$;
```

[a] i.e., $W_q = argmin_{W \in \mathcal{WS}} \{soundness(W, traces(W)) \mid \lambda^{-1}(W)$ is a leaf of $T\}$

Fig. 2. Algorithm `HierarchyDiscovery`

- $\mathcal{WS}$ is a set of workflow schemas for P;
- $T = \langle V, E, v_0 \rangle$ is a tree, where V (resp. E) denotes the set of vertices (resp. edges), and $v_0 \in V$ is the root;
- $\lambda : V \mapsto \mathcal{WS}$ is a bijective function associating each vertex $v \in V$ with a workflow schema $\lambda(v)$ in $\mathcal{WS}$;

Soundness and completeness of $\mathcal{H}$ are defined as follows: *(i)* $soundness(\mathcal{H}, \mathcal{L}_P)$ is the percentage of the instances modelled by the schemas associated with the leaves of T that have some corresponding trace in $\mathcal{L}_P$, *(ii)* $completeness(\mathcal{H}, \mathcal{L}_P)$ is the percentage of traces in $\mathcal{L}_P$ that are compliant with at least one schema associated with a leaf of T. □

Notice that for each vertex v in V, the set S_v of the schemas associated with the children of v, i.e., $S_v = \{\lambda(v_i^c) \mid (v, v_i^c) \in E\}$, is essentially meant to model the same set of instances modelled by $\lambda(v)$, but in a sounder way. Therefore, the union of all the schemas associated with the leaves constitute, as a whole, the soundest model for the process.

Given a log $\mathcal{L}_P$, we can discover a schema hierarchy for P by recursively partitioning the traces in $\mathcal{L}_P$ into clusters, according to the different behavioral patterns they exhibit, and building a schema for each of these clusters. This is accomplished by the algorithm `HierarchyDiscovery` (see Fig. 2), where the function `mineWFschema` is exploited for discovering each single workflow schema in the hierarchy. Some possible implementations of `mineWFschema` are discussed in [1–8], and essentially consist in discovering precedence relationships and constraints that involve the activities.

The meaning of the other input parameters is as follows: γ is a (lower) threshold for the soundness of the mined hierarchy H, while $maxSize$ and k bound the total number of nodes in H and their out-degrees, respectively. Notice that we here assume that the discovered model must have maximal completeness. Obviously, we can straightforwardly extend the approach to discovering not fully complete models, e.g., by introducing a threshold for completeness and using some implementation of `mineWFschema` taking account for such a threshold.

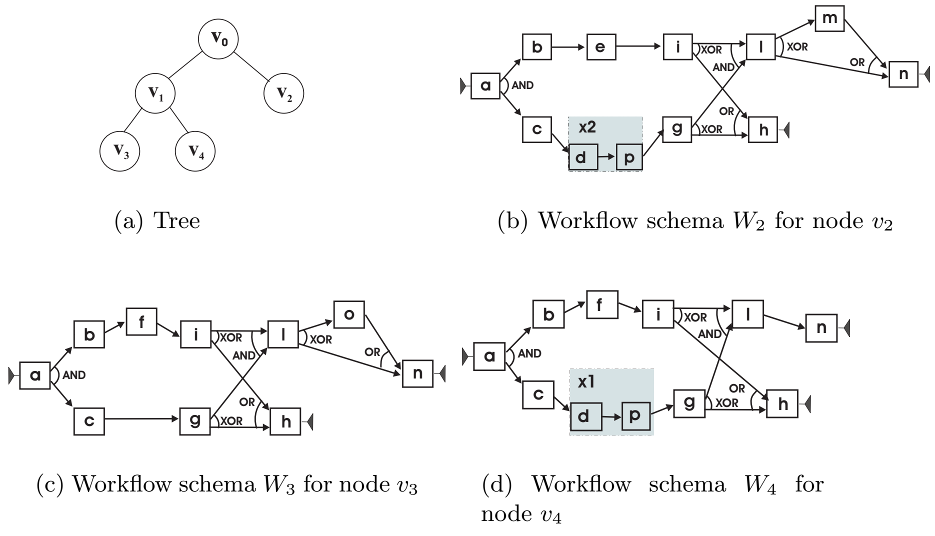
The algorithm starts by building a workflow schema W_0 (Line 1) which is a first attempt to represent the behavior captured in the log traces, and which will be the only component of $\mathcal{WS}$ (Line 2). The schema $\mathcal{WS}_0$ is associated with the whole log via the auxiliary structure $Traces$ (Line 3), which enables for recording the set of traces each discovered schema was derived from. Moreover, the tree T is initialized with a single node (its root) v_0, which is associated with W_0 by properly setting the function λ (Lines 4-5).

In order to produce a more accurate model, we greedily chose to refine the least sound schema W_q in $\mathcal{WS}$ and to derive a set of more refined schemas (Lines 7-18) as children of node corresponding to W_q. To this purpose, the set of traces modelled by the selected schema W_q is partitioned through the procedure `partition-FB` (Line 9) into a set of clusters which, in a sense, are more homogeneous from a behavioral viewpoint. Roughly speaking, the procedure mainly relies on the discovery of frequent rules representing behavioral patterns that were unexpected with respect to W_q. Such rules are then used to map the traces into a feature space, where classical clustering methods can be applied (see [8] for more details).

