Mining Taxonomies of Process Models [1]

Gianluigi Greco [a], Antonella Guzzo [c,*], Luigi Pontieri [b]

[a] *Dept. of Mathematics, University of Calabria, Via P.Bucci 30B, 87036 Rende, Italy*

[b] *Institute ICAR, CNR, Via P.Bucci 41C, 87036 Rende, Italy*

[c] *Dept. DEIS, University of Calabria, Via P.Bucci 41C, 87036 Rende, Italy*

Abstract

Process mining techniques have been receiving great attention in the literature for their ability to automatically support process (re)design. Typically, these techniques discover a concrete workflow schema modelling all possible execution patterns registered in a given log, which can be exploited subsequently to support further-coming enactments. In this paper, an approach to process mining is introduced that extends classical discovery mechanisms by means of an abstraction method aimed at producing a taxonomy of workflow models. The taxonomy is built to capture the process behavior at different levels of detail. Indeed, the most-detailed mined models, i.e., the leafs of the taxonomy, are meant to support the design of concrete workflows, as it happens with existing techniques in the literature. The other models, i.e., non-leaf nodes of the taxonomy, represent instead abstract views over the process behavior that can be used to support advanced monitoring and analysis tasks.

All the techniques discussed in the paper have been implemented, tested, and made available as a plugin for a popular process mining framework (*ProM*). A series of tests, performed on different synthesized and real datasets, evidenced the capability of the approach to characterize the behavior encoded in input logs in a precise and complete way, achieving compelling conformance results even in presence of complex behavior and noisy data. Moreover, encouraging results have been obtained in a real-life application scenario, where it is shown how the taxonomical view of the process can effectively support an explorative ex-post analysis, hinged on the different kinds of process execution discovered from the logs.

Key words: Process Mining, Abstraction, Knowledge Discovery, Workflow Management

1 Introduction

Designing and comprehending business processes are often difficult and time-consuming tasks, which typically require the intervention of a business consultant armed with an appropriate level of experience and knowledge on the process itself. In this context, process mining techniques have been getting increasing attention in recent years, for their ability to automatically extract useful knowledge about the actual behavior of a process, which can be a valid support to both analysis and design tasks. Indeed, these techniques are conceived to (automatically) discover a workflow-based model for a process, given some information about past executions, as for it is encoded in audit trails of workflow management systems as well as in transaction logs of various infrastructures for managing business processes (e.g., Enterprise Resource Planning, and Supply Chain Management systems).

Different approaches to process mining [2,9,19,20,32,43,46] have been proposed in the literature, which mainly differ in the language used for representing process models, and in the specific algorithm employed for their discovery (see [44] for a survey on this topic). In fact, the aim shared by the different proposals is to mine a model that can serve the purpose of automatically supporting further enactments of the process. Therefore, algorithms are commonly designed to maximize the accuracy of the mined model, i.e., they equip the model with as many details as they are required to explain the events registered in the logs. Consequently, when a large number of activities and complex behavioral patterns are involved in the analysis, traditional process mining techniques risk failing in representing the process in a clear and concise manner, and the resulting schema, while being well-suited for supporting the enactment, might be less useful for a business user who wants to monitor and analyze business operations at some lower level of detail.

In these scenarios, it is indeed desirable to deal with abstraction mechanisms offering the capability of discovering modular, yet expressive, descriptions for processes. 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 [7], ARIS [25]). Actually, these tools allow users to define the relationships among the real process and the abstract views in a manual way only. Hence, providing the user with some kinds of automatic tool capable of building a description of the process at different abstraction

* **Corresponding Author.** Phone: +39 0984 494618. Fax. +39 0984 839054
Email addresses: ggreco@mat.unical.it (Gianluigi Greco), guzzo@deis.unical.it (Antonella Guzzo), pontieri@icar.cnr.it (Luigi Pontieri).

[1] Portions of this paper appeared in a preliminary form in [18].

levels might represent an important capability for such platforms, in order to effectively reuse, customize, and semantically consolidate process knowledge.

A first step towards achieving a satisfactory trade-off between the accuracy and the comprehensibility of mined process models has recently been proposed in [19]. The idea is to recognize different "variants" (i.e., different usage scenarios) of the process by means of a structural clustering algorithm, and then equip each of these variants with a specific workflow schema. As a result, instead of a single, possibly intricate and complex workflow model for the whole process, a collection of more compact and easier to understand workflows can be discovered. Yet, the approach does not offer any support for restructuring the knowledge encoded in the various usage scenarios into in a taxonomy of process models that capture the behavior of the process at hand, by providing different views at different levels of details.

1.1 Overview of the Approach and Contribution

By continuing along the line of research paved in [19], the main aim of the paper is to combine the core idea of clustering process executions with ad-hoc abstraction techniques that produce a compact and handy representation for each high-level schema, by emphasizing the most relevant behavioral features while abstracting from specific details. The result is a novel process mining approach capable of building a *taxonomy* of process models. In a nutshell, the taxonomy is modelled as a tree of workflow schemas, where 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 this abstract model, in which some specific details are introduced. In other words, leaf nodes stand for concrete usage scenarios (computed through the clustering), whereas each non-leaf node (computed through abstraction mechanisms) is meant to provide a unified representation for all the process models associated with its children. Technically, the combination of process mining and abstraction methods, which is the main distinguishing feature of our approach, is carried out in two phases:

- In the first phase, we use a generalization of the top-down clustering algorithm presented in [19] to hierarchically decompose the process model into a number of sub-processes. Since, at each step, the algorithm splits a cluster whose associated schema is expected to mix different usage scenarios, those clusters over which no further splitting is applied effectively model different concrete variants for the process and can support further-coming enactments. Note that, because of the application of a hierarchical clustering, the result of the algorithm is a tree-like schema whose nodes correspond to set of executions and where, for each node, its children correspond to the splits originated by that executions. In fact, while the algorithm in [19] ex-

ploits a peculiar approach to equip each leaf with a workflow schema, the generalization discussed in this paper accommodates the use of any arbitrary process mining technique.
- In the second phase, we navigate bottom-up the tree, i.e., from the leaves to the root, in order to build a taxonomy. In particular, we equip each non-leaf node with an abstract schema that generalizes all the schemas associated with its children. To this aim, an abstraction method is defined which supports the generalization by properly replacing groups of "specific" activities with higher-level ones.

Both above phases have been implemented in a prototype system. Even though the resulting tool could be a valuable add-on for any advanced process management platform, we decided to implement it as a plugin for the process mining framework *ProM*[47], which is a well-established platform for developing and testing novel process mining algorithms. The system (in the respect of both above phases) has thoroughly been tested on syntectic and real datasets. In particular:

- As for the first phase, an experimental analysis has been conducted to compare the quality of the schemas induced by the clusters with those that can be obtained with other process mining techniques without clustering the process executions. In these experiments, differently from [19], several process mining algorithms have also been used to equip with a model the various clusters.
- As for the second phase, a case study has been discussed where the salient features of the abstraction mechanisms have been stressed. In this respect, it is worthwhile noticing that the availability of some background knowledge about a given domain might be quite useful to improve the quality of the mined taxonomy, especially in order to find the appropriate level of details. Actually, in this case study, we have been experiencing that, even in absence of context-dependent background knowledge, the taxonomical representation of the process model effectively enables for an explorative analysis where users may navigate within the tree-like structure, focusing on those concrete execution scenarios that most attract their attention. Hence, independently of what the right level of abstraction is (which is indeed a difficult and application-depended issue), a major advantage of having higher-level abstract schemas is to facilitate such an interactive analysis of process executions. Investigating on how to incorporate context-dependent information within the algorithm for inducing process taxonomies is outside the scope of this paper, but yet constitutes an interesting avenue of further research.

1.2 Plan of the Paper

The rest of the paper is organized as follows. In Section 2, a few preliminaries on process models are discussed. The top-down clustering algorithm is illustrated in Section 3, while the abstraction techniques aimed at building the schema taxonomy are presented in Section 4. Section 5 then provides an overview of the system that was integrated in the *ProM* framework. The practical application of the approach to different log data is discussed in Section 6, where several experimental results are reported to assess the effectiveness of both the hierarchical clustering and the abstraction mechanisms. In Section 7 we briefly survey some works recently appeared in the literature which are connected with our research, yet catching the opportunity to emphasize the main distinguishing points in our approach. A few concluding remarks are drawn in Section 8, which also depicts some directions of future work.

2 Process Models and Process Logs

A significant amount of research has been done in the context of specification mechanisms for process modelling (see, e.g., [48,42,41,15,36,45]). Process models are aimed at representing all possible execution flows along the activities of a given process, by means of a set of constraints defining "legal" execution in terms of simple relationships of precedence and/or more elaborate constructs such as loops, parallelism, synchronization and choice (just to cite a few).

The whole methodology discussed in this paper is completely orthogonal to the specific model adopted to represent processes (in fact, experiments will be discussed showing applications on a few relevant process models). Thus, to our aims, it suffices to assume that a set of activity identifiers, say A, is given and that a modelling language, say $\mathcal{M}$, (e.g., *Heuristics Nets* [48], *event driven process chains* [42], *WF-nets* [41], or others) is adopted. Then, we shall denote by $\mathcal{M}(A)$ the set of *workflow schemas* that can be built by using the constructs in $\mathcal{M}$ over the activities in A.

Example 1 Figure 1 shows a possible workflow schema for the *OrderManagement* process of handling customers' orders in a business company, which refers to a simple and intuitive modelling language where precedence relationships are depicted as arrows between nodes in a *control flow* graph, and where some constructs drawn by means of labels beside the tasks (cf. *and*, *or*, *xor*) are used to state more elaborate constraints of execution.

Roughly speaking, a node whose input is an "and" acts as synchronizer (i.e., it can be executed only after all its predecessors have been completed), whereas a node whose input is an "or" can start as soon as at least one of its predecessors

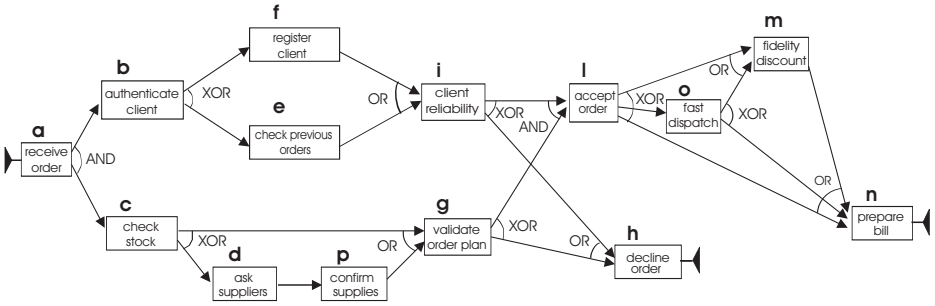


Fig. 1. Workflow schema for the sample *OrderManagement* process.

has been completed. Furthermore, once finished, a node whose output is an "and" (resp., "or", "xor") activates all (resp., some of, exactly one of) its outgoing activities. ◁

Each time a workflow schema $W \in \mathcal{M}(A)$ is enacted in a workflow management system, the activities in A are executed according to the constraints of W, till some final configuration is reached for which there is no constraint in W forcing the execution of some further activity. As an example, we can refer again to the schema of Figure 1, and notice that the sequence `acbgih` might be the result of an enactment. In fact, the sequencing of the events related to these various executions is stored by the system over each enactment, and will constitute the input for the *process mining* problem [46,44,43,2,9,32,19].

More formally, given a workflow schema W over a modelling language $\mathcal{M}$ and over the activities in A, we shall denote by $\mathcal{L}(W)$ the set of all the traces that correspond to the enactments satisfying the various constraints in W. Then, the input of the process mining problem is a *log* $\mathcal{L} \subseteq \mathcal{L}(W)$, i.e., a bag of traces that are legal according to the model W. Based on $\mathcal{L}$, the goal is to discover a workflow schema, say W', such that $\mathcal{L} = \mathcal{L}(W')$, i.e., a schema that is capable of explaining exactly all the episodes in $\mathcal{L}$. The rationale of the approach is that for a sufficiently large input log $\mathcal{L}$ (ideally $\mathcal{L} = \mathcal{L}(W)$), the schema W' would behave in practice as W.

Clearly enough, in real applicative scenarios, we have little chance of discovering such an ideal W', either because the input log $\mathcal{L}$ is *not complete*, i.e., $\mathcal{L} \subset \mathcal{L}(W)$, or because it is *not sound*, i.e., $\mathcal{L}$ contains some trace that is not in $\mathcal{L}(W)$ due to noise and/or malfunctioning occurred in some enactments. Thus, several measures of qualities for the discovered workflow schema W' can be used to asses how much W' is close to W.

Again, we note that adopting some specific measure is an orthogonal issue w.r.t. our approach (and, in fact, different measures have experimentally been tested). For the sake of exposition, we hence shall assume the availability of a template function *conformance* that on input W' and $\mathcal{L}$ reports a real number in [0..1] estimating how much W' is good to model the logs in $\mathcal{L}$ (the higher is the score, the better is the model). Then, process mining algorithms

are naturally aimed at deriving a schema W' such that *conformance*$(W', \mathcal{L})$ is maximized for some conformance function and on any input log $\mathcal{L}$. In the following section, we shall discuss how this problem can be generalized to cope with taxonomies of process models.

3 Hierarchically Mining Workflow Schemas

As outlined in the Introduction, our method for discovering expressive process models at different levels of detail is articulated in two phases. First, we exploit a hierarchical top-down clustering algorithm, called `HierarchyDiscovery`. Then, we visit the mined model in a bottom-up way, and we restructure it at several levels of abstraction, by means of the algorithm `BuildTaxonomy`.

In this section, we provide details on the former phase. More precisely, after formally defining the notion of *schema decomposition*, we describe the algorithm `HierarchyDiscovery` and present an application example.

3.1 Algorithm `HierarchyDiscovery`

A process mining technique that is specifically tailored for complex processes, involving lots of activities and exhibiting different variants was presented in [19], which relies on the idea of explicitly representing all the possible usage scenarios by means of a collection of different, specific, workflow schemas. Here, we propose a new algorithm that extends the one presented in [19], and where the mined model is meant to represent the process at different levels of granularity. Indeed, the algorithm will compute a tree-like decomposition of the process where each node corresponds to a schema, and where the set of schemas associated with the children of a node v models the behavior encoded in the same set of executions supported by v, but in a more detailed way, as different subclasses of executions are described separately. To this aim, it is convenient to introduce the following notion of schema decomposition.

Definition 2 (Schema Decomposition) Let A be a set of activities, and let $\mathcal{M}$ be a modelling language. Then, a *schema decomposition (over $\mathcal{M}$ and A)* is a tuple $\mathcal{H} = \langle \mathcal{WS}, T, \lambda \rangle$, such that:

- $\mathcal{WS} \subseteq \mathcal{M}(A)$ is a set of workflow schemas;
- $T = \langle V, E, v_0 \rangle$ 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 $\mathcal{WS}$. □

Input: A modelling language $\mathcal{M}$, a set of log traces $\mathcal{L}$ over activities in A, two natural numbers $maxSize$ and k, a threshold γ.
Output: A schema decomposition $\langle \mathcal{WS}, T, \lambda \rangle$ over $\mathcal{M}$ and A.
Method: Perform the following steps:

1 W_0 :=`mineWFschema`($\mathcal{L}$);
2 $\mathcal{WS} := \{W_0\}$;
3 $Traces[W_0] := \mathcal{L}$; *// $Traces[W_i]$ refers to the log traces modelled by W_i, $\forall W_i \in \mathcal{WS}$*
4 $V := \{v_0\}$; $E := \emptyset$; $T := \langle V, E, v_0 \rangle$;
5 $\lambda(v_0) := W_0$;
6 **while** $|\mathcal{WS}| \leq maxSize$ and $conformance(\langle \mathcal{WS}, T, \lambda \rangle, \mathcal{L}) < \gamma$ **do**
7 **let** W_q be the least conforming "leaf" schema [a] and v_q=$\lambda^{-1}(W_q)$ be its associated node in T;
8 **let** n=$|\mathcal{WS}|$ be the number of schemas currently stored in $\mathcal{WS}$;
9 $\langle L_{n+1}, ..., L_{n+k} \rangle$:= `partition-FB`($Traces[W_q]$);
10 **if** $k > 1$ **then**
11 **for** $h = 1..k$ **do**
12 W_{n+h} := `mineWFschema`(L_{n+h});
13 $\mathcal{WS} := \mathcal{WS} \cup \{W_{n+h}\}$;
14 $Traces[W_{n+h}] := L_{n+h}$;
15 $V := V \cup \{v_{n+h}\}$; $E := E \cup \{(v_q, v_{n+h})\}$;
16 $\lambda(v_{n+h}) := W_{n+h}$;
17 **end for**
18 **end if**
19 **end while**
20 **return** $\langle \mathcal{WS}, T, \lambda \rangle$;

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

Fig. 2. **Algorithm** `HierarchyDiscovery`

The input for the process mining problem is, as usual, a log $\mathcal{L}$. Based on $\mathcal{L}$, our aim is to build a schema decomposition, where 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 more specific way. This is carried out by recursively partitioning the traces in $\mathcal{L}$ into clusters, according to the different behavioral patterns they exhibit, and by building a schema for each of these clusters.