For each new cluster L_{i+h} a specific workflow schema W_{i+h} is extracted, by using again function `mineWFschema`, and added to $\mathcal{WS}$ (Lines 10-11). Moreover, W_{i+h} is associated with the cluster L_{i+h} it was induced from, and with a new node in the tree, which is a child of the node corresponding to the refined schema W_q (Lines 14-16). The whole process of refining a schema can then be iterated in a recursive way, by selecting again the least sound leaf schema in the current hierarchy, until the desired value γ of soundness has been achieved or too many schemas (i.e., $maxSize$ or more) are already in $\mathcal{WS}$ (Line 6).

Example 1. In order to provide some insight on how the algorithm works, we report a few notes on its behavior when used to mine a synthesized log. To this purpose $100,000$ traces for the workflow schema shown in Fig. 1 were randomly generated by means of the generator described in [17]. Notably, in the generation of the log, we also required that task m could not occur in any execution trace containing f, and that task o could not appear in any trace containing d and p, thereby modelling the intuitive restriction that a fidelity discount in never applied to a new customer, and that a fast dispatching procedure cannot be performed whenever some external supplies were asked for. These additional constraints allow us to simulate the presence of different usage scenarios that cannot be captured by a simple workflow schema.

The output of `HierarchyDiscovery`, for $maxSize = 5$ and $\gamma = 0.85$, is the schema hierarchy reported in Fig. 3.(a), where each node logically corresponds



(a) Tree

(b) Workflow schema W_2 for node v_2

(c) Workflow schema W_3 for node v_3

(d) Workflow schema W_4 for node v_4

Fig. 3. Hierarchy generated by `HierarchyDiscovery` (details for leaf schemas only).

to both a cluster of traces and a workflow schema induced from that cluster by means of traditional algorithms for process mining. Thus, node v_0 corresponds to the whole set of traces and to an associated (mined) workflow. Actually, the algorithm `HierarchyDiscovery` finds that the schema of v_0 is not as sound as required by the user, and therefore partitions the traces by means of a clustering algorithm (k-means in the implementation). In the example, we fix $k = 2$ and the algorithm generates two children v_1 and v_2; then, v_2 is not further refined (due to its high soundness), while traces associated with v_1 are split again into v_3 and v_4. At the end, the schemas associated with the leaves of the tree are those shown in the Figure. As a matter of fact, schemas W_0 and W_1 (associated with v_0 and v_1, respectively) are only preliminary attempts to model executions that are, indeed, modelled in a sounder way by the leaf schemas. Nevertheless, the whole hierarchy is an important result as well, for it somehow structures the discovered execution classes, and is a basis for deriving a schema taxonomy representing the process at different abstraction levels, as it will be discussed in Section 4. $\square$

4 Restructuring Schema Hierarchies

In the second phase of our approach, we exploit the schema hierarchy produced by `HierarchyDiscovery`, in order to restructure it for producing a description of the process at different levels of details. Intuitively, leaf nodes stand for concrete usage scenarios, whereas non-leaf nodes are meant to represent suitable generalizations of the different process models corresponding to their children. Relations among activities are next formalized by means of abstraction dictionaries.

4.1 Abstraction Relationships

Let A be a set *activities*. An abstraction dictionary for A is a tuple $\mathcal{D} = \langle \mathcal{I}sa, \mathcal{P}artOf \rangle$, such that $\mathcal{D}.\mathcal{I}sa \subseteq A \times A$, $\mathcal{D}.\mathcal{P}artOf \subseteq A \times A$ and, for each $a \in A$, $(a, a) \notin \mathcal{D}.\mathcal{P}artOf$ and $(a, a) \notin \mathcal{D}.\mathcal{I}sa$. Roughly speaking, for two activities a and b, $(b, a) \in \mathcal{D}.\mathcal{I}sa$ indicates that b is a refinement of a; conversely, $(b, a) \in \mathcal{D}.\mathcal{P}artOf$ indicates that b is a component of a.

Given two activities a and a', we say that a *generalizes* a' w.r.t. a given abstraction dictionary $\mathcal{D}$, denoted by $a \upharpoonleft^{\mathcal{D}} a'$, if there is a sequence of activities $a_0, a_1, .., a_n$ such that $a_0{=}a'$, $a_n{=}a$ and $(a_i, a_{i-1}) \in \mathcal{D}.\mathcal{I}sa$ for each $i = 1..n$; we call such a sequence a *genpath* from a' to a with length n. Moreover, the *generalization distance* between a and a' w.r.t. $\mathcal{D}$, denoted by $dist_G^{\mathcal{D}}$, is the minimal length of the *genpaths* connecting a' to a, or vice-versa. As a special case, we assume that $dist_G^{\mathcal{D}}(a, a) = 0$ for any activity a. Finally, the *most specific generalization* of two activities x and y w.r.t. $\mathcal{D}$, denoted by $msg^{\mathcal{D}}(x, y)$, is the closest activity, if there exists one, that generalizes them both, i.e., $msg^{\mathcal{D}}(x, y) = argmin_z\{dist_G^{\mathcal{D}}(x, z) + dist_G^{\mathcal{D}}(y, z) \mid z \upharpoonleft^{\mathcal{D}} x \text{ and } z \upharpoonleft^{\mathcal{D}} y\}$.

Given two activities a and a' and an abstraction dictionary $\mathcal{D}$, we say that a *implies* a' w.r.t. $\mathcal{D}$, denoted by $a \longrightarrow^{\mathcal{D}} a'$, if $(a', a) \in \mathcal{D}.\mathcal{I}sa$ or $(a', a) \in \mathcal{D}.\mathcal{P}artOf$ or, recursively, there exists an activity x such that $a \longrightarrow^{\mathcal{D}} x$ and $x \longrightarrow^{\mathcal{D}} a'$. The set of activities implied by a w.r.t. $\mathcal{D}$ is referred to as $impl^{\mathcal{D}}(a)$, i.e., $impl^{\mathcal{D}}(a) = \{a' \mid a \longrightarrow^{\mathcal{D}} a'\}$. An activity a is then said to be *complex* if there exists at least one activity x such that $a \longrightarrow^{\mathcal{D}} x$; otherwise, a is a *basic* activity. In other words, complex activities represent higher level concepts defined by aggregating or generalizing basics activities actually occurring in real process executions.

The above relationship between activities is a basic block for building taxonomies that can significantly reduce the efforts for comprehending and reusing process models, for they structuring process knowledge into different abstraction levels. Let $\mathcal{W}_1$ and $\mathcal{W}_2$ be two workflow schemas over the sets of activities A_1 and A_2, respectively. Then, we say that $\mathcal{W}_2$ *specializes* $\mathcal{W}_1$ ($\mathcal{W}_1$ *generalizes* $\mathcal{W}_2$) w.r.t. a given abstraction dictionary $\mathcal{D}$, denoted by $\mathcal{W}_2 \prec^{\mathcal{D}} \mathcal{W}_1$, if *(i)* for each activity a_2 in A_2 there exists at least one activity a_1 in A_1 such that $a_1 \longrightarrow^{\mathcal{D}} a_2$, and *(ii)* there is no activity b_1 in A_1 such that $a_2 \longrightarrow^{\mathcal{D}} b_1$.

The output of the restructuring of a schema hierarchy is an abstraction dictionary and a schema taxonomy as for formalized below.

Definition 2. Let $\mathcal{D}$ be an abstraction dictionary for the activities of a given process P, and $\mathcal{H} = \langle \mathcal{WS}, T, \lambda \rangle$ be a schema hierarchy for P. Then, $\mathcal{H}$ is a *schema taxonomy* for P w.r.t. $\mathcal{D}$ if for any pair of nodes v and v_c in V such that $(v, v_c) \in T.E$ (i.e., v_c is a child of v) $\lambda(v) \prec^{\mathcal{D}} \lambda(v_c)$. □

4.2 Algorithm `BuildTaxonomy`

In Fig. 4 we illustrate an algorithm, called `BuildTaxonomy`, for restructuring a schema hierarchy into a schema taxonomy, representing the process at hand at several abstraction levels. The algorithms takes in input a schema hierarchy $\mathcal{H}$ and produces a taxonomy $\mathcal{G}$ and an abstraction dictionary $\mathcal{D}$, which $\mathcal{G}$ has been build according to. Roughly speaking, the basic task allowing for such a generalization consists in replacing groups of "specific" activities, appearing in the schemas to be generalized, with new "virtual" activities which represent them at a higher level of abstraction. In this way, a more compact description of the process is obtained, where portion of the actual workflow are represented at a lower level of granularity. Indeed, during such a restructuring process, the abstraction dictionary $\mathcal{D}$ is required to maintain the relationships between the activities that were abstracted and the new higher-level concepts replacing them.