An algorithm, named `HierarchyDiscovery`, implementing this approach is reported in Figure 2, where the function `mineWFschema` is exploited for associating a single workflow schema in $\mathcal{W}(A)$ with each cluster of the hierarchy—one may use any standard process mining algorithm here. The algorithm starts by building a workflow schema W_0 (Line 1) that is the first attempt to represent the behavior captured in the log traces, and that will be the only component of $\mathcal{WS}$ (Line 2). The schema W_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).

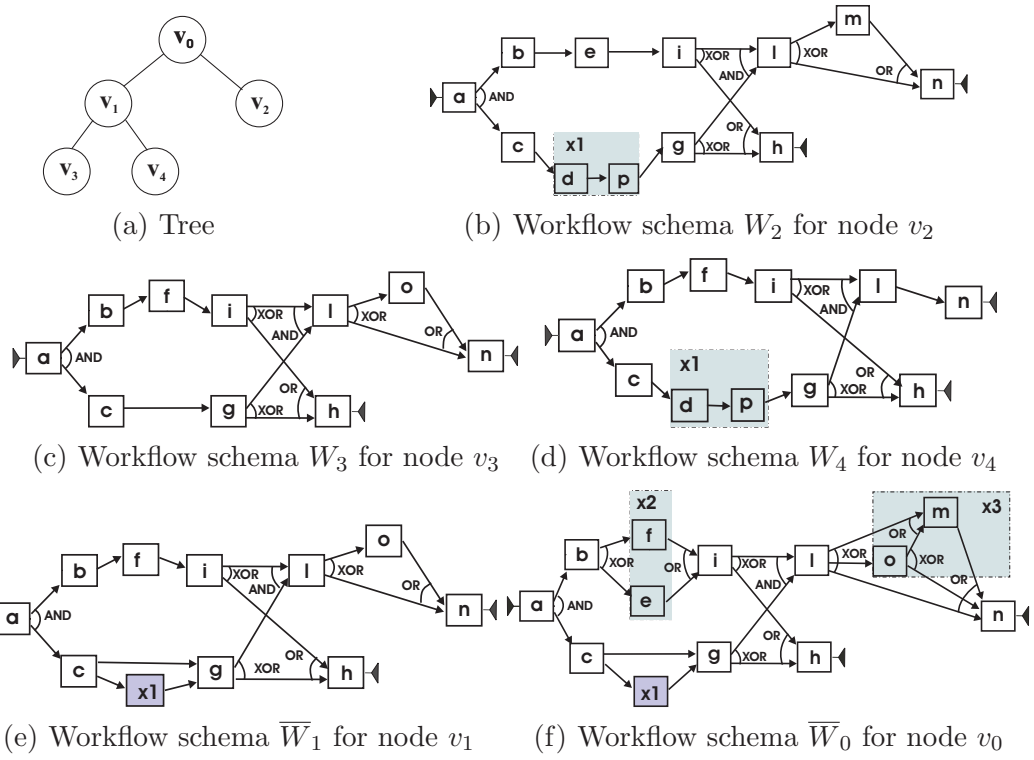
In order to get a more accurate model, the refinement is carried out with the aim of reducing as much as possible the number of spurious executions supported by the model that do not correspond to any of the instances registered in the input log $\mathcal{L}$. Indeed, algorithm `HierarchyDiscovery` is designed to greedily select to refine the "least conforming" schema W_q in $\mathcal{WS}$ (i.e., the one that gets the lowest score by the chosen *conformance* function) and to derive a set of more refined schemas (Lines 7-18) as children of the node corresponding to W_q. Therefore, the set of traces modelled by the selected schema W_q is partitioned through the procedure `partition-FB` (Line 9) into a set of k clusters which, in a sense, are more homogeneous from a behavioral viewpoint.

Note that, different clustering algorithms could be used to partition the traces associated with the selected schema W_q. In particular, like in [3,4,21], we can exploit the discovery of frequent patterns (see, e.g., [28,31,33]), in order to map the traces into a feature space, where classical clustering methods can be eventually applied. From this exposition, we omit details on the specific way `partition-FB` is implemented, and we address the reader to, e.g.,[19], where a special kind of sequences of frequent activities is used to define the feature space, which are capable of capturing behaviors that are unexpected w.r.t. the current workflow schema (i.e., W_q, in Figure 2), and which can efficiently be discovered via a level-wise search strategy.

Once that traces associated with W_q have been partitioned, for each of the new clusters so found, say L_{n+h}, a specific workflow schema W_{n+h} is discovered (again with function `mineWFschema`), and added to $\mathcal{WS}$ (Lines 10-11). The algorithm keeps trace of the connection between the schema W_{n+h} and the trace cluster, L_{n+h}, it has been discovered from (Line 14). Moreover, W_{n+h} is associated with a new node v_{n+h}, which is added to the tree T as a child of the node v_q that corresponds to the schema W_q being just refined (Lines 15-16).

The whole process of refining a schema can then be iterated in a recursive way, by selecting again the least conforming leaf schema in the current hierarchy. The operations are in fact repeated until the number of schemas stored in $\mathcal{WS}$ is greater than the input parameter $maxSize$ or the global conformance, denoted by $conformance(\langle \mathcal{WS}, T, \lambda \rangle, \mathcal{L})$ and measured as the minimum conformance value over the schemas associated with the leaves of T that have some corresponding trace in $\mathcal{L}$, exceeds a threshold γ.

We conclude this section by noticing that, by a direct application of computational results in [19], the `HierarchyDiscovery` algorithm can be implemented to perform $O(maxSize \times k \times p)$ scans of the input log, where p is the number of scans of $\mathcal{L}$ required by each call to function `partition-FB` in algorithm `HierarchyDiscovery`. For instance, in [19], `partition-FB` requires a number of scans which is bounded by the length of the task sequences used as features.



(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

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

Fig. 3. Process taxonomy for the *OrderManagement* process.

3.2 An Example Scenario

In this section we illustrate the application of the mining algorithm to a simple scenario, which refers to the workflow schema shown in Figure 1. Based on this schema, we randomly generated 100,000 traces compliant with it by using the generator described in [17]—notice that more traces actually encode the same task sequence. In this generation, we also required that task m could not occur in any execution containing f, and that task o could not appear in any trace containing d and p, thereby modelling the restriction that a fidelity discount is 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 decomposition reported in Figure 3. More specifically, Figure 3.(a) sketches the tree-like structure of the decomposition, where each node logically corresponds to both a cluster of traces and a workflow schema induced from that cluster, by means of traditional algorithms for process mining.

The workflow schemas eventually extracted for the leaves of this tree are shown

in the Figures 3.(b), 3.(c), and 3.(d) [2]. In particular, 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 "good" as required by the user, and therefore partitions the traces by means of a clustering algorithm (k-means in our 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 conformance), 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 facts, 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 better way by the leaf schemas. Nevertheless, the whole decomposition is an important result as well, for it structures the discovered execution classes and for it forms the basis for deriving a schema taxonomy representing the process at different abstraction levels (cf. definition of $\overline{W}_0$ and $\overline{W}_1$).

4 Building Schema Taxonomies

The second phase of our approach is devoted to exploit the schema decomposition produced by `HierarchyDiscovery` in order to derive a taxonomy of workflow schemas that collectively represent the process at hand at several abstraction levels. In this section, after formally defining the concept of schema taxonomy, we overview the main algorithm, called `BuildTaxonomy`, that restructures each non-leaf schema in the input hierarchy by making it a generalization over all its children. The general idea of the algorithm is discussed in Section 4.1, while some important details will be illustrated in Section 4.2 and Section 4.3. An exemplification on our *OrderManagement* process will finally be discussed in Section 4.4.

4.1 Algorithm `BuildTaxonomy`

Based on a schema decomposition, we next consider the problem of consolidating the knowledge about a process behavior into different levels of detail. To this purpose, we first introduce some basic abstraction relationships over activities, by assuming they are gathered in a suitable repository, called abstraction dictionary, which is defined as follows.

Definition 3 (Abstraction Dictionary) An abstraction dictionary is a tu-

[2] Please ignore, for now, the label x_1 and the associated highlighted box (in the Figures 3.(b) and 3.(d)), which will be discussed in the following.

ple $\mathcal{D} = \langle \mathcal{A}, \mathcal{I}sa, \mathcal{P}artOf \rangle$, where $\mathcal{A}$ is a set *activities*. $\mathcal{I}sa \subseteq \mathcal{A} \times \mathcal{A}$, $\mathcal{P}artOf \subseteq \mathcal{A} \times \mathcal{A}$ and, for each $a \in \mathcal{A}$, $(a, a) \notin \mathcal{P}artOf$ and $(a, a) \notin \mathcal{I}sa$. □

Intuitively, for activities a and b, $(b, a) \in \mathcal{I}sa$ indicates that b is a refinement of a; conversely, $(b, a) \in \mathcal{P}artOf$ indicates that b is a component of a. Based on the content of the abstraction dictionary, we can define some more general relationships that are meant to capture scenarios where an activity is an abstract version of another one.

Definition 4 (Implied Activities) Given an abstraction dictionary $\mathcal{D} = \langle \mathcal{A}, \mathcal{I}sa, \mathcal{P}artOf \rangle$, and two activities a and a', we say that a *implies* a' w.r.t. $\mathcal{D}$, denoted by $a \longrightarrow^{\mathcal{D}} a'$, if $a, a' \in \mathcal{A}$ and either

- $(a', a) \in \mathcal{I}sa$ or $(a', a) \in \mathcal{P}artOf$, or
- there exists an activity $x \in \mathcal{A}$ such that $a \longrightarrow^{\mathcal{D}} x$ and $x \longrightarrow^{\mathcal{D}} a'$.

An activity a is 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. Finally, the set of all basic activities implied by a w.r.t. $\mathcal{D}$ is denoted by $impl^{\mathcal{D}}(a)$, i.e., $impl^{\mathcal{D}}(a) = \{a' \mid a \longrightarrow^{\mathcal{D}} a' \text{ and } \neg\exists a'' \text{ s.t. } a' \longrightarrow^{\mathcal{D}} a''\}$. □

Notice that complex activities represent high-level concepts defined by aggregating or generalizing basics activities actually occurring in real process executions. This notion is the 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. Before providing a formal definition of taxonomical process models, we next illustrate the simple underlying notion of generalization between workflow schemas.

Definition 5 (Schema Generalization) 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 $\mathcal{W}_2 \preceq^{\mathcal{D}} \mathcal{W}_1$, if for each activity a_2 in A_2 *(i)* either $a_2 \in A_1$ or 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 $a_2 \longrightarrow^{\mathcal{D}} b_1$. □

Example 6 Consider the workflow schema W_4 in Figure 3.(d), and the schema $\overline{W}_1$ in Figure 3.(e). Assume that the pairs (d, x_1) and (p, x_1) already belong to the $\mathcal{P}artOf$ relationship of a given abstraction dictionary—the containment of d and p in x_1 is emphasized in a graphical way in both those figures. Then, $\overline{W}_1$ generalizes W_4. Also, it trivially holds that $\overline{W}_1$ generalizes the schema W_3 depicted in Figure 3.(c). ◁

We are now in position to formalize the concept of schema taxonomy, which is the output of the second phase of our approach.

Definition 7 (Schema Taxonomy) Let $\mathcal{D}$ be an abstraction dictionary and

Input: A schema decomposition $\mathcal{H} = \langle \mathcal{WS}, T, \lambda \rangle$ over a set of activities A;
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** $\mathcal{D} := \langle A, \emptyset, \emptyset \rangle$;
2 $Done := \{ v \in V \mid \nexists v' \in V \text{ s.t. } (v, v') \in E \}$; // *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** $ChildSchemas = \{ \lambda(c) \mid v \in V \text{ and } (v, c) \in E \}$, i.e., the schemas of all v's children;
5 $\lambda'(v) :=$ `generalizeSchemas`$(ChildSchemas, \mathcal{D})$;
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`($\mathcal{WS} = \{W_1, ..., W_k\}$: set of workflow schemas,
var $\mathcal{D}$: abstraction dictionary): workflow schema;

g1 **let** A_h be the activities occurring in the schema W_h;
g2 let $\overline{W}$ be a schema over $\overline{A} = \bigcup_{h=1}^{k} A_h$ admitting all traces in W_h, $1 \leq h \leq k$;
g3 **for each** $h = 1..k$ **do**
g4 `abstractActivities`$(A_h, \overline{W}, \mathcal{D})$;
g5 `abstractActivities`$(\overline{A}, \overline{W}, \mathcal{D})$;
g6 **return** $\overline{W}$;

Fig. 4. **Algorithm** `BuildTaxonomy`

let $\mathcal{H} = \langle \mathcal{WS}, T = (V, E), \lambda \rangle$ be a schema decomposition. Then, $\mathcal{G}$ is a *schema taxonomy* w.r.t. $\mathcal{D}$ if for any pair of nodes v and v_c in V such that $(v, v_c) \in E$ (i.e., v_c is a child of v), it holds: $\lambda(v) \preceq^{\mathcal{D}} \lambda(v_c)$. □

Note that schema taxonomies found on a notion of generalization that refers to basic abstraction relationships between activities and disregards the ordering of activities and other routing constraints they are involved in. Indeed, these taxonomies are primarily devoted to provide a multi-layered description of the process, where only the leaf schemas are meant as precise models for distinct classes of execution. By contrast, any high-level (i.e., non-leaf) schema essentially offers a compact and possibly approximated description over heterogenous behavioral classes, which can eventually support high-level explorative analysis of the process. We can hence admit a loss in precision in the representation of these latter schemas. Therefore, our perspective is completely orthogonal w.r.t. other proposals where inheritance notions are defined to take into account behavioral features (see, e.g., the notion of Most Specific Generalization introduced in [6], for the case of Petri nets).

Armed with these notions, we can now discuss the `BuildTaxonomy` algorithm, which is reported in Figure 4. The algorithm takes in input a schema decomposition $\mathcal{H}$ (computed by `HierarchyDiscovery`) and produces a taxonomy $\mathcal{G}$ and an abstraction dictionary $\mathcal{D}$, which $\mathcal{G}$ has been built according to.

The basic task in the generalization consists of replacing groups of "specific" activities appearing in the schemas to be generalized, with new "virtual" (i.e., complex) activities representing them at a higher level of abstraction. By this way, a more compact description of the process is obtained, yet providing that the abstraction dictionary $\mathcal{D}$ is updated to maintain the relationships between the activities that were abstracted and the new higher-level concepts replacing them. This dictionary is simply initialized in a way that it just contains all the (basic) activities appearing in any schema of $\mathcal{H}$, while both relations $\mathcal{I}sa$ and $\mathcal{P}artOf$ are empty (Line 1). In general, however, one could think of exploiting abstraction relationships that are already available (as a form of background knowledge), by suitably encoding them into it.