```
Input: A schema hierarchy $\mathcal{H} = \langle \mathcal{WS}, T, \lambda \rangle$;
Output: A schema taxonomy $\mathcal{G}$, an abstraction dictionary $\mathcal{D}$;
Method: Perform the following steps:
 1   let $T = \langle V, E, v_0 \rangle$, and let $\mathcal{D} := \emptyset$;
 2   $Done := \{ v \in V \mid \nexists v' \in V$ s.t. $(v, v') \in E \}$;   // $Done$ initially contains the leaves of $T$;
 3   while $\exists v \in V$ such that $v \notin Done$, and $\{c \mid (c, v) \in E\} \subseteq Done$ do
 4     let $ChildSchs = \{ \lambda(c) \mid v \in V$ and $(v, c) \in E \}$, i.e., the schemas of all $v$'s children;
 5     $\lambda'(v) :=$ generalizeSchemas($ChildSchs, \mathcal{D}$);
 6     $Done := Done \cup \{v\}$;
 7   end while
 8   $\mathcal{G} := \langle \mathcal{WS}, T, \lambda' \rangle$;
 9   normalizeDictionary($\mathcal{G}, \mathcal{D}$);
10   return $(\mathcal{G}, \mathcal{D})$;
-------------------------------------------------------------------------------
Procedure generalizeSchemas( $\mathcal{WS} = \{W_1, ..., W_n\}$: set of workflow schemas,
                              var $\mathcal{D}$: abstraction dictionary ): workflow schema;
 g1   let $W_h = \langle A_h, E_h, a^0_h, F_h, \mathcal{C}_h \rangle$ for $h = 1..n$;
 g2   let $I = \bigcap_{i=1}^{n} A_i$;
 g4   $\overline{W} := \left\langle \bigcup_{i=1}^{n} A_i,\ \bigcup_{i=1}^{n} E_i,\ \bigcup_{i=1}^{n} a^0_i,\ \bigcup_{i=1}^{n} F_i,\ \emptyset \right\rangle$;
 g5   mergeConstraints($\overline{W}$, $\{\mathcal{C}_h \mid h = 1..n\}$);
 g6   for each $i = 1..n$ do
 g7       abstractActivities($A_i$-$I$, $\overline{W}$, $\mathcal{D}$);
 g8   end for
 g9   abstractActivities($\overline{W}.A$-$I$, $\overline{W}$, $\mathcal{D}$);
g10   return $\overline{W}$;
```

Fig. 4. Algorithm `BuildTaxonomy`

The algorithm works in a bottom-up fashion (Line 2-7): starting from the leaves of the input hierarchy, it produces, for each non-leaf node v, a novel workflow schema that generalizes all the schemas associated with the children of v. Notably, such a schema is meant to accurately represents only the features that are shared by all the subsets of executions corresponding to the children of v, while abstracting from specific activities, which are actually merged into new high-level (i.e., *complex*) activities. Such a generalization task is carried out by providing the procedure `generalizeSchemas` with the schemas associated with the children of v, along with the abstraction dictionary $\mathcal{D}$, initially empty (Line

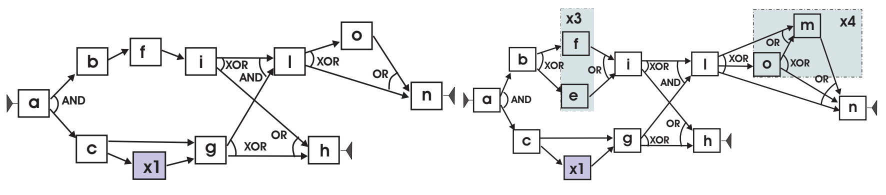
5). As a result, a new generalized schema is computed and assigned to v through the function λ'; moreover, $\mathcal{D}$ is updated to suitably relate the activities that were abstracted with the complex ones replacing them in the generalized schema.

As a final step, after the schema taxonomy $\mathcal{G}$ has been computed, the algorithm also restructures the abstraction dictionary $\mathcal{D}$ by using the procedure `normalizeDictionary` (Line 9), which actually removes all "superfluous" activities that were created during the generalization. In particular, this step will eliminate any complex activity a not appearing in any schema of $\mathcal{G}$, which can be abstracted into another, higher-level, complex activity b, provided that this latter can suitably abstract all the activities implied by a.

Clearly enough, the effectiveness of the technique depends on the way the generalization of the activities and the updating of the dictionary are carried out. Procedure `generalizeSchemas` (reported in Fig. 4 as well) first merges all the input workflow schemas into a preliminary workflow schema $\overline{W}$ (Line g4), which represents all the possible flow links in the input workflows by roughly performing the union of their corresponding control flow graphs. Subsequently, the set of constraints of $\overline{W}$ (initially empty) is populated by suitably combining the constraints specified in the input schemas, by means of procedure `mergeConstraints` (Line g5); as a matter of fact, this latter procedure derives a split (resp., join) condition for each activity a of $\overline{W}$, based on the split (resp., join) conditions a is associated with in each input schema, yet taking into account all the control flow relationships a takes part to, in the involved schemas.

The main task in the generalization process is performed by repeatedly applying the procedure `abstractActivities`, which transforms $\overline{W}$ by merging activities in the reference set it receives as the first parameter, and by updating the associated constraints and the abstraction dictionary $\mathcal{D}$ as well. In particular, `abstractActivities` is first applied for merging only activities that derived from the same input schema – at step i only activities coming from the i-th schema can be merged (Line g7). A further application of `abstractActivities` is then performed to possibly abstract any non-shared activity in the current schema, independently of its origin. Due to its relevance to the generalization algorithm, `abstractActivities` is illustrated in details in Section 5; however, we conclude this description by providing an intuition on its behavior.

Example 2. Consider again the schema hierarchy shown in Fig. 3. Then, algorithm `BuildTaxonomy` starts generalizing from the leafs, thus first processing the schemas W_3 and W_4 associated with v_3 and v_4, respectively. The result of this generalization is the schema $\overline{W}_1$ shown in Fig. 5.(a), which is obtained by first merging all the activities and flow links contained in either W_3 or W_4, and by then performing a series of abstractions steps over all non-shared activities, namely o, d and p. As we shall formalize in Section 5, in general, we iteratively abstract a pair of activities into a complex one, trying to minimize the number of spurious flow links that their merging introduces between the remaining activities, and yet considering their mutual similarity w.r.t. the contents of the abstraction dictionary. When deriving the schema $\overline{W}_1$, only the activities d and p are abstracted, by aggregating them both into the new complex activity x_1;



(a) Workflow schema $\overline{W}_1$ for node v_1 (b) Workflow schema $\overline{W}_0$ for node v_0

Fig. 5. Generalized workflow schemas in the resulting taxonomy.