The algorithm works in a bottom-up fashion (Lines 2-7): starting from the leaves of the input decomposition, it produces, for each non-leaf node v, a novel workflow schema that generalizes all the schemas associated with the children of v. Notably, this 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. This generalization is carried out by providing the procedure `generalizeSchemas` with the schemas associated with the children of v, along with the abstraction dictionary $\mathcal{D}$ (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. Note that this normalization step is just a heuristic post-processing, which is not crucial for ensuring the correctness of the approach.

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 Figure 4 as well) first merges all the input workflow schemas into a preliminary workflow schema $\overline{W}$ (Line g2), which represents all the possible flow links in the input workflows—basically, this procedures can be seen as performing the union of the control flow graphs corresponding to each W_h. Subsequently, activities are abstracted by 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 $\mathcal{D}$. In particular, `abstractActivities` is first applied for merging only those activities that are derived from the same input schema—at step h, the activities coming from the h-th schema can only be merged (Line g4). A further application is then performed to abstract non-shared activities in the current schema, independently of their origin (line g5).

Details on this procedure are reported in the following. Here, we just note that `abstractActivities` will be such that for every activity a occurring in some schema of $\mathcal{WS}$, either (i) a is kept in $\overline{W}$, or (ii) a is abstracted into some chain of high-level activities, the last of which appears in $\overline{W}$. Therefore, we will immediately be guaranteed that the output returned by algorithm `BuildTaxonomy` complies with Definition 5.

4.2 Matching Activities for Abstraction Purposes

In order to discuss the implementation of `abstractActivities`, we first introduce some similarity functions that we exploit for singling out those activities that can safely be abstracted into higher-level ones. In particular, we consider two kinds of function, one devoted to capture semantical affinities between activities, on the basis of a given abstraction dictionary $\mathcal{D}$, and another devoted to compare activities from a topological viewpoint. To introduce similarity functions of the former kind, we preliminary need some further concepts related to abstraction dictionaries.

Given two activities a and a', we say that a *generalizes* a' w.r.t. a given abstraction dictionary $\mathcal{D} = \langle \mathcal{A}, \mathcal{I}sa, \mathcal{P}artOf \rangle$, denoted by $a \uparrow^{\mathcal{D}} a'$, if there is a sequence of activities $a_0, a_1, .., a_n$ from $\mathcal{A}$ such that $a_0{=}a'$, $a_n{=}a$ and $(a_i, a_{i-1}) \in \mathcal{I}sa$ for each $i = 1..n$; we call this 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. As a special case, we let $dist_G^{\mathcal{D}}(a, a) = 0$, for any activity a.

The notions of *genpath* and $dist_G^{\mathcal{D}}$ introduced above are exploited in the following in order to define the *most specific generalization* of a pair of activities, which roughly represents the common ancestor in the hierarchies induced by $\mathcal{I}sa$ links that is maximally close to both of them.

Definition 8 (Most Specific Generalization) Let x and y be two activities, and $ancestors_G^{\mathcal{D}}(x, y)$ be the set of all the activities that generalize both x and y, i.e., $ancestors_G^{\mathcal{D}}(x, y) = \{z \in \mathcal{A} \mid z \uparrow^{\mathcal{D}} x \text{ and } z \uparrow^{\mathcal{D}} y\}$. The *most specific*

generalization of w.r.t. $\mathcal{D}$, denoted by $msg^{\mathcal{D}}(x, y)$, is defined as follows:

$$msg^{\mathcal{D}}(x,y) = \begin{cases} \varepsilon, & \text{if } ancestors^{\mathcal{D}}_G(x,y) = \emptyset \\ \text{an activity } z \in ancestors^{\mathcal{D}}_G(x,y) \text{ s.t. } \forall z' \in ancestors^{\mathcal{D}}_G(x,y) & \\ dist^{\mathcal{D}}_G(x,z') + dist^{\mathcal{D}}_G(y,z') \geq dist^{\mathcal{D}}_G(x,z) + dist^{\mathcal{D}}_G(y,z), & \text{otherwise} \end{cases}$$

□

In words, the most specific generalization of x and y is an activity, if there exists one, that generalizes both x and y, and is maximally close to both of them them according to the function $dist^{\mathcal{D}}_G$. Since this definition may allow for multiple candidates, in this case we hereinafter assume that the function $msg^{\mathcal{D}}(x, y)$ just returns one of them, chosen at random.

We can now define some dictionary-based similarities.

Definition 9 (Dictionary-based similarities) Let x and y be two activities, and $\mathcal{D} = \langle \mathcal{A}, \mathcal{I}sa, \mathcal{P}artOf \rangle$ be an abstraction dictionary. Then the *implication-oriented similarity* between x and y, denoted by $sim^{\mathcal{D}}(x, y)$, is:

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

while the *generalization-oriented similarity* between x and y, denoted by $sim^{\mathcal{D}}_G(x, y)$, is:

$$sim^{\mathcal{D}}_G(x,y) = \begin{cases} 0, & \text{if } msg^{\mathcal{D}}(x,y) = \varepsilon \\ 1 - \frac{dist^{\mathcal{D}}_G(x, msg^{\mathcal{D}}(x,y)) + dist^{\mathcal{D}}_G(y, msg^{\mathcal{D}}(x,y))}{|\mathcal{A}|}, & \text{otherwise} \end{cases}$$

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

Intuitively, for any two activities a and b, $sim^{\mathcal{D}}(a, b)$ evaluates how many sub-activities a and b actually share, by comparing their implied activities (see Definition 4), extended with a and b, respectively. Conversely, function $sim^{\mathcal{D}}_G$ takes into account the generalization relationships that derive from $\mathcal{D}$, and evaluates the semantical similarity of two activities, by measuring their distance to their most specific generalization (i.e., their closest common ancestor in the hierarchy induced by $\mathcal{I}sa$ links, see Definition 8). Hence, the closer is the most specific generalization of x and y to both of them, the more similar to each other the two activities are deemed by function $sim^{\mathcal{D}}_G$.

We now turn to describe a function aimed at comparing activities from a topological viewpoint. In the following, let E be a directed binary relation encoding the set of precedence constraints in some given workflow schema $\mathcal{W}$ (elements in E can roughly be seen as edges in the control flow graph). The

basic idea is to consider similar two given activities, if they can be "merged" by limiting the creation of spurious control flow paths among the remaining activities in the workflow schema. 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 10 (Merge-Safety) Given a binary relation E, we say that a (unordered) pair of activities (x, y) is *merge-safe* if either

(a) $\{(x, y), (y, x)\} \cap E \neq \emptyset$ and $\{(x, y), (y, x)\} \cap (E - \{(x, y), (y, x)\})^* = \emptyset$—intuitively, x and y are directly linked by some edges in E and after removing these edges no other path exists connecting x and y; or,
(b) $\{(x, y), (y, x)\} \cap E^* = \emptyset$—intuitively, there is no path connecting x and y,

where, for any set $F \subseteq E$, F^* denotes the transitive closure of F. □

Notably, only in the case (b) of Definition 10 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 $\{(z, y), (x, w)\} \subseteq E$. A straightforward way to prevent this problem consists in requiring that at least one of the following conditions holds:

- $\mathcal{P}_x^E = \mathcal{P}_y^E$,
- $\mathcal{S}_x^E = \mathcal{S}_y^E$,
- $\mathcal{P}_x^E \subseteq \mathcal{P}_y^E$ and $\mathcal{S}_x^E \subseteq \mathcal{S}_y^E$,
- $\mathcal{P}_y^E \subseteq \mathcal{P}_x^E$ and $\mathcal{S}_y^E \subseteq \mathcal{S}_x^E$.

where $\mathcal{P}_a^E$ (resp. $\mathcal{S}_a^E$) denotes the set of predecessors (resp. successors) of activity a, according to the arcs in E.

The function $sim^E(x, y)$, reported below, incorporates these measure in a smoothed way, with the aim of evaluating the similarity of a pair of activities according to the number of spurious flows that would be generated when merging them (the more spurious flows are introduced, the lower is the score):

Definition 11 (Topological similarity) Let x and y be two activities, and E be a set of activity pairs, encoding control flow (precedence) relationships.

$$sim^E(x, y) = \begin{cases} 0, \text{ if } (x, y) \text{ is not a merge-safe pair of activities} \\ \frac{\alpha(\mathcal{P}_x^E, \mathcal{P}_y^E) \cdot \alpha(\mathcal{S}_x^E, \mathcal{S}_y^E) + \beta(\mathcal{P}_x^E, \mathcal{P}_y^E) \cdot \beta(\mathcal{S}_x^E, \mathcal{S}_y^E)}{2}, \text{ otherwise} \end{cases}$$

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

Notice that sim^E produces a maximal value whenever one of the "strong" conditions discussed before holds, and, in general, tends to assign high values to activities matching over many of their predecessors (successors).

4.3 Procedure `abstractActivities`

We can now discuss the `abstractActivities` procedure in Figure 5. The procedure takes as input a workflow schema $\overline{W}$, a set of activities S that are involved in $\overline{W}$, and an abstraction dictionary $\mathcal{D}$. As a result, it transforms $\overline{W}$ by replacing the abstracted activities with the associated complex ones, and modifies $\mathcal{D}$ in order to record the performed abstraction transformations.

The procedure 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. These activities are identified by 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 $\mathcal{I}sa$. Since either m_1 or m_2 might coincide with p, the set *ActuallyAbstracted* is used to keep trace of which activities must actually be abstracted, in that they are really distinct from p (Line 4). Anyway, the dictionary is correctly updated to include the activity p (Line 5). The algorithm then calls the procedure `deriveConstraints` (see Line 10), which is responsible of deriving some local constraints on the basis of those activities m_1 and m_2 that are being merged into it. Note that when carrying out procedure `deriveConstraints`, problems may occur when two constraints are discovered over m_1 and m_2 that are conflicting, i.e., when they can not both hold at the same time. In our implementation, we decided to tolerate loss of accuracy in representing the actual behavior of the process when merging activities; thus, in presence of conflicts the most stringent constraint is removed. As an example, when *join* and *or* constraints are in conflict, their merging causes the removal of the *join* one. Finally, m_1 and m_2 are removed from both A and the reference set S (Lines 11-12), and a novel activity pair is searched for, in order to reiterate the whole abstraction procedure.

A crucial aspect in this approach pertains the way activities to be abstracted are identified. To this end, procedure `getBestAbstraction` takes as input a set S of activities and an associated set E, along with an abstraction dictionary $\mathcal{D}$ and exploits the similarity measures introduced in Section 4.2, which evaluate how much two activities are suitable to be merged into a single higher-level activity. We briefly recall here that, given two any activities a and b, the topological similarity $sim^E(a, b)$ (see Definition 11) compares them from a topological viewpoint according to a given set E, whereas the two other functions (see Definition 9), measure semantical affinities between a and b according to an abstraction dictionary $\mathcal{D}$. More precisely, the implication-oriented similarity $sim^{\mathcal{D}}(a, b)$ takes into account all the activities implied by a and b, while

Procedure `abstractActivities`(S: set of activities; var $\overline{W}$: a workflow schema;
var $\mathcal{D} = \langle \mathcal{A}, \mathcal{I}sa, \mathcal{P}artOf \rangle$: 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 $\mathcal{A} := \mathcal{A} \cup p$;
6 **if** $mode =$ `ISA` **then**
7 $\mathcal{I}sa := \mathcal{I}sa \cup \{(x, p)$ s.t. $x \in ActuallyAbstracted\}$;
8 **else**
9 $\mathcal{P}artOf := \mathcal{P}artOf \cup \{(x, p)$ s.t. $x \in ActuallyAbstracted\}$;
10 **end if**
11 `deriveConstraints`$(\bar{W}, p, m_1, m_2)$;
11 $A := A - ActuallyAbstracted \cup \{p\}$;
12 $S := S - ActuallyAbstracted \cup \{p\}$;
13 $\langle m_1, m_2, mode, p \rangle$:=`getBestAbstraction`$(S, E', \mathcal{D})$;
14 **end while**

Procedure `getBestAbstraction`(S: set of activities; E: set of activity pairs; $\mathcal{D}$: abstraction dictionary):
a tuple in $\langle \varepsilon, \varepsilon, \varepsilon, \varepsilon \rangle \cup S \times S \times \{\mathcal{I}sa, \mathcal{P}artOf\} \times \mathcal{U}$; [a]

b1 **let** $score(x, y) = max\{sim^E(x, y), sim^{\mathcal{D}}(x, y), sim^{\mathcal{D}}_G(x, y)\}$, for any activities x and y
b2 **if** $|S| < 2$ **then**
b3 **return** $\langle \varepsilon, \varepsilon, \varepsilon, \varepsilon \rangle$;
b4 **else**
b5 **let** a and b be two activities such that $score(a, b) = max\{score(x, y) \mid x, y \in S\}$;
b6 **if** $score(a, b) < \rho$ **then return** $\langle \varepsilon, \varepsilon, \varepsilon, \varepsilon \rangle$;
b7 **else if** $sim^{\mathcal{D}}_G(a, b) \geq \rho^s$ and $msg^{\mathcal{D}}(a, b) \neq \varepsilon$ **then return** $\langle a, b, \mathcal{I}sa, msg^{\mathcal{D}}(a, b) \rangle$;
b8 **else if** $\exists p \in \mathcal{D}.\mathcal{A}$ s.t. $\{(a, p), (b, p)\} \subseteq \mathcal{D}.\mathcal{P}artOf$ **then return** $\langle a, b, \mathcal{P}artOf, p \rangle$;
b9 **else if** $impl^{\mathcal{D}}(b) \subseteq impl^{\mathcal{D}}(a)$ **then return** $\langle a, b, \mathcal{P}artOf, a \rangle$;
b10 **else if** $impl^{\mathcal{D}}(a) \subseteq impl^{\mathcal{D}}(b)$ **then return** $\langle a, b, \mathcal{P}artOf, b \rangle$;
b11 **else if** $sim^{\mathcal{D}}_P(a, b) \geq \rho^s$ and $\{(a, x), (b, x)\} \cup \mathcal{I}sa = \emptyset$ **then return** $\langle a, b, \mathcal{I}sa,$ a new activity$\rangle$;
b12 **else return** $\langle a, b, \mathcal{P}artOf,$ a new activity$\rangle$;
b13 **end if**
b14 **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 – $\mathcal{U}$ denotes the universe of all activities.

Fig. 5. **Procedure** `abstractActivities`

the generalization-oriented similarity $sim^{\mathcal{D}}_G(a, b)$ evaluates how much close to both activities is their common ancestor in the $\mathcal{I}sa$-based hierarchy, if there is any. Based on all of these functions a single overall score can be assigned to each pair of activities, actually via the function *score* (Line b1). If no activity pair gets a sufficient matching score, w.r.t. a given threshold ρ (Line b3), `getBestAbstraction` returns the tuple $\langle \varepsilon, \varepsilon, \varepsilon, \varepsilon \rangle$ (Lines b3 and b6), simply meaning that no abstraction can be performed over the activities in S.

In the other case, 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. The choice of both the abstracting activity and the abstraction mode are essentially based on the similarity values computed via $sim^{\mathcal{D}}$ and $sim^{\mathcal{D}}_{G}$. 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 fact, we heuristically prefer to abstract two given activities by means the of ISA relation, whenever their mutual similarity is quite enough to consider them as different specializations of the same abstract activity.

Before considering the creation of a new activity for generalizing m_1 and m_2 (Lines b11-b12), it is first checked whether there is some existing activity that can abstract them both. More specifically, if there exists an activity o that already contains m_1 and m_2 the procedure simply indicates to (re)aggregate them into p via a $\mathcal{P}artOf$ link (Line b8); actually, this will not produce any real modification in the abstraction dictionary, in the calling procedure `AbstractActivities`. Otherwise, i.e., in the case one of the activities is implied by other, the former is abstracted into the latter by way of an aggregation relationship (Lines b9-b10); notice, in fact, that it can be excluded that the implied activity is a specialization of the other, since this condition was tested previously (Line b7). Eventually, if a new activity has instead to be created, we try to abstract m_1 and m_2 by using either an $\mathcal{I}sa$ relationship (Line b11) or a $\mathcal{P}artOf$ relationship (Lines b12).

4.4 An example Scenario (contd.)

Consider again the schema decomposition shown in Figure 3. Then, algorithm `BuildTaxonomy` starts generalizing from the leafs, thus first processing the schemas W_3, shown in Figure 3.(c), and W_4, shown in Figure 3.(d), associated with v_3 and v_4, respectively.

The result of this generalization is the schema $\overline{W}_1$ shown in Figure 3.(e), 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. The basic idea of the abstraction process is to iteratively aggregate pairs of activities into complex activities, yet 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 content of the abstraction dictionary.

In particular, when deriving the schema $\overline{W}_1$, only the activities d and p are aggregated again into the complex activity x_1, created previously, as it actually contains both of them; consequently, d and p are replaced with x_1 in the

schema. The schema $\overline{W}_1$ is then merged with the schema W_2 associated with v_2, and a new generalized schema, shown in Figure 3.(f), is derived for the root v_0. In fact, when abstracting activities coming from W_2, d and p are aggregated together, again into the complex activity x_1. Furthermore, the activities e and f are aggregated into the complex activity x_2, while m and o are aggregated into x_3. As a consequence, the pairs $(e, x_2), (f, x_2), (m, x_3)$ and (o, x_3) are inserted in the $\mathcal{P}artOf$ relation.

5 A System for Discovering Expressive Process Models

The whole approach discussed in the paper has been implemented in Java and integrated as a *mining* plugin in the process mining framework ProM [47]. This framework is quite popular in the Process Mining research community, and provides a platform for effectively developing algorithms for mining and analyzing process log data. In this section, we first discuss the conceptual architecture of our plugin and its implementation in ProM, and then we illustrate its application on our running example.

5.1 System Architecture and Integration in ProM

The ProM framework has been developed as a completely plug-able environment. An important feature of ProM is that it provides various components for implementing several basic functionalities (e.g., loading log files, setting mining parameters, visualizing mining results), as well as data structures for representing workflow schemas and intermediate results. The framework can be extended by adding five different kinds of plugins:

- *Import* plugin, for implementing "open" functions for external data, e.g., loading instance-EPCs from ARIS PPM;
- *Export* plugin, for developing "save as" functions for some objects, e.g., saving a workflow model as an EPC, or Petri net, or AND/OR graph, etc.;
- *Mining* plugin, for implementing mining algorithms;
- *Analysis* plugin, for implementing analysis functions on either log data or mining results;
- *Conversion* plugin, for enabling conversions between different data formats, e.g., from EPCs to Petri nets.

All the algorithms presented in Section 3 and Section 4 have been embedded in the AWS (Abstraction Workflow Schema) plugin, which is loaded dynamically at the start up of the framework. Figure 6 shows the conceptual architecture of the implementation, where colored modules are components made available by ProM. For instance, the plugin exploits the *Log filter* component of ProM,

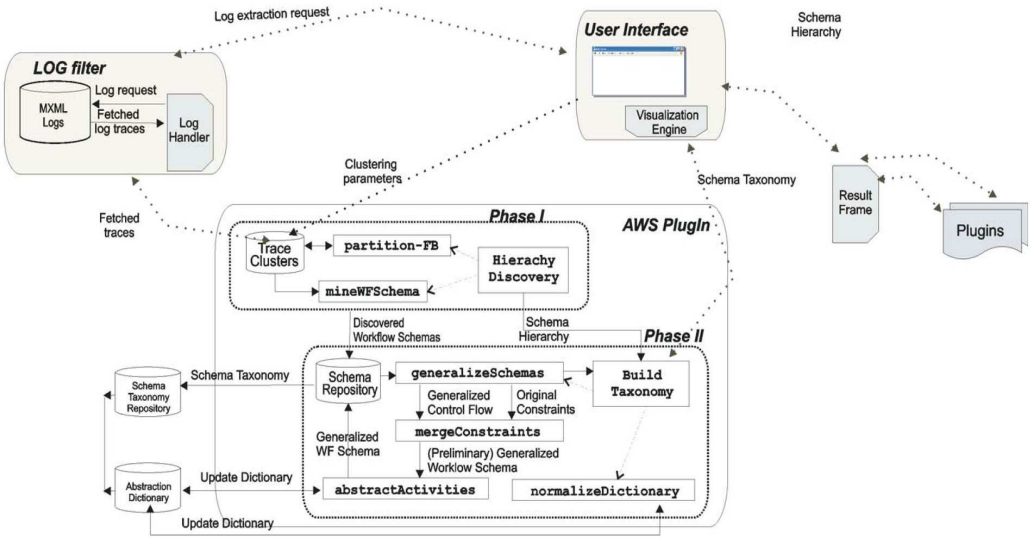


Fig. 6. Conceptual architecture of the AWS plugin.

which allows to load an MXML file (i.e., a process log in an XML-based format) in the framework and to make it available for data processing.

For the sake of clarity and conciseness, the major functional elements in AWS 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. By the way, one further, "internal", repository is used to maintain and share data on the trace clusters produced by the clustering algorithm and the schemas generated during both the mining phase and the restructuring one.

By reflecting the nature of our approach, the plugin works in two phases: first, a schema decomposition is computed by means of the `HierarchyDiscovery` module implementing the hierarchical clustering algorithm. Then, in the second phase, the mined model is visited in a bottom-up way, and it is restructured at several levels of abstraction, by means of the `BuildTaxonomy` module. The results are made accessible to other plugins through the standard *Result Frame* component, while they are presented to the user by exploiting the *Visualization Engine* component of ProM.

Note that the submodule `Partition-FB` has been realized by exploiting some of the components in the "DWS analysis" plugin available in ProM, which allows to cluster a given set of traces, based on a special kind of sequential patterns (used as features for the clustering)—in accordance with the approach proposed in [19]. In fact, besides the desired number of clusters, this plugin must receive two parameters, named σ_{DWS} and γ_{DWS}, which both range in the real interval $[0, 1]$. Roughly speaking, σ_{DWS} is a lower threshold for the

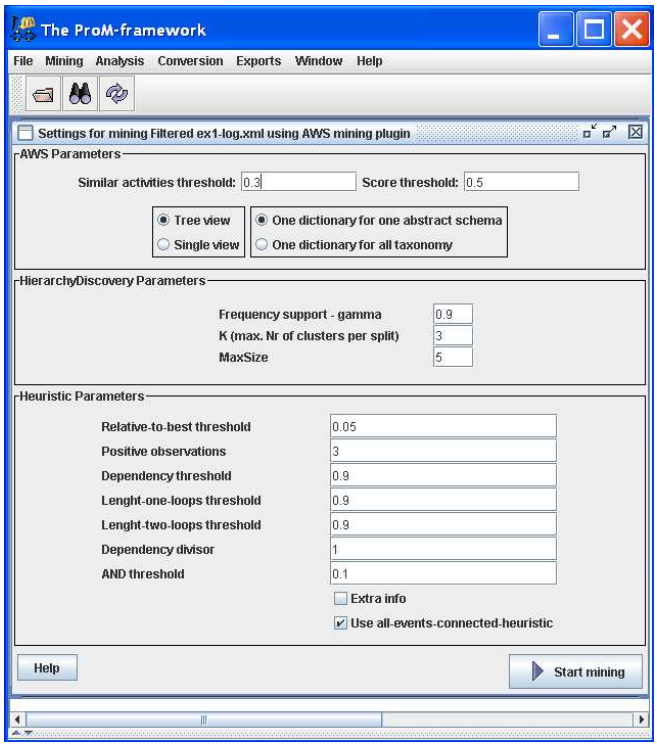


Fig. 7. Input panel of AWS plugin.

frequency of the patterns, while γ_{DWS} is an upper threshold for their degree "expectedness" w.r.t. the current workflow model (the lower is γ_{DWS} the more unexpected must be the pattern). In the current implementation of AWS both these thresholds are chosen in an automated way, so freeing the user of such an uneasy task. In more detail, while always fixing $\sigma_{DWS} = 0.05$, AWS first tries to make DWS use highly unexpected patterns for the clustering, by setting $\gamma_{DWS} = 0.01$; if no patterns are found, DWS is launched with $\gamma_{DWS} = 0.8$. In any case, arbitrary values can be set by interleaving DWS and AWS.

Finally, we note that within the architecture of ProM different process mining algorithms could be exploited to implement the submodule `MineWFSchema`. By default, we use the ProM *HeuristicMiner* plugin for this purpose, which is basically an implementation of the approach discussed in [48,13].

5.2 The system in action

We next illustrate the application of the AWS plugin on a sample log generated for the process *Order Management* depicted in Figure 1, which we have been using as running example throughout the paper. As shown in Figure 7, three groups of parameters have to be specified in the input panel of the plugin, which conceptually correspond to the functional modules `HierarchyDiscovery`, `BuildTaxonomy`, and `MineWFSchema`.

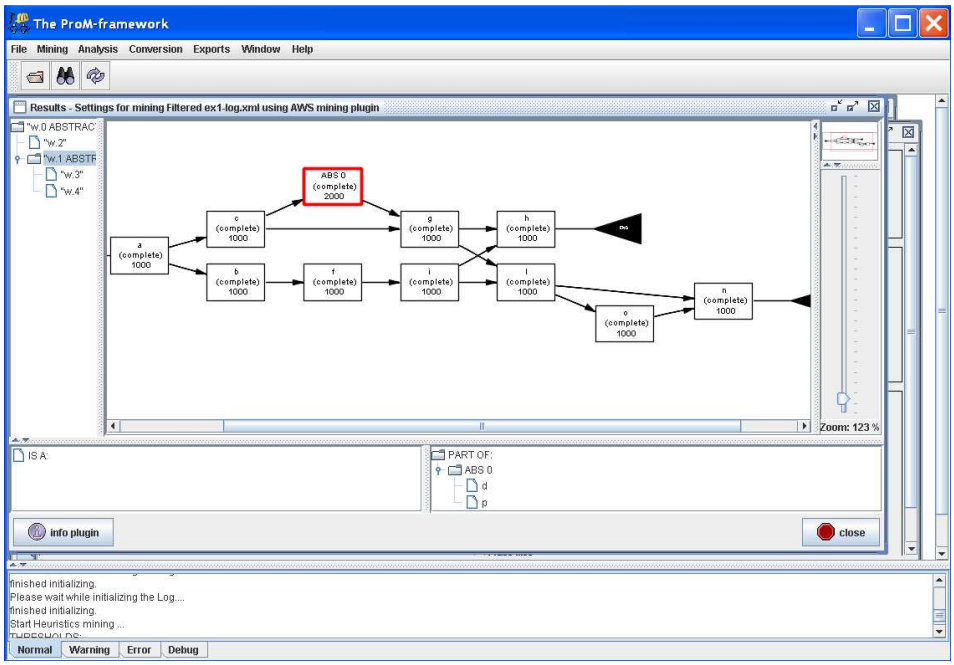


Fig. 8. Browsing the hierarchy: details on the non-leaf schema $w.1$.

In the upper part of the input panel, the user can set the parameters needed by the `BuildTaxonomy` module: the *Similar activities threshold* (corresponding to parameter ρ in the `getBestAbstraction` procedure of Figure 5), expressing the minimal score necessary for merging together a pair of activities, and the *Score threshold* (corresponding to parameter ρ^S in Figure 5), which influences the kind of abstraction relationship used to abstract activities. Moreover, the *Tree/Single view* options allow the user to choose whether either the whole hierarchy of schemas or simply its root shall be shown. The second set of parameters (in the middle part of the input panel) are those required by the `HierarchyDiscovery` module, i.e.: the lower threshold γ and the two upper bounds $maxSize$ and k. Finally, the last section of the panel concerns the parameters required by the algorithm used for mining each single workflow schema in the hierarchy (i.e., the functional module `MineWFSchema`). In fact, in the bottom of the input panel we can see the parameters of the *Heuristic Miner* algorithm.

In Figure 8, we report the results obtained by applying AWS plugin when *Similar activities threshold*=0.3, *Score threshold*=0.5, *Tree View* button is checked, *One dictionary for each abstract schema* button is checked. As we expected, the algorithm generates a schema taxonomy where the non-leaf schemas $w.0$ and $w.1$ are abstract schemas. Notice that when selecting any node in the hierarchy (shown in the right corner of the panel), the associated schema is visualized in the central part of the panel (in the figure, this is the case of node $w.1$). Finally, we note that in the bottom of the panel, the Abstraction Dictionary is made available to the user.

6 Experimental results

This section is meant to illustrate a thorough experimentation work we have conducted over different process logs to assess the effectiveness of our approach with regard to both the clustering scheme and the abstraction mechanisms. In particular, we shall describe: (i) the application of the approach to a simple benchmark log, in order to provide some further intuition on the whole abstraction mechanism; (ii) the result of tests aimed at assessing the quality of the clustering scheme; and, (iii) a concrete case study concerning a maritime freight harbor, providing some hints of the benefits that the combined usage of hierarchical clustering and abstraction mechanisms could yield in real-life application contexts.

6.1 Qualitative analysis on a benchmark log

In order to give some further intuition on the behavior of our approach, we considered one of the example log files (namely, *ExamplesOntologies-SAMXML.mxml*) that accompany the current version (4.2) of the ProM framework. This log consists of 500 traces and regards a process for the repairing of phone devices, which appears as a running example in a tutorial of ProM (available at `is.tm.tue.nl/~cgunther/dev/prom/PromTutorial.pdf`), and which is referred to as `PhoneRepair` hereinafter. In short, the process starts by registering a phone device (activity `Register`), which was sent by some customer. As soon as it is identified which sort of defect affects the phone (through the activity `Analyze Defect`), a letter is sent to the customer (task `Inform User`), while the device is sent to the Repair department. Actually, two kinds of repair activities can be performed based on the severity of the defect that has to be fixed, which are indeed named `Repair (Simple)` and `Repair (Complex)`. Every time a repair employee finishes working on a phone, this latter is sent to the QA department, in order to check whether the defect has really been fixed (activity `Test Repair`). If the phone is now working well, the case is archived and the phone is returned to the customer (activity `Archive Repair`); otherwise, the phone is sent again to the Repair department (activity `Restart Repair`).

Figure 9 reports the workflow schemas that were mined out of these log traces when using two classical process mining techniques, which are available as mining plugins in the ProM framework: *HeuristicMiner*, which essentially implements the approach introduced in [48,13], and α^{++}, which implements the approach in [49].