consequently, d and p are replaced with x_1, while the pairs (d, x_1) and (p, x_1) are inserted in the $\mathcal{P}artOf$ relationship. The schema $\overline{W}_1$ is then merged with the schema W_2 associated with v_2, and a new generalized schema, shown in Fig. 5.(b), is derived for the root v_0. In fact, when abstracting activities coming from W_2, d and p are aggregated again together, into a new complex activity x_2; however, in a subsequent step x_2 is incorporated into x_1, as these two complex activities have the same set of sub-activities and the same control flow links. Furthermore, the activities e and f are aggregated into the complex activity x_3, while m and o are aggregated into x_4. As a consequence, the pairs $(e, x_3), (f, x_3), (m, x_4)$ and (o, x_4) are added to the $\mathcal{P}artOf$ relationship. $\square$

5 Abstracting Workflow Activities

In this section, we discuss the implementation of the `abstractActivities` procedure. To this aim, we preliminary introduce some metrics that we exploit for singling out those activities that can be safely abstracted into higher-level ones.

5.1 Matching Activities for Abstraction Purposes

We next describe a series of functions which are meant to provide different ways for evaluating how much two activities are suitable to being abstracted by a single higher-level activity. Roughly speaking, $sim_P^{\mathcal{D}}$ and $sim_G^{\mathcal{D}}$ aims at capturing semantical affinities based on the contents of a given abstraction dictionary $\mathcal{D}$; on the contrary, sim^E just compares two activities from a topological viewpoint according to a set E of control flow edges.

While merging tasks in a workflow schema, a major concern is to limit the creation of spurious control flow paths among the remaining activities in the workflow schema, yet admitting to lose some precedence relationships involving the abstracted ones. In this respect, we focus on two cases that can lead to a meaningful merging without upsetting the topology of the control flow graph, as formalized in the following definition.

Definition 3. Given a set of edges E , we say that an (unordered) pair of activities (x, y) is *merge-safe* if one of the following conditions holds:

a) x and y are directly linked by some edges in E and after removing these edges no other path exists connecting x and y, i.e., $\{(x, y), (y, x)\} \cap E \neq \emptyset$ and $\{(x, y), (y, x)\} \cap (E - \{(x, y), (y, x)\})^* = \emptyset$

b) there is no path in E connecting x and y, i.e., $\{(x, y), (y, x)\} \cap E^* = \emptyset$

where E^* denotes the transitive closure of E. $\square$

Notably, only in the case (b) of Definition 3 the merging of x and y may lead to spurious dependencies among other activities in the schema. Indeed, this happens when there are two other activities z and w such that $(z, w) \notin E^*$, and either $\{(z, x), (y, w)\} \subseteq E$ or $\{(z, y), (x, w)\} \subseteq E$.

By the way, a straight way for preventing this problem, consists in requiring that at least one of the following conditions holds: *(i)* $\mathcal{P}_x = \mathcal{P}_y$, *(ii)* $\mathcal{S}_x = \mathcal{S}_y$, *(iii)* $\mathcal{P}_x \subseteq \mathcal{P}_y$ and $\mathcal{S}_x \subseteq \mathcal{S}_y$, *(iv)* $\mathcal{P}_y \subseteq \mathcal{P}_x$ and $\mathcal{S}_y \subseteq \mathcal{S}_x$, where $\mathcal{P}_a$ (resp. $\mathcal{S}_a$) denotes the set of predecessors (resp. successors) of activity a, according to the arcs in E. Actually, in order to also deal with the presence of complex activities in the set of predecessors (resp., successors), we extend the above expressions by replacing $\mathcal{P}_a$ (resp., $\mathcal{S}$) with $\mathcal{P}_a^+$ (resp., $\mathcal{S}_a^+$), defined as follows:

$$\mathcal{P}_a^+ = \bigcup_{b \in \mathcal{P}_a} impl(b) \qquad \mathcal{S}_a^+ = \bigcup_{b \in \mathcal{S}_a} impl(b)$$

However, the above requirements on the flow relationships of two activities could not allow for an appreciable level of abstraction. Therefore, we somehow incorporate them, in a smoothed way, into the function $sim^E(x, y)$, reported below, which is meant to evaluate a pair of activities according to the number of spurious flows that would be generated when merging them, in an inverse manner (i.e, the more spurious flows are introduced, the lower is the score):

$$sim^E(x, y) = \frac{\alpha(\mathcal{P}_x^+, \mathcal{P}_y^+) \times \alpha(\mathcal{S}_x^+, \mathcal{S}_y^+) + \beta(\mathcal{P}_x^+, \mathcal{P}_y^+) \times \beta(\mathcal{S}_x^+, \mathcal{S}_y^+)}{2}$$

where, for any two sets B and C, $\alpha(B, C) = \frac{|B \cap C|}{min(|B|, |C|)}$ and $\beta(B, C) = \frac{|B \cap C|}{|B \cup C|}$.

As a matter of facts, sim^E produces a maximal value whenever one of the "strong" conditions discussed before holds, and, in general, tends to attribute high similarity to activities matching in most of their predecessors (successors).

On the contrary, function $sim_P^{\mathcal{D}}$ provides a way for measuring "semantical" similarities between two activities x and y, based on the implied activities they actually share. It is defined as:

$$sim_P^{\mathcal{D}}(x, y) = \beta(impl^{\mathcal{D}}(x) \cup \{x\}, impl^{\mathcal{D}}(y) \cup \{y\})$$

Moreover, function $sim_G^{\mathcal{D}}$, which is instead devoted to compare two activities based on the generalization relationships recorded in $\mathcal{D}.\mathcal{I}sa$, is defined as follows:

$$sim_G^{\mathcal{D}}(x,y) = 1 - \frac{dist_G^{\mathcal{D}}(x, msg^{\mathcal{D}}(x,y)) + dist_G^{\mathcal{D}}(y, msg^{\mathcal{D}}(x,y))}{max\{dist_G^{\mathcal{D}}(a,b) \mid a,b \in A \text{ and } b \uparrow^{\mathcal{D}} a\}}$$

Finally, an overall score can be assigned to each pair of activities in order to rank them for abstraction purposes, as follows:

$$score^{\mathcal{D},E}(x,y) = \begin{cases} 0, \text{ if } (x,y) \text{ is not a merge-safe pair of activities} \\ max\{sim^E(x,y), sim_P^{\mathcal{D}}(x,y), sim_G^{\mathcal{D}}(x,y)\}, \text{ otherwise} \end{cases}$$

5.2 Abstracting Activities

Fig. 6 provides a detailed description of procedure `abstractActivities`, that is meant to merge activities in S for a given schema $\bar{W}$ and to abstract them via higher-level, complex, activities. To this aim, besides $\bar{W}$ and S, the procedure takes in input an abstraction dictionary $\mathcal{D}$. As a result, it transforms $\bar{W}$ by replacing the abstracted activities with the associated complex ones, and modifies $\mathcal{D}$ in order to suitably record the performed abstraction transformations.

```
Procedure abstractActivities($S$: set of activities; var $\bar{W} = \langle A, E, a^0, F, \mathcal{C}\rangle$: a workflow schema;
                    var $\mathcal{D} = \langle \mathcal{P}artOf, \mathcal{I}sa\rangle$: abstraction dictionary; )
  1 let $E' = \{(x,y) \in E$ s.t. $x \in S$ and $y \in S\}$;
  2 $\langle m_1, m_2, p, mode\rangle$ :=getBestAbstraction($S, E', \mathcal{D}$);
  3 while $p \neq \varepsilon$ do
  4   let $ActuallyAbstracted = \{m_1, m_2\} - \{p\}$;
  5   if $mode$ = ISA then
  6     $\mathcal{I}sa := \mathcal{I}sa \cup \{(x,p)$ s.t. $x \in ActuallyAbstracted\}$;
  7   else
  8     $\mathcal{P}artOf := \mathcal{P}artOf \cup \{(x,p)$ s.t. $x \in ActuallyAbstracted\}$;
  9   end if
 10   deriveConstraints($\mathcal{C}, m_1, m_2, p, E$);
 11   arrangeEdges($E, ActuallyAbstracted, p$);
 12   $A := A - ActuallyAbstracted \cup \{p\}$;
 13   $S := S - ActuallyAbstracted \cup \{p\}$;
 14   $\langle p, m_1, m_2\rangle$ :=getBestAbstraction($S, E', \mathcal{D}$);
 15 end while
```

```
Procedure getBestAbstraction($S$: set of activities; $E$: set of activity pairs;
                    $\mathcal{D}$: abstraction dictionary): a tuple in $S \times S \times \{\mathcal{I}sa, \mathcal{P}artOf, \varepsilon\} \times \mathcal{A}$; [a]
 b1 if $|S| < 2$ then
 b2   return $\langle \varepsilon, \varepsilon, \varepsilon, \varepsilon\rangle$;
 b3 else
 b4   let $a$ and $b$ be two activities s.t. $score(a,b) = max\{score^{\mathcal{D},E}(x,y) \mid x,y \in S\}$;
 b5   if $score^{\mathcal{D},E}(a,b) < \rho$ then return $\langle \varepsilon, \varepsilon, \varepsilon, \varepsilon\rangle$;
 b6   else if $sim_G^{\mathcal{D}}(a,b) \geq \rho^s$ then return $\langle a, b, \mathcal{I}sa, msg^{\mathcal{D}}(a,b)\rangle$;
 b7   else if $impl^{\mathcal{D}}(b) \subseteq impl^{\mathcal{D}}(a)$ then return $\langle a, b, \mathcal{P}artOf, a\rangle$;
 b8   else if $impl^{\mathcal{D}}(a) \subseteq impl^{\mathcal{D}}(b)$ then return $\langle a, b, \mathcal{P}artOf, b\rangle$;
 b9   else if $sim_P^{\mathcal{D}}(a,b) \geq \rho^s$ then return $\langle a, b, \mathcal{I}sa$, a new activity$\rangle$;
b10   else return $\langle a, b, \mathcal{P}artOf$, a new activity$\rangle$;
b11   end if
b12 end if
```

[a] in any tuple $\langle m_1, m_2, M, p\rangle$ the procedure returns, m_1 and m_2 are the abstracted activity, p is the abstracting one, and M indicates the abstraction mode – $\mathcal{A}$ denotes the universe of all activities.