We then launched the *AWS* plugin described in Section 5, which implements our approach by using the *HeuristicMiner* for mining the workflow schema of

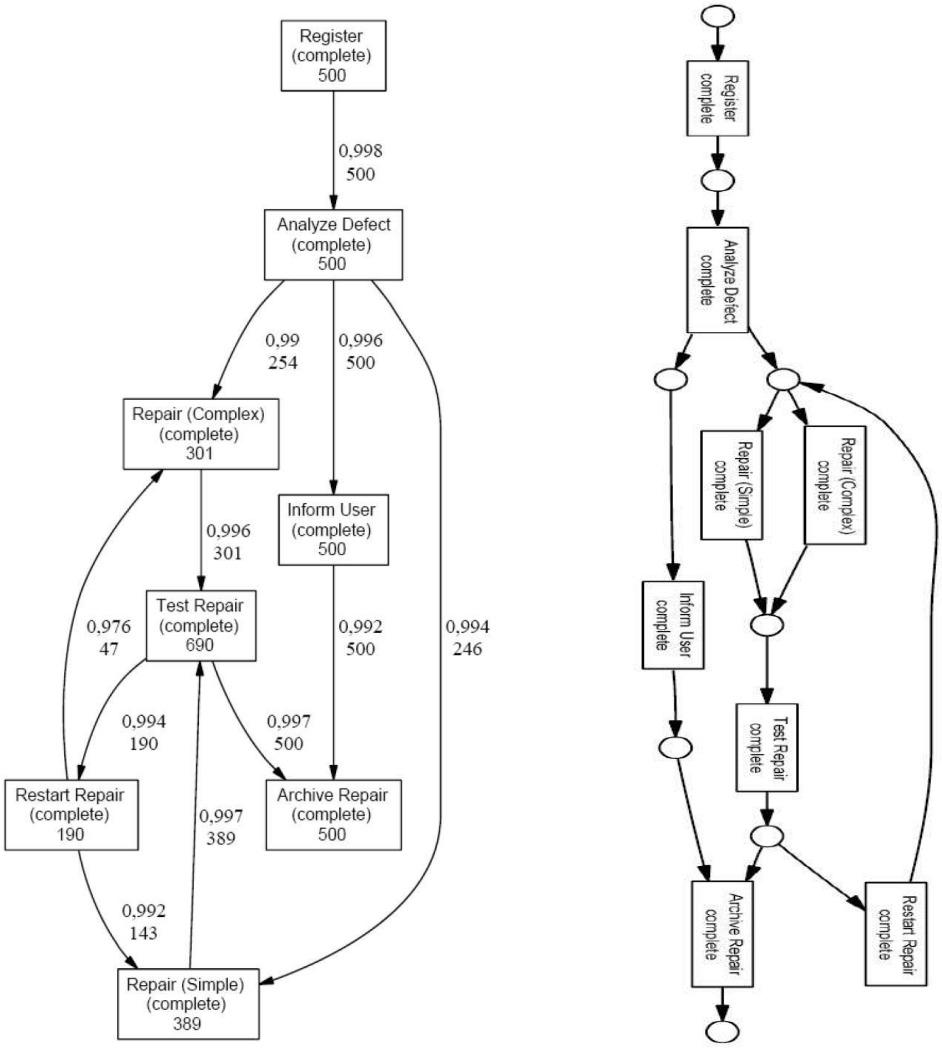


Fig. 9. Workflow schemas discovered for process `PhoneRepair` with the algorithms *HeuristicMiner* (left) and α^{++} (right).

each single cluster. As for the parameters of AWS, we required a single split of the log into at most 3 clusters, and set both abstraction thresholds (i.e., ρ and ρ^s) to their default value.

Figure 10 shows a screenshot of AWS, where the whole clustering hierarchy is sketched in the leftmost panel. The workflow schemas discovered for the leaves of the hierarchy, reported in Figure 11, evidence three main usage scenarios for the process, which are featured by the traces in the input log: in 125 cases, the defect was fixed by just one simple repair (schema $T.0$); in 232 cases, the defect was fixed by just one complex repair (schema $T.1$); and, in 143 cases, after trying to perform a simple repair, some further repairs (either complex or simple) were necessary to fix the problem.

In addition to offering a way to classify the executions registered in the log under analysis, the three schemas described above collectively provide a sounder

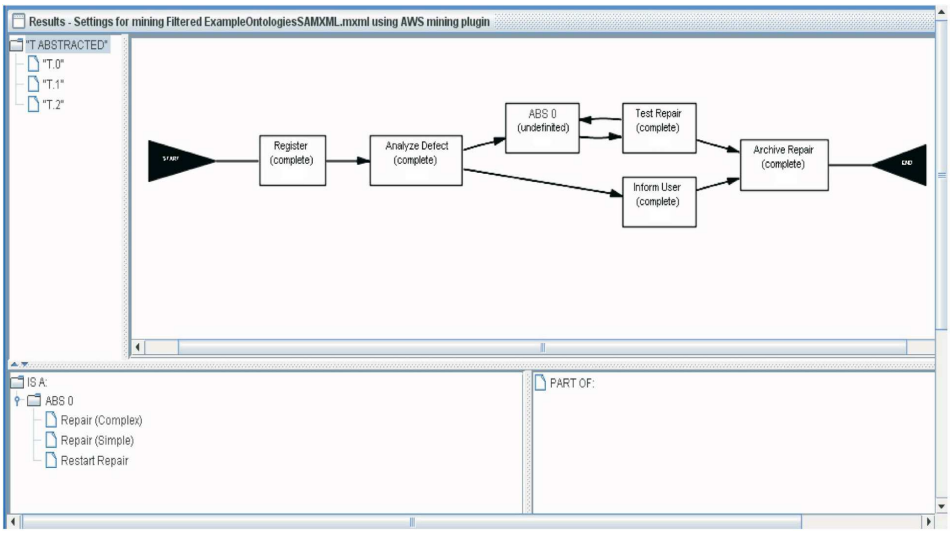


Fig. 10. Results obtained with the *AWS* plugin on the `PhoneRepair` process.

representation than the flat (single schema) models discovered with the algorithms *HeuristicMiner* and α^{++}. Indeed, it can be verified—e.g., with the help of the *LTL Checker* analysis plugin available in ProM—that the task `Repair (Simple)` does not occur in any of the traces where `Repair (Complex)` appears as the first repair action, despite this behavior is well allowed by the workflow schemas in Figure 9. Actually, this interesting result comes at the cost of some marginal decrease in the completeness of the discovered process model. In fact, the schemas of Figure 11 fail to model 22 log traces where, indeed, multiple repair actions occur with a complex one as the first of them.

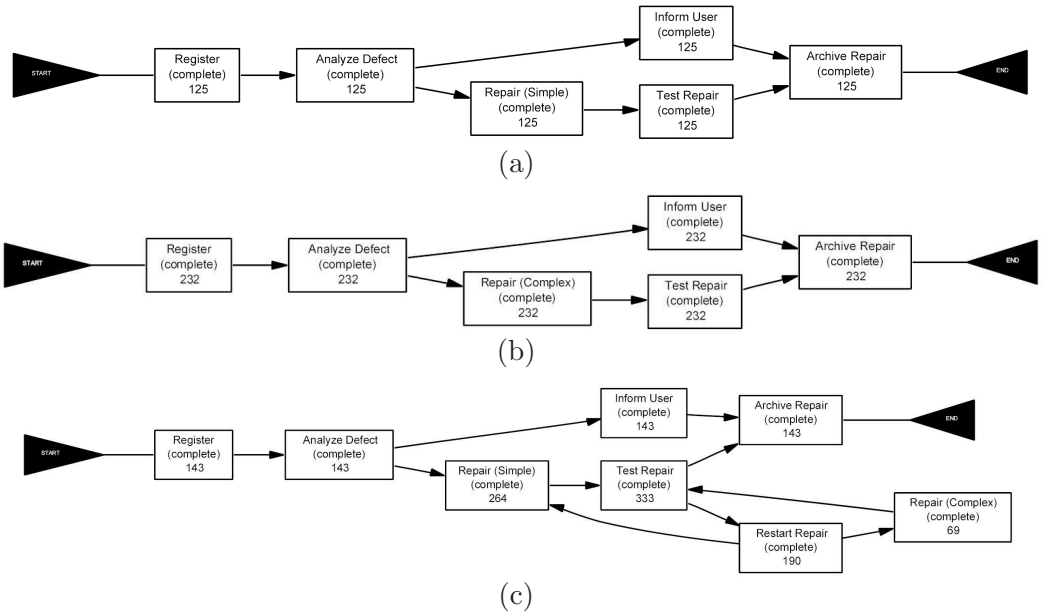


Fig. 11. Workflow schemas discovered by AWS for the leaf nodes $T.0$ (a), $T.1$ (b) and $T.2$ (c) of Figure 10.

In any case, if completeness is a major issue, then we could perform some finer clustering of the log, or mine each cluster schema with an algorithm different than *HeuristicMiner*, or simply use some different parameters setting for this latter—further analysis in such a direction is omitted for the sake of presentation.

The workflow schema that is derived by abstracting $T.0$, $T.1$ and $T.2$ is again shown in Figure 10. Notably, the activities `Repair (Complex)`, `Repair (Simple)` and `Restart Repair`, which do not appear in all of the leaf schemas, have been replaced with a new activity, named `ABS0`, abstracting them all. In fact, the abstraction dictionary computed by the AWS plugin contains an IS-A link between each of these three activities and `ABS0`, as it is shown at the bottom of the screenshot in Figure 10.

6.2 Conformance analysis on different benchmark logs

We next discuss some experiments we carried out to evaluate the effectiveness of our clustering-based process mining approach (sketched in Figure 2). Experiments have been performed on 10 log files, provided as benchmark datasets in the ProM framework and which have actually been used in the literature for the evaluation of other process mining approaches. These logs were generated synthetically, by considering different kinds of behavior, ranging from basic routing constructs, such as sequence, choice, parallel execution, and loop, to more complex ones, such as non-free-choice and invisible tasks.

In order to compare our approach with a few well-known process mining techniques, we took advantage of the following ProM plugins:

- *HeuristicMiner*, substantially implementing the technique described in [48],
- α^{++}, which implements the technique in [49], and
- *GeneticMiner*, implementing the approach in [14].

The two former techniques have also been used within our hierarchical clustering scheme as two different implementations for the procedure `MineWorkflow`, devoted to build a workflow for each discovered cluster. As a matter of fact, due to the expensive computation performed by the *GeneticMiner*, we did not explore the integration of this latter algorithm within our iterative clustering scheme, as this solution risks being inefficient in many real-world applications. Hence, two different configurations for our approach will be considered in the remainder of this subsection:

- *AWS-HN*, where `MineWorkflow` is implemented with *HeuristicMiner*, and
- *AWS-*α^{++}, where `MineWorkflow` is instantiated with α^{++}.

As discussed in Section 2, in order to evaluate the effectiveness of any process

mining technique, some quality measure is needed to express the capability of the discovered model to accurately capture the behavior recorded in the log, yet avoiding to introduce an excessive number of features and execution paths. In actual fact, different validation approaches have been proposed in the literature for this purpose, although none of them is a standard. Interestingly, the ProM plugin *ConformanceChecker* provides a shared basis to evaluate the conformance of a process model w.r.t. the log it was discovered from, on the condition that the model is represented as a Petri net [3]. In particular, we considered three conformance measures (ranging, as usual, in [0..1]) available with the *ConformanceChecker* plugin:

- *Fitness*, a sort of completeness measure defined in [34], which evaluates the compliance of the log traces with respect to a given process model. Roughly speaking, this measure considers the number of mismatches that occur when performing a non-blocking replay of all the log traces through the model (i.e., the tokens that must be created artificially, as well as those left unconsumed): the more the mismatches the lower the measure. In a sense, it quantifies the ability of the model to parse all the traces in the log.
- *Simple behavioral appropriateness* (shortly denoted by *SB-Precision* hereinafter), a precision measure defined in [34], which aims at estimating the amount of the "extra behavior" allowed by the model, with respect to that actually registered in the log: the more extra behavior, the lower the precision of the model. To this purpose, the possible behavior admitted by the model is quantified according to the average number of transitions that are enabled during a replay of the log—indeed, an increase in this number hints some higher degree of choice and parallelism.
- *Advanced Behavioral Appropriateness* (shortly referred to as *AB-Precision* henceforth), which is still a precision measure defined in [34] to express the amount of model flexibility (i.e., alternative or parallel behavior) that was not exploited to produce the real executions registered in the log. Notably, this measure is normalized by the so-called *degree of model flexibility*—which is 0 when just one particular sequence of the activities is admitted, and 1 when every possible sequencing of them is allowed—, estimated via a state space analysis of the model.

The above measures, being defined for a single workflow schema, cannot be directly applied over the tree-schemas mined with our approach. In this regard, we first remark that any conformance analysis has to be restricted to the leaf schemas only, since any other schema just offers some compact and approximated view over heterogeneous process instances. Moreover, in order to easy the comparison with classical process mining techniques, for each conformance

[3] A practical way to extend such a conformance test to other kinds of models (such as, e.g., the *Heuristics nets* returned by *HeuristicMiner*) is to exploit the conversion plugin available in ProM to translate such models into Petri nets.

measure, we report a single overall score for an entire schema decomposition, by averaging the values computed for all the leaf schemas. In more detail, the conformance values of these latter schemas are added up in a weighted way, where the weight of each schema is the fraction of original log traces that constitutes the cluster it was mined from.

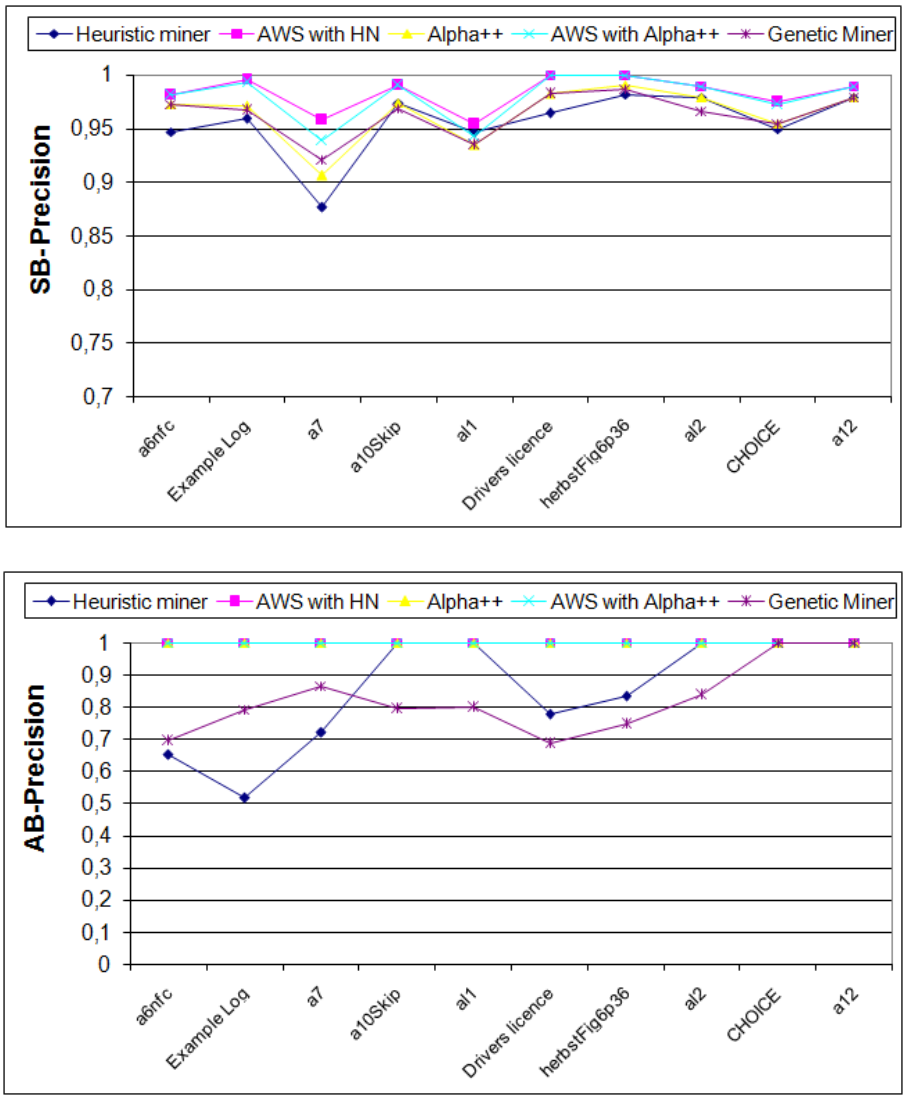


Fig. 12. Results on benchmark logs: behavioral appropriateness.