Fig. 6. Procedure `abstractActivities`

The procedure abstractActivities works in a pairwise fashion by repeatedly abstracting two activities m_1 and m_2, both taken from S, by means of a complex activity p. All such activities are identified with the help of the function getBestAbstraction that returns a tuple indicating, besides p, m_1 and m_2, the kind of abstraction relationship to be used, i.e., $\mathcal{P}artOf$ or $\mathcal{I}sa$. As a special case, procedure getBestAbstraction will return the tuple $\langle \varepsilon, \varepsilon, \varepsilon, \varepsilon \rangle$ if there is no pair of activities in S that can be suitably abstracted. In such a case the condition $p = \varepsilon$ will hold, thus causing the termination of the abstraction procedure. Otherwise, in the resulting tuple $\langle m_1, m_2, mode, p \rangle$, m_1 and m_2 denote the two activities to abstract, and p is the complex activity which will replace them both, while *mode* denotes which kind of abstraction must be stored in $\mathcal{D}$: aggregation, via the $\mathcal{P}artOf$ relationship, or specialization, via the $\mathcal{I}sa$ relationship.

Procedure getBestAbstraction, still shown in Figure 6, essentially relies on the matching measures defined in Section 5.1. In more detail, the procedure takes as input a set S of activities and an associated set E of control flow edges, along with an abstraction dictionary $\mathcal{D}$. If there is no *merge-safe* pair in S that receives a sufficient score (w.r.t. a threshold ρ), then getBestAbstraction returns the tuple $\langle \varepsilon, \varepsilon, \varepsilon, \varepsilon \rangle$ (Lines b2 and b5), simply meaning that no abstraction can be performed over the activities in S. Otherwise, the procedure computes a tuple whose elements, respectively, specify the two activities to be abstracted, the kind of abstraction relationship to be used (i.e., $\mathcal{P}artOf$ or $\mathcal{I}sa$), and the complex activity which will abstract both of them. As a matter of facts, the choice of the abstracting activity and of the abstraction mode is based again on the similarity values computed via $sim_P^{\mathcal{D}}$ and $sim_G^{\mathcal{D}}$. In principle, if either of these measures is above the threshold ρ^S, the two activities are deemed similar enough to be looked at as two variants of some activity that generalizes them both. In particular, if $sim_G > \rho^S$ such an activity already exists: that is $msg^{\mathcal{D}}(m_1, m_2)$, which is indeed returned in the resulting tuple (Line b6). Before considering the creation of a new activity for generalizing m_1 and m_2 (Line b9), we check whether one of them implies the other: in such a case the implied activity can be abstracted by the other via an aggregation relationship (Lines b7-b8); we can, indeed, exclude that the implied activity is a specialization of the other, since such a condition was tested previously (Line b6). If none of the above cases applies, the two activities are eventually abstracted by a new activity via aggregation (Line b10).

As concerning the remainder of procedure abstractActivities, since either m_1 or m_2 might coincide with p, the set *ActuallyAbstracted* is used to keep trace of which of them should be really abstracted, for it actually being distinct from p (Line 4). Procedure deriveConstraints (see Line 20) is then applied to suitably derive the split and join conditions for p, based on those of the activities m_1 and m_2 that are being merged into it. Notice that, in principle, a looser join (resp., split) condition might be computed for p than those associated with m_1 and m_2, whenever these latter activities do not exactly match in their predecessor (resp., successor) nodes and in their join (resp., split) conditions. For space reasons, we skip here a detailed description of this procedure. In order to properly replace the abstracted activities, the control flow graph is properly settled by using

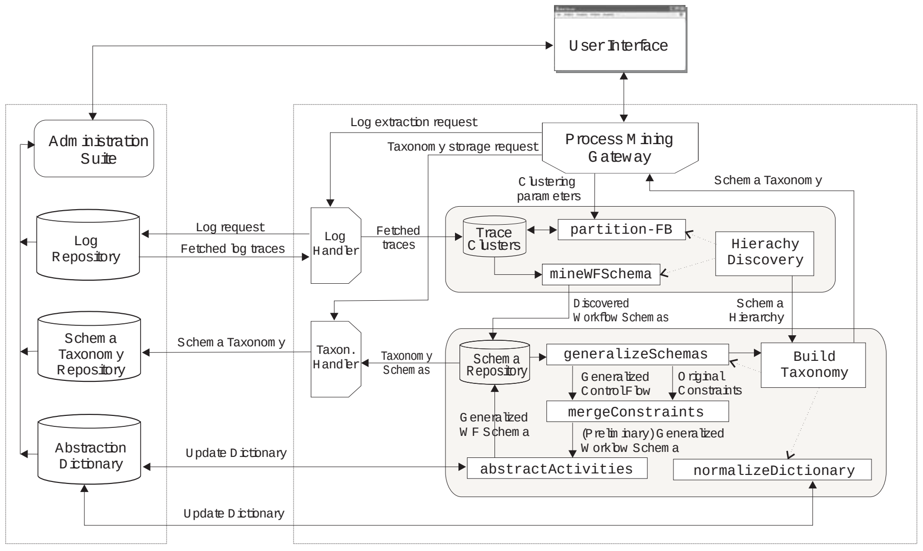


Fig. 7. System Architecture.

procedure `arrangeEdges`, which simply transfers the edges of the abstracted activities to p (Line 11). Finally, m_1 and m_2 are removed from both A and the reference set S (Lines 12-13), and a novel activity pair is searched for, in order to reiterate the whole abstraction procedure.

6 Discussion and Conclusions

We proposed a process mining approach that is meant to discover a hierarchical model representing the analyzed process through different views, at different abstraction levels. The approach consists of several mining and abstraction techniques, which are exploited in an integrated way. In particular, a preliminary schema hierarchy, accurately modelling the process at hand, is first discovered, by using a divisive clustering algorithm; the hierarchy is then restructured into a taxonomy, by equipping each non leaf node with an abstract schema that generalizes all the different schemas in the corresponding subtree.

The algorithms proposed in the paper have been implemented in JAVA and integrated into a stand-alone system architecture that is sketched in Fig. 7. For the sake of clarity and conciseness, major modules in the architecture are labelled with the names of the algorithms and procedures previously presented in the paper. Notably, different repositories are exploited to specifically manage the main kinds of information involved in the process mining task: log data, schema taxonomies, and abstraction relationships. Actually, a separate administration suite allows for effectively browsing and exploiting all such data. By the way, two further, "internal", repositories are used to maintain and share data on the trace clusters produced by the clustering algorithm and, respectively, the schemas generated during both the mining phase and the restructuring one. Currently, in order to offer the functionalities presented to a larger community of users, we are working at integrating the architecture into the *ProM* [18] process mining framework. At the time of writing, the hierarchical clustering module is already available as an additional, plug-in, component for *ProM*.

References

1. van der Aalst, W., Weijters, A., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering (TKDE) **16** (2004) 1128–1142
2. van der Aalst, W., van Dongen, B., Herbst, J., Maruster, L., G.Schimm, Weijters, A.: Workflow mining: A survey of issues and approaches. Data and Knowledge Engineering **47** (2003) 237–267
3. van der Aalst, W., Hirnschall, A., Verbeek, H.: An alternative way to analyze workflow graphs. In: Proc. 14th Int. Conf. on Advanced Information Systems Engineering. (2002) 534–552
4. van der Aalst, W., van Dongen, B.: Discovering workflow performance models from timed logs. In: Proc. Int. Conf. on Engineering and Deployment of Cooperative Information Systems (EDCIS 2002). (2002) 45–63
5. Agrawal, R., Gunopulos, D., Leymann, F.: Mining process models from workflow logs. In: Proc. 6th Int. Conf. on Extending Database Technology (EDBT'98). (1998) 469–483
6. Cook, J., Wolf, A.: Automating process discovery through event-data analysis. In: Proc. 17th Int. Conf. on Software Engineering (ICSE'95). (1995) 73–82
7. Muth, P., Weifenfels, J., M.Gillmann, Weikum, G.: Integrating light-weight workflow management systems within existing business environments. In: Proc. 15th IEEE Int. Conf. on Data Engineering (ICDE'99). (1999) 286–293
8. Greco, G., Guzzo, A., Pontieri, L., Saccà, D.: Mining expressive process models by clustering workflow traces. In: Proc. 8th Pacific-Asia Conference (PAKDD'04). (2004) 52–62
9. Castellanos, M., Casati, F., Dayal, U., Shan, M.C.: ibom: A platform for business operation management. In: Proc. Intl. Conf. on Data Engineering (ICDE05). (2005)
10. IDS Prof. Scheer, G.: (Aris-tool set. version 2.0 manual.) Saarbrcken 1994.
11. Malone, T.W., et al.: Tools for inventing organizations: Toward a handbook of organizational processes. Management Science **45** (1999) 425–443
12. Stumptner, M., Schrefl, M.: Behavior consistent refinement of object life cycles. ACM Transactions on Software Engineering and Methodology **11** (2002) 92–148
13. Stumptner, M., Schrefl, M.: Behavior consistent inheritance in uml. In: Proc. 19th Int. Conf. on Conceptual Modeling (ER 2000). (2000) 527–542
14. Basten, T., van der Aalst, W.: Inheritance of behavior. Journal of Logic and Algebraic Programming **47** (2001) 47–145
15. Lee, J., Wyner, G.M.: Defining specialization for dataflow diagrams. Information Systems **28** (2003) 651–671
16. Liu, D.R., Shen, M.: Workflow modeling for virtual processes: an order-preserving process-view approach. Information Systems **28** (2003) 505–532
17. Greco, G., Guzzo, A., Manco, G., Saccà, D.: Mining frequent instances on workflows. In: Proc. 7th Pacific-Asia Conference (PAKDD'03). (2003) 209–221
18. ProM: http://www.daimi.au.dk/PetriNets/tools/db/promframework.html.