Figure 12 illustrates the precision of the models discovered with the techniques mentioned above, for each benchmark log. Note that, our clustering-based approach achieves outstanding results w.r.t. both precision metrics, almost independently of which basic technique is used to mine the workflow schema of each cluster. A finer grain analysis of the *SB-precision* measure, indicates that, as already noticed in the literature, both *GeneticMiner* and α^{++} work better than *HeuristicMiner* over logs following non-local (non-free choice) constraints, such as *a6nfc*, *Drivers licence*, *herbstFig6p36*. Moreover, our methods outperform classical ones and, in particular, the best results are

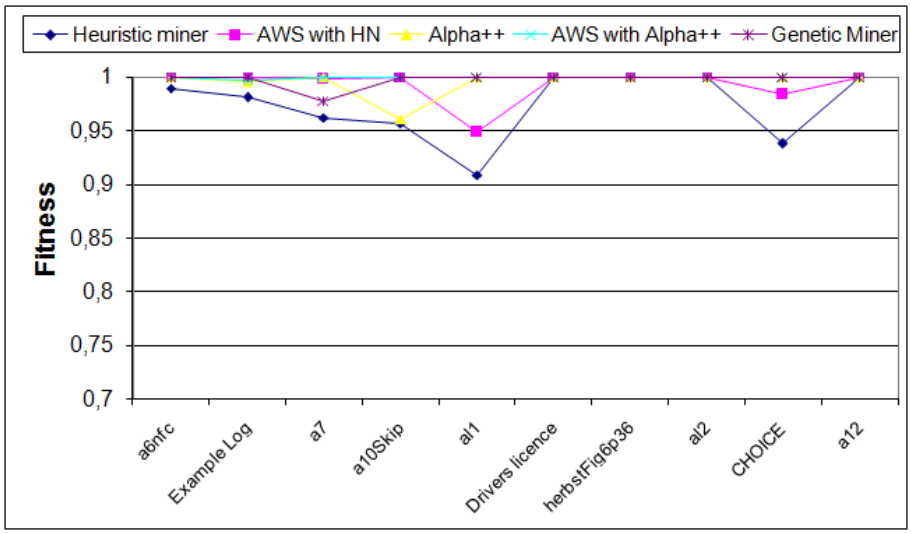
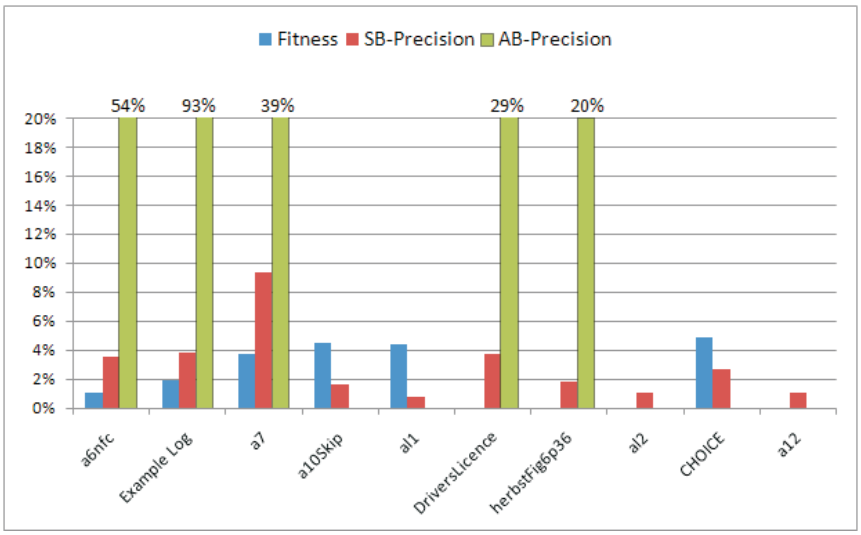


Fig. 13. Results on benchmark logs: fitness.

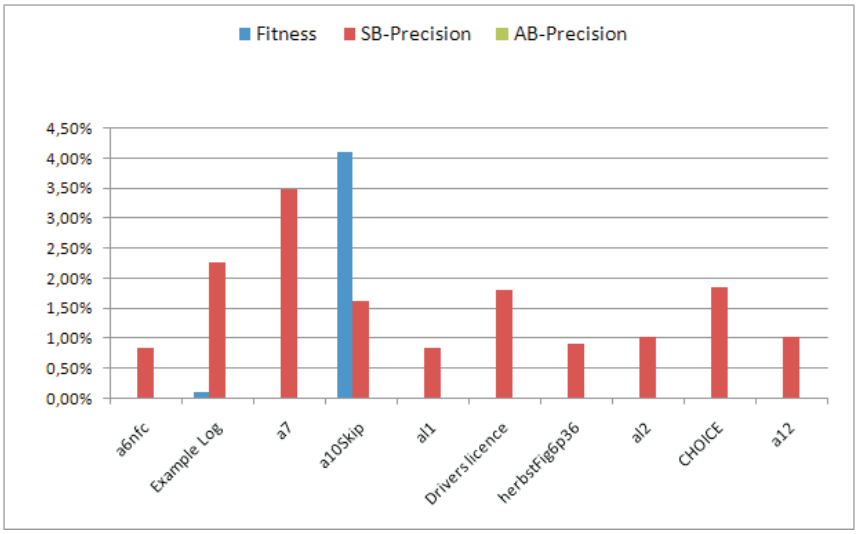
achieved when our clustering scheme is combined with *HeuristicMiner* (i.e., when the *AWS-HN* method is used). As for the *AB-precision* metric instead, it is worth noticing the impressive performance of the α^{++} algorithm, which actually reaches a maximal score (i.e., 1) against all the logs. By contrast, the other classical techniques seem to go worse with a number of logs. Interestingly enough, our approach succeeds in ensuring excellent results not only when it encompasses the α^{++} algorithm—in this case, in fact, it keeps achieving maximal precision—, but also when the cluster schemas are mined with *HeuristicMiner*.

In Figure 13 we reported result for the fitness measure. Note that very good results are obtained by α^{++} and by *Genetic miner*. In fact, this latter method seems to get nearly optimal results over all the logs but *a7*, which actually contains parallel branches. Very good results are obtained as well by both implementations of our approach, which always outperform their respective basic versions. In particular, the method *AWS*-α^{++} achieve optimal fitness against all the logs.

As a general remark, we may observe that over each possible conformance measure, both α^{++} and *HeuristicMiner* tend to receive substantial benefits by their integration within our clustering scheme. This effect is emphasized in Figure 14, which reports the increase (in percent) in the value of the three conformance measures that is achieved when passing from either α^{++} or *HeuristicMiner* to the respective clustering-enhanced version (i.e., *AWS-HN* and *AWS*-α^{++}, respectively). It is quite remarkable the behavior of *HeuristicMiner*, which suffers considerably when applied to logs involving complex routing constructs and non-local task dependencies (e.g., the log files *a6nfc*, *herbstFig6p36*, *DriversLicence*). Such deficiency, indeed, seems to be well overcome when the same algorithm is combined with our approach (i.e., in the case



(a) Algorithm *HeuristicMiner*



(b) Algorithm α^{++}

Fig. 14. Improvement of two classical process mining algorithms when integrated in our clustering scheme.

of method *AWS-HN*). This proves that more complete and precise process models can often be discovered with our approach, where different execution scenarios are modeled separately, and where some suitable kind of behavioral patterns, capable of capturing complex task relationships, are exploited to cluster the log traces. As for α^{++}, an appreciable improvement can still be observed for the *SB-Precision*; on the other hand, the other two measures do not receive any substantial increase, if we exclude the only case of the fitness value obtained on the log *a10Skip*. This fact is not so surprising, seeing that the basic version of algorithm α^{++} already reached an optimal performance for both these measures.

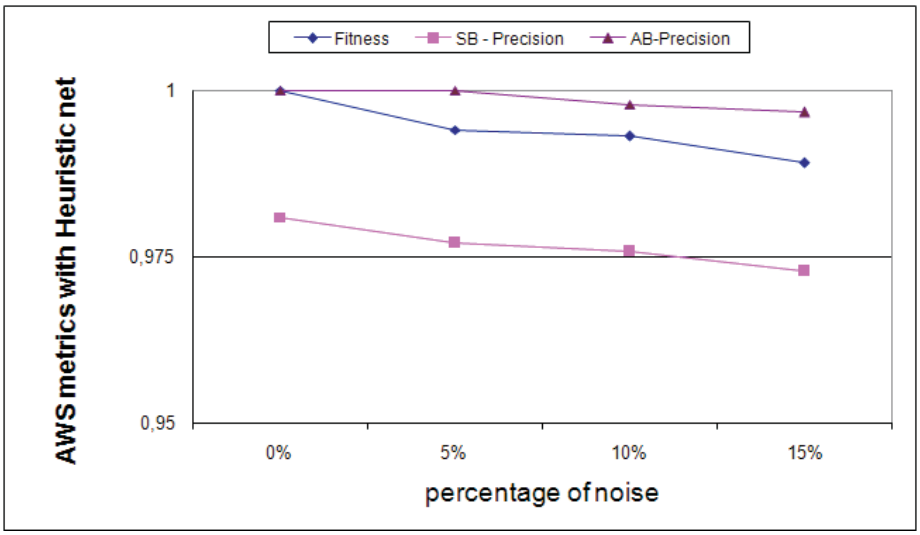


Fig. 15. Results on benchmark logs: sensitivity to noise.

Finally, a further series of tests were carried out to evaluate the impact of noise on the results obtained by the default instantiation of our approach (namely, method *AWS-HN*). Figure 15 refers to one of these experiments, where the benchmark log *afnc*[4] was randomly perturbed by adding different kinds of noise to a number of traces (i.e., missing head/body/tail, deletion of a task, swap between two tasks). More precisely, the figure reports the conformance measures obtained by the method *AWS-HN* on the original log (0% noise), as well as on three perturbed versions of it, containing 5% to 15% noisy traces. The figure allows to appreciate that, even in presence of some notable amount of noise in the input log, our approach manages to achieve very good results with all the conformance measures.

6.3 Experiments on real data

In order to assess the validity of the approach in practical application contexts, we applied it on some real log data coming from an important Italian harbor (Gioia Tauro) acting as a maritime freight hub.

The application scenario. The operational systems used in the harbor continually support and register several logistic activities for each container that passes through the port (about 4 millions of containers per year). Containers reach and leave the port either by ground or by sea. In our analysis, we shall deal with the handling of containers which arrive and depart by ship ("transhipment" flows), and focus on the different kinds of moves they undergo

[4] An analogous behavior was experienced with the other benchmark logs, but we omit further details for simplicity of presentation.

over the "yard", i.e., the main area used in the harbor for storage purposes, which measures about $800,000$ square meters.

The yard is logically partitioned into a finite number of bi-dimensional *slots*, which constitute the unitary amounts of yard space that can be used for the storage of containers. Slots are grouped in a fixed number of *blocks*, which are, in their turn, organized in *sectors*. Conventionally, the name of a sector, say `A`, is a prefix of the name of any block contained in it, e.g., `A_1`, and `A_2`. Moreover, inside most yard sectors, multiple containers can be stocked in a single slot, by piling them up, one on top of another. Therefore, a vertical coordinate, named *layer*, is introduced to univocally identify any distinct storage position.

In a sense, containers are the atomic subject of the logistic processes in the hub system. The container life cycle can be summarized as follows. The container is initially unloaded from the ship, with the help of a crane, and temporarily stocked within a zone near to the dock. Then, it is carried to some slot of the yard, which is chosen based on expectation information that concerns both the ship on which it is going to embark and the boarding time. Different kinds of vehicles can be selected for carrying the container, which are chosen mainly based on both the sector which must be reached and the distance to cover. These vehicles include, in particular, different kinds of cranes, straddle-carriers (a vehicle capable of picking and carrying a container, by possibly lifting it up), and multi-trailers (a sort of train-like vehicle that can transport many containers). In particular, a yard crane is involved when the final destination is too high to be be reached by the vehicle used for carrying the container. Symmetrically, at boarding time, the container is first placed in a yard area close to the dock, and then loaded on the cargo by means of a (dock) crane. This basic life cycle may be extended with a number of additional movements—classified as "house-keeping" in the jargon used in the harbor—which are meant to let the container approach its final embark point, or to leave room for other containers.

Most of the operations traced in the hub systems correspond to some move performed on a container by a human operator, with the help of some suitable harbor vehicle. As far as concern the experimental setting considered here, the following basic operations can be applied to any container c:

- `MOV`, when c is moved from a yard position to another by a straddle carrier;
- `DRB`, when c is moved from a yard position to another by a multi-trailer;
- `DRG`, when a multi-trailer moves to get c;
- `LOAD`, when c is charged on a multi-trailer;
- `DIS`, when c is discharged off a multi-trailer;
- `SHF`, when c is moved upward or downward, possibly in order to switch its position with another container;
- `OUT`, when c is embarked on the ship with a dock crane.

The mission of the hub is to offer high quality of service to the navigation lines, while reducing the overall cost of internal logistic processes. Critical performance measures are the latency time elapsed when serving a ship (where, typically, a number of containers are both discharged off and charged on), and the overall costs of moving the containers around the yard. A key factor impacting on both these measures is the number of "house-keeping" moves that are applied to the containers. Minimizing these operations is a major goal of the policies established to decide where to place a batch of containers which are coming to the harbor, based on different features of the containers, such as the kind of conveyed goods (possibly needing some refrigerating equipment), their origin and their next destination. These decisions are eventually taken by harbor managers with the help of sophisticated planning tools, which mainly rely on some, necessarily simplified, model of the operational environment as well as on the knowledge of future events (e.g., which ships are going to take each container and when this is going to happen). Unfortunately this information is often unavailable or even incorrect, while diverse types of delays or malfunctioning are likely to occur.

The conspicuous level of complexity and unpredictability that affects the application scenario described above calls for the introduction of techniques for the ex-post analysis of yard operations, which could give some feedback for the policies ruling the allocation of the yard space, and which could support both tactic and strategic decision processes. In particular, the discovery of workflow models can be very beneficial in this context, in order to gain a compact representation of the logistic processes captured in the log, by illustrating the flows of work that were really carried out.

Application of the approach to the harbor logs. We selected a subset of the historical data registered in the harbor systems, corresponding to the logistic operations performed on all the containers that completed their entire life cycle in the hub along the first two months of year 2006, and that were exchanged with other ports around the Mediterranean sea—about 50Mb log data pertaining 5336 containers. In order to apply our analysis approach, we regarded those data according to a process-oriented perspective, by considering the transit of any container through the hub as a single enactment case of the (logistic) process under analysis.

Test (A): Our first experimentation was focused on extracting a model describing the life cycle of the containers, in terms of the basic logistic yard operations mentioned above (i.e., `MOV`, `DRB`, `DRG`, `LOAD`, `DIS`, `SHF`, `OUT`). To this aim, for each container, we built a distinct log trace recording the sequence of operations that were applied to that container. In order to guarantee the existence of well specified start and end activities, we also introduced two dummy activities to mark the beginning and the end of each log trace, denoted by

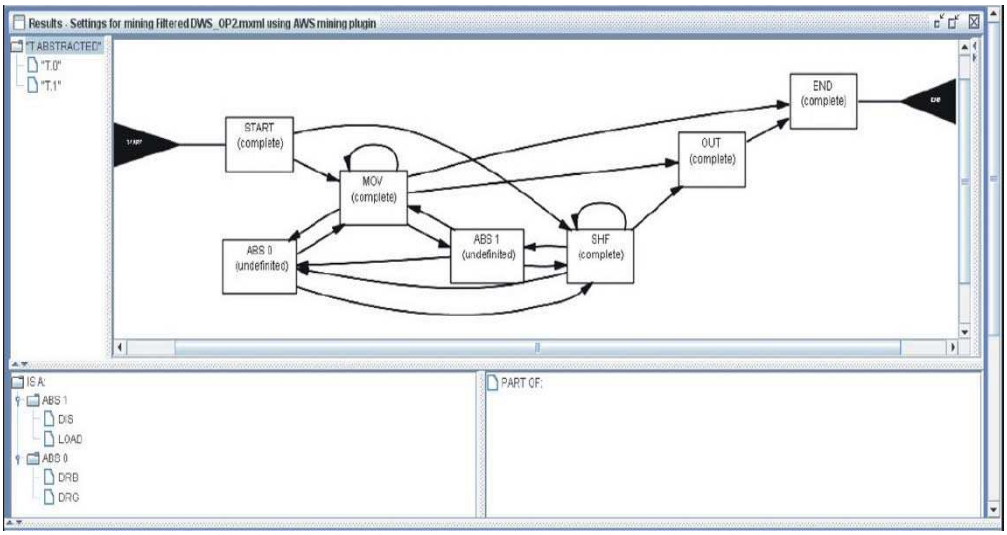


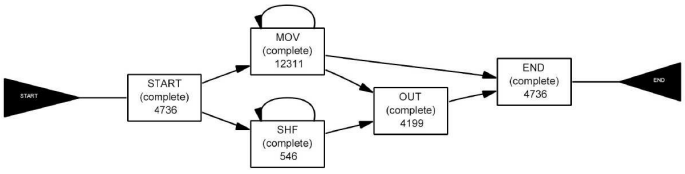
Fig. 16. The schema hierarchy and abstraction relationships for the container life–cycle: focus on the (abstract) root schema.

START and END, respectively.

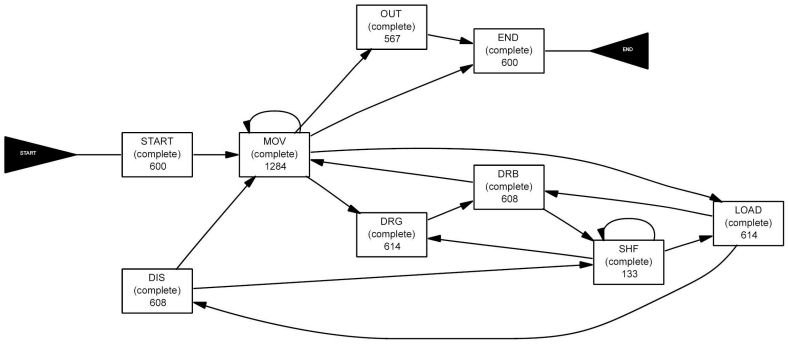
Figure 16 is a screenshot of the AWS plugin, showing the schema taxonomy discovered on these data, and the workflow schema associated with the root node. This schema provides a high-level view over the logistic process, which features only three of the original basic operations, namely MOV, SHF, and OUT, in addition to the two dummy ones (i.e., START and END). In fact, the remaining four operations have been abstracted into two higher-level activities, as it is shown in the bottom of the figure, where the contents of the abstraction dictionary are pictured. Note that the tool has grouped those operations related to the movement of a multi-trailer (i.e., DRB and DRG), and those concerning the action of charging/discharging a multi-trailer (i.e., LOAD and DIS).

The leaves of the taxonomy are illustrated in Figure 17, from which it emerges the existence of two different behavioral scenarios. Essentially, the workflow schema of node *T.0* (see Figure 17.(a)) only models the cases that did not involve any of the operations related to multi-trailer vehicles, which conversely characterize the other leaf schema (see (Figure 17.(b)). Notably, the former schema represents the most frequent way of handling containers, for it captures 4736 log traces out of the original 5336 ones. This is in line with a prominent goal of yard allocation policies, which attempt to keep each container as near as possible to its positions of disembarkation/embark, and to perform just a few short movements by means of straddle-carriers.

Test (B): A second kind of experiments were conducted to gain some expressive model for the flowing of containers through the yard, which could well help in analyzing the usage of the different storage areas, as well as to detect and explain overly intricate or long (and hence costly) moving patterns. Different options exist for such a kind of analysis w.r.t. how to deal with the



(a) Workflow schema for node T.0

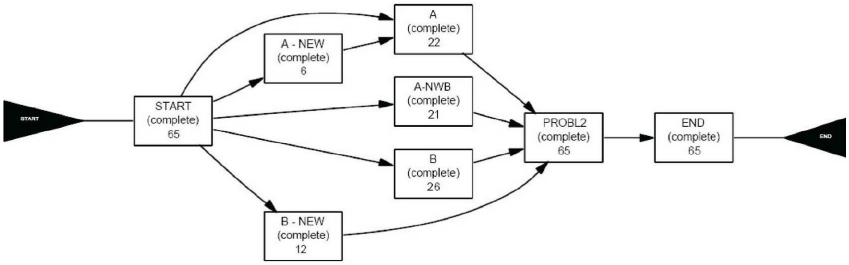


(b) Workflow schema for node T.1

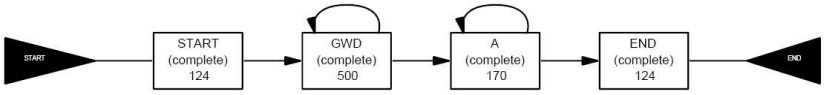
Fig. 17. Detailed workflow schemas discovered for the container life cycle.

granularity of yard positions.

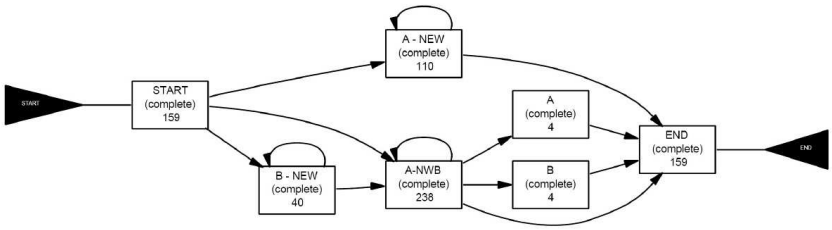
Firstly, we considered the idea of capturing the passage of containers through the different sectors of the yard. In this case, each log trace encodes the sequence of yard sectors occupied by a single container during its stay. Figure 18 shows three of the 5 leaf schemas discovered with the AWS tool from these data, which describe quite different scenarios for the movement of containers. In particular, the upmost schema (see Figure 18.(a)) give some insight on the path followed by a group of containers that incurred into exceptional events, as it is witnessed by the label `PROBL2`, actually referring to an area devoted to special checking activities. Another important usage case is captured by the schema in Figure 18.(b), evidencing the stationing of containers over two distinct sectors, namely `GWD` and `A`, with a rather high frequency of shifts within the first of them. Indeed, this evidences a rather ineffective, and yet not so sporadic, pattern for the handling of containers. Similar considerations apply to the schema in Figure 18.(c), which models a subset of handling cases, where the storage of containers substantially hinged on the usage of sectors `A-NWB` and `A-NEW`, which are quite close to each other. In particular, an interesting moving pattern is revealed, indeed, for the containers passed through sector `A-NWB`: in most cases a number of house-keeping moves were made inside that sector, while a quota of containers were even dispaced to further sectors (namely, `D-NEW`, `A`, `B`). Further analysis performed on all the 5 clusters, with the help of classical data mining techniques, has allowed to discover that some of them are correlated with the occurrence of critical environmental conditions, as well as with differences in the modus operandi of some yard managers. In this way, in addition to support the analysis and tuning of the



(a)



(b)



(c)

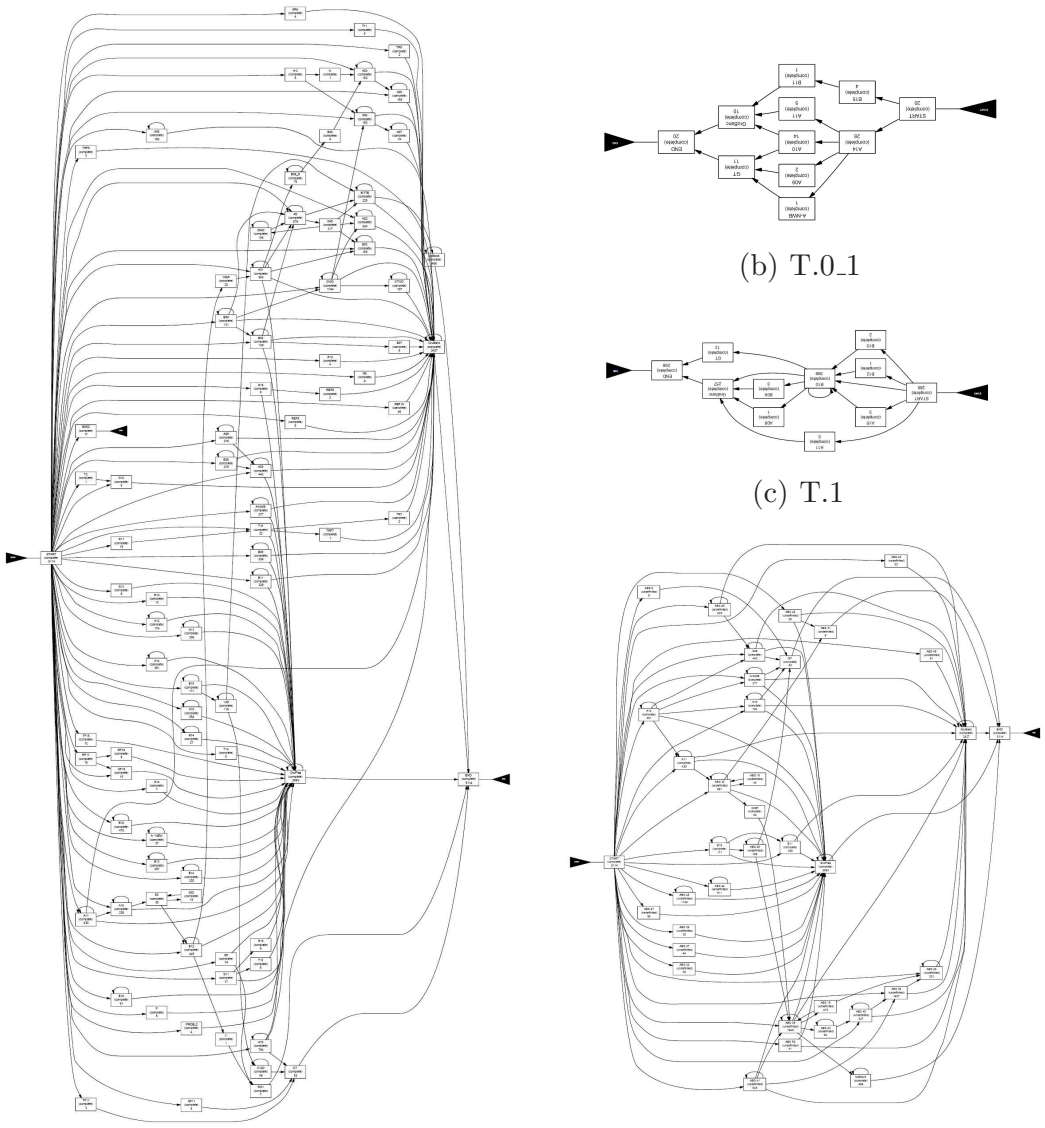
Fig. 18. Transit of containers through the yard blocks: detailed workflow schemas for three discovered clusters.

yard allocation policies, the approach contributed to foster the elicitation of some know-how, which was not yet encoded explicitly in the process models used at the harbor.

Test (C): In our final experimentation, in order to provide an idea of the practical advantages that can come from enhancing a process mining approach with abstraction mechanisms, we looked at the paths of containers over the yard, but at the level of the (nearly 100) blocks forming the yard space.

By applying the AWS plugin to these data with $maxSize = 5$, $k = 3$ and $\gamma = 0.8$, we obtained a taxonomy of workflow schemas consisting of five nodes, structured into three abstraction levels: the root T, two nodes $T.0$ and $T.1$, as children of T, and two children of $T.0$, denoted as $T.0\_0$ and $T.0\_1$.

The workflow schemas discovered for the two latter levels are shown in Figure 19. Note that a neat difference in complexity exists between the schemas $T.0\_1$ and $T.1$ (shown in the Figures 19.(b) and 19.(c), respectively) on the one hand, and the other leaf schema $T.0\_0$ (shown in Figure 19.(a), and about 90 nodes) on the other hand. However, a more succinct representation has been obtained for the higher level views: the schema of node $T.0$, shown in



(a) T.0_0 (b) T.0_1 (c) T.1 (d) T.0

Fig. 19. Transit of containers through the yard blocks: three concrete workflow schemas (a,b, and c) and an intermediate abstract view (d) over two of them.

Figure 19.(d), which consists of 37 nodes, and the root schema, reported in Figure 20, where the number of nodes falls down to 32.

It is interesting to observe that the simplification in the description of high-level schemas is not merely syntactical. Indeed, many of the activities that have been abstracted together exhibit some sort of affinity, thereby evidencing that the algorithm went beyond the structural properties. As an evidence for such an outcome, we next report a few couples of activities (i.e., yard blocks in this case) that were merged together by means of *IS-A* or *PART-OF* relationships:

- (`004D`,`04D`)—subsequently reckoned as two different names for the same

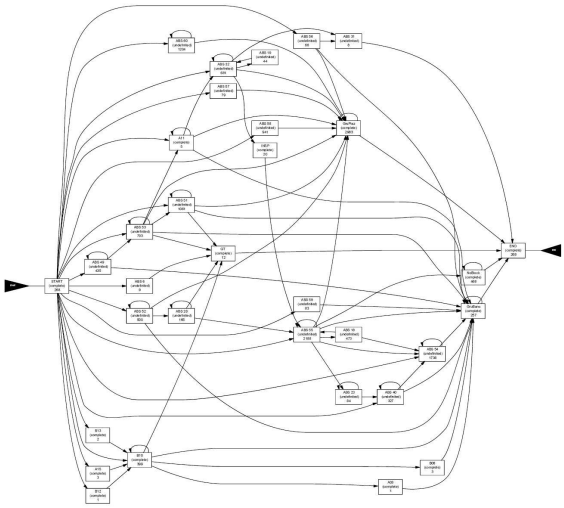


Fig. 20. Transit of containers through the yard blocks: the most abstract schema.

block, due to a mispelling error;

- (`REF01`,`REF5`)—two blocks of the same sector, both equipped with refrigerating systems;
- (`TR5`,`TRF5`), (`TR7`,`TRF7`)—codes for multi-trailer vehicles registered as (mobile) positions in the hub system;
- (`45D`,`8D`), (`012D`,`12D`), (`A`,`A-2`)—each of which is a couple of blocks belonging to the same sector.

We note that, due to the high number of activity labels that come in this case scenario, whatever flat representation of the container flows, which just consisted of a single workflow schema, would have been rather cumbersome and unsuitable for analysis purposes. By contrast, the taxonomies of schemas computed with our approach have been proved to effectively enable an explorative analysis of the available log data, by virtue of both the explicit separation of distinct classes of behavior and to the compactness of higher-level schemas.

7 Related Work

Several process mining approaches have been proposed in the literature, and many of them have been already integrated in the process mining framework ProM [47]. Most of the differences among these proposals resides in the modelling features that can be used to represent a workflow model and in the specific algorithms used for discovering it. For example, in [2], processes are intuitively represented through pure directed graphs, which allow to express precedence relationships only, while disregarding richer control flow constructs, such as concurrency, synchronization and choice. Many other proposals exploit, instead, more expressive languages, which sometimes enjoy deep formal foundation for modelling and analyzing workflow processes, as in the case of

Petri-nets, used, e.g., in [45,44,46].

In particular, in all these latter works, a special kind of Petri nets, named Workflow nets (WF-net), is adopted for modelling a process, which, in addition to the basic routing constructs, can express additional constructs, such as loops, deterministic choice, etc. The general problem of discovering a WF-net workflow model is specifically analyzed in [46], where the concept of *structured workflow (SWF) net* is introduced to capture a class of WF-nets that a process mining algorithm should be able to rediscover. Here an algorithm, named α, is presented which can rediscover an SWS net out of a log, provided that the log is guaranteed to enjoy some well-specified properties. The α algorithm was extended in [13] with some preprocessing and postprocessing strategies that make it capable to discover short loops. Moreover in [49] it is devised an extension of the algorithm that can explicitly capture non-free-choice constructs, which are a form of implicit dependencies between the process tasks.

In [48] a heuristic approach is presented that exploits simple metrics concerning task dependency and task frequency, in order to eventually produce a graph-based process model, called "dependency/frequency graph". Notably, the approach is meant to cope with the presence of noise in the logs.

A different approach to mining a process model from event logs is described in [35], where a mining tool is presented that can discover hierarchically structured workflow processes. Such a model corresponds to an expression tree, where the leafs represent tasks (operands) while any other node is associated with a control flow operator. In this context, the mining algorithm mainly consists of suitable term rewriting systems.

Yet another approach is adopted in [23,24], where a subset of the ADONIS definition language [26] is used to represent a block-structured workflow model. The peculiarity of the approach mainly resides in its capability of recognizing duplicate tasks in the control flow graph, i.e., many nodes associated with the same task. The algorithm proposed there, named "InWoLvE", solves the process mining problem in two steps: an induction step, where a stochastic activity graph (SAG) is extracted out of the input log, and a transformation step, where the SAG is transformed into a block-structured workflow-model. Recently, some extensions to this approach have been proposed in [20], in an interactive setting, where the analyst can iteratively refine the process mining results by evaluating the mined models and varying the parameters of the mining tool. After discussing a number of issues involved in interactive process mining, the authors of [20] introduce some techniques supporting such a setting, which primarily include a validation procedure for checking the (preliminary) mined model, and a structured layout algorithm that is stable against small changes of the mined model.

The problem of discovering a process model from execution logs is also con-

sidered in [8], as a special case of the Maximal Overlap Sets problem in graph matching. The paradigm of planning and scheduling by resource management is used there in order to devise an efficient approach tackling the combinatorial complexity of the problem.

The approach proposed in [14] tries to overcome the difficulty encountered by previous process mining techniques when dealing with non-trivial routing constructs and noisy data, by resorting to the use of genetic algorithms. Indeed, this kind of algorithm offers a way to discover non-trivial constructs, mainly due to the global search they perform over candidate process models.

A similar motivation inspired the work in [19], where a process mining algorithm is proposed that can account for the identification of different variants of the process at hand. The technique mainly founds on clustering log traces according to structural patterns, and eventually produce a different, specific, workflow schema for each of the discovered process variants. The technique proposed in this paper shares with the one presented in [19] the basic idea of explicitly recognizing different use cases of a process by means of a clustering process. However, three main points make different our approach from the one in [19]: (i) the core, clustering-based, mining algorithm is made able to compute a hierarchy of workflow schemas, rather than just a flat collection of workflow schemas, (ii) the algorithm is extended to accommodates the use of any arbitrary process mining technique for equipping each node with a model, and (iii) the clustering is integrated with an abstraction-based restructuring method that allows to eventually produce a taxonomy of process models that represent an articulated view of the process, at different abstraction levels. As a matter of facts, the latter feature makes our approach neatly different from any other process mining approach as well.

Taxonomical structures are widely recognized as a valuable tool for eliciting, consolidating and sharing relevant knowledge in disparate application contexts, which can profitably support a variety of tasks (see, e.g., [39,40,1,12,50] for some works on this topic). However, defining a taxonomy and, more generally, an ontology is quite a difficult and time-consuming task, especially when it is intended to capture the structure of a rich and complex application domain. Therefore, some efforts have been spent to facilitate this task, by developing automatic techniques supporting the extraction of abstract concepts (see, e.g., [11] for the case of tendering systems in an e-commmerce scenario).

The possibility of defining taxonomies for business processes was first considered in [30], where a repository of process descriptions is envisaged to support both design and sharing of process models. Several frameworks for precisely defining a specialization/generalization of a process model, according to some suitable behavioral semantics, have been proposed for different modelling formalisms, such as, e.g., Object Behavior Diagrams [38], UML diagrams [37], process-algebra specifications and Petri-nets [6,43], DataFlow diagrams [27].

On the other side, a large body of work was done with regards the transformation of various kinds of schemata by means of abstraction techniques, in order to reduce their complexity. In particular, the definition of aggregated view over a given workflow schema is examined, e.g., in [7,29]. In particular, in the latter work a technique for deriving such a view is defined that automatically aggregates real activities into "virtual" ones, yet guaranteeing that all original ordering relationships among the activities are preserved.

However, we pinpoint that no substantial human intervention is required by our technique for generalizing process schemata, differently from [7] and [29], where the user is in charge of selecting which activities should be abstracted. In fact, the main distinguishing feature of our approach is the combination of mining and abstraction methods for automatically produce a hierarchical process model, which satisfactorily captures the behavior of the process at hand, without any pretension of being an executable workflow. Indeed, very few efforts have been paid to support some kinds of abstraction (e.g., techniques in [10,48,22] are able to produce models that focus on the main behavior as reported in the log, by properly dealing with noise).

As a final remark, we notice that, in our context, abstraction is exploited to make more compact the schemata at higher levels of the hierarchy, which are essentially meant as synthesized views over the whole process, or over some of its variants. Therefore, we do tolerate the introduction of some approximation in the control flow relationships, especially as concerns those involving abstracted activities, differently from [29] and other formal methods for dealing with process specializations [37,38,6,43,27].

8 Conclusions

We have proposed an automatic process mining approach that is meant to discover a taxonomical 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 decomposition, accurately modelling the process at hand, is first discovered by using a divisive clustering algorithm; the schema 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 approach has been encoded in a series of algorithms, which have been implemented as a plugin for the ProM framework.

A number of issues are still open and can be subject of future work. First, the recognition of activities to be abstracted together, which currently relies on simple matching functions, could benefit from the availability of background knowledge on the semantics of activities, possibly extracted from a given the-

saurus or a process ontology. Moreover, we think that the discovered process taxonomy can profitably be exploited to analyzing relevant measures, such as usage statistics and performance metrics, along the different usage scenarios of the process at hand. Specifically, by using a taxonomy as an aggregation hierarchy for multi-dimensional OLAP analysis, it is possible to enable the user to interactively evaluate such measures over different groups of process instances. Analogously, abstract activities and their associated abstraction relationships produced during the abstraction process can be a basis for defining aggregation hierarchies. Notably, the extension of the proposed approach with OLAP features can be a valuable tool in an interactive process mining setting, like the one considered in [20], since it can effectively support the user in evaluating the discovered process models, as well as in tuning the parameters in subsequent mining sessions. On the other hand, the discovered taxonomies can be exploited as a basis for further knowledge discovery tasks, such as the mining of generalized association rules between, e.g., the users or the resources involved in the workflow process under analysis; in particular, we believe that the application of techniques allowing for multiple support thresholds (such as, e.g., [39]) can effectively help recognizing interesting deviations or exceptions in the enactments of the process.

Finally, we are planning to make the approach parametric w.r.t. the algorithm used for recursively partitioning the input log, in the first phase of the approach, by possibly reusing the different clustering methods that have recently been integrated in the ProM framework.

Acknowledgements. The authors thank the anonymous referees for their useful comments and suggestions, which helped improving the quality of the paper. The work was partially supported by M.I.U.R. under project TOCAI.IT and by the R&D.LOG Consortium under project PROMIS.

References

[1] G. Adami, P. Avesani and D. Sona. Clustering documents into a web directory for bootstrapping a supervised classification. *Data & Knowledge Engineering*, 54(3):301–325, 2005.

[2] R. Agrawal, D. Gunopulos, and F. Leymann. Mining process models from workflow logs. In *Proc. 6th Int. Conf. on Extending Database Technology (EDBT'98)*, pages 469–483, 1998.

[3] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. of SIGMOD'93*, pages 207–216, 1993.

[4] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th Int'l Conference on Very Large Databases*, pages 487–499,

1994.

[5] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *Proc. of Intl. Conf. on Very Large Data Bases (VLDB06)*, pages 918–929, 2006.

[6] T. Basten and W. M. P. van der Aalst. Inheritance of behavior. *Journal of Logic and Algebraic Programming*, 47(2):47–145, 2001.

[7] F. Casati, M. Castellanos, U. Dayal, and M.-C. Shan. iBOM: a platform for intelligent business operation management. In *Proc. 21st Int. Conf. on Data Engineering, (ICDE 2005)*, pages 1084–1095, 2005.

[8] C. W. K. Chen and D. Y. Y. Yun. Discovering process models from execution history by graph matching. In *Proc. 4th Intl. Conf. on Intelligent Data Engineering and Automated Learning (IDEAL 2003)*, pages 887–892, 2003.

[9] J. E. Cook and A. L. Wolf. Automating process discovery through event-data analysis. In *Proc. 17th Int. Conf. on Software Engineering (ICSE'95)*, pages 73–82, 1995.

[10] J. E. Cook and A. L. Wolf. Event-based detection of concurrency. In *Proc. 6th Int. Symposium on the Foundations of Software Engineering (FSE'98)*, pages 35–45, 1998.

[11] A. Kayed and R. M. Colomb. Extracting ontological concepts for tendering conceptual structures. *Data & Knowledge Engineering*, 40(1):71–89, 2002.

[12] C. Makris, Y. Panagis, E. Sakkopoulos and A. Tsakalidis. Category ranking for personalized search. *Data & Knowledge Engineering*, 60(1):109–125, 2007.

[13] A. K. A de Medeiros, B. F. van Dongen, W. M. P. van der Aalst, and A. J. M. M. Weijters. Process mining: Extending the α-algorithm to mine short loops. Technical report, University of Technology, Eindhoven, 2004. BETA Working Paper Series, WP 113.

[14] A. K. A. de Medeiros, A. J. M. M. Weijters and W. M. P. van der Aalst Genetic Process Mining: An Experimental Evaluation. In *Data Mining and Knowledge Discovery*, 14 (2), pages 245-304, 2007.

[15] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[16] A. Gionis, D. Gunopulos, and N. Koudas. Efficient and tunable similar set retrieval. In *Proc. of the 2001 ACM SIGMOD Conference (SIGMOD01)*, pages 247–258, 2001.

[17] G. Greco, A. Guzzo, G. Manco, and D. Saccà. Mining and reasoning on workflows. *IEEE Trans. on Data and Knowledge Engineering*, 17(4):519–534, 2005.

[18] G. Greco, A. Guzzo, and L. Pontieri. Mining hierarchies of models: From abstract views to concrete specifications. In *Proc. 3rd Intl. Conf. on Business Process Management (BPM'05)*, pages 32–47, 2005.

[19] G. Greco, A. Guzzo, L. Pontieri, and D. Saccà. Discovering expressive process models by clustering log traces. *IEEE Trans. on Knowledge and Data Engineering*, 18(8):1010–1027, 2006.

[20] M. Hammori, J. Herbst, and N. Kleiner. Interactive workflow mining – requirements, concepts and implementation. *Data & Knowledge Engineering*, 56(1):41–63, 2006.

[21] J. Han, J. Pei, and Y. Yi. Mining frequent patterns without candidate generation. In *Proc. Int. ACM Conf. on Management of Data (SIGMOD'00)*, pages 1–12, 2000.

[22] J. Herbst. Dealing with concurrency in workflow induction. In *Proc. European Concurrent Engineering Conference*, 2000.

[23] J. Herbst and D. Karagiannis. Integrating machine learning and workflow management to support acquisition and adaptation of workflow models. *Journal of Intelligent Systems in Accounting, Finance and Management*, 9:67–92, 2000.

[24] J. Herbst and D. Karagiannis. Workflow mining with InWoLvE. *Computers in Industry. Special Issue: Process/Workflow Mining*, 53(3):245–264, 2003.

[25] IDS Scheer. ARIS Process Performance Manager (ARIS PPM): Measure, analyzeand optimize your business process performance (whitepaper).

[26] S. Junginger, H. Kuhn, R. Strobl, and D. Karagiannis. Ein geschafts-prozessmanagement-werkzeug der nachsten generation - ADONIS: Konzeption und anwendungen. *Wirtschaftsinformatik*, 42(3):392–401, 2000.

[27] J. Lee and G. M. Wyner. Defining specialization for dataflow diagrams. *Information Systems*, 28(6):651–671, 2003.

[28] N. Lesh, M. J. Zaki, and M. Ogihara. Mining features for sequence classification. In *Proc. 6th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD'00)*, pages 342–346, 1999.

[29] D.-R. Liu and M. Shen. Workflow modeling for virtual processes: an order-preserving process-view approach. *Information Systems*, 28:505–532, 2003.

[30] T. W. Malone, K. Crowston, J. Lee, B. Pentland, C. Dellarocas, G. Wyner, J. Quimby, C. Osborne, A. Bernstein, G. Herman, M. Klein, and E. O'Donnell. Tools for inventing organizations: Toward a handbook of organizational processes. *Management Science*, 45(3):425–443, 1999.

[31] H. Motoda and H. Liu. Data reduction: feature selection. *Handbook of data mining and knowledge discovery*, pages 208–213, 2002.

[32] P. Muth, J. Weifenfels, M. Gillmann, and G. Weikum. Integrating light-weight workflow management systems within existing business environments. In *Proc. 15th IEEE Int. Conf. on Data Engineering (ICDE'99)*, pages 286–293, 1999.

[33] B. Padmanabhan and A. Tuzhilin. Small is beautiful: discovering the minimal set of unexpected patterns. In *Proc. 6th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD'00)*, pages 54–63, 2000.

[34] A. Rozinat and W. M. P. van der Aalst Conformance Checking of Processes Based on Monitoring Real Behavior. In *Information Systems*, 33(1):64–95, 2008.

[35] G. Schimm. Mining most specific workflow models from event-based data. In *Proc. of Int. Conf. on Business Process Management*, pages 25–40, 2003.

[36] H. Schuldt, G. Alonso, C. Beeri, and H. Schek. Atomicity and isolation for transactional processes. *ACM Trans. Database Syst.*, 27(1):63–116, 2002.

[37] M. Stumptner and M. Schrefl. Behavior consistent inheritance in UML. In *Proc. 19th Int. Conf. on Conceptual Modeling (ER 2000)*, pages 527–542, 2000.

[38] M. Stumptner and M. Schrefl. Behavior consistent refinement of object life cycles. *ACM Transactions on Software Engineering and Methodology*, 11(1):92–148, 2002.

[39] M.-C. Tseng, and W.-Y. Lin. Efficient mining of generalized association rules with non-uniform minimum support. *Data & Knowledge Engineering*, 62(1):41–64, 2007.

[40] Y. Tzitzikas, A. Analyti, N. Spyratos and P. Constantopoulos. An algebra for specifying valid compound terms in faceted taxonomies. *Data & Knowledge Engineering*, 62(1):1–40, 2007.

[41] W. M. P. van der Aalst. The application of Petri nets to worflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

[42] W. M. P. van der Aalst, J. Desel, and E Kindler. On the semantics of EPCs: A vicious circle. In *Proc. EPK 2002: Business Process Management using EPCs*, pages 71–80, 2002.

[43] W. M. P. van der Aalst, A. Hirnschall, and H. M. W. Verbeek. An alternative way to analyze workflow graphs. In *Proc. 14th Int. Conf. on Advanced Information Systems Engineering*, pages 534–552, 2002.

[44] W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow mining: A survey of issues and approaches. *Data & Knowledge Engineering*, 47(2):237–267, 2003.

[45] W. M. P. van der Aalst and K. M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.

[46] W. M. P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(9):1128–1142, 2004.

[47] B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and W. M. P. van der Aalst. The ProM framework: A new era in process mining tool support. In *Proc. of 26th International Conference on Applications and Theory of Petri Nets (ICATPN'05)*, pages 444–454, 2005.

[48] A. J. M. M. Weijters and W. M. P. van der Aalst. Rediscovering workflow models from event-based data using Little Thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, 2003.

[49] L. Wen and J. Wang and J.-G. Sun Detecting Implicit Dependencies Between Tasks from Event Logs. *Proc. of the 8th Asia-Pacific Web Conference (APWeb 2006)*, pages 591–603, 2006.

[50] O. R. Zaïane. Building virtual web views. *Data & Knowledge Engineering*, 39(2):143–163, 2001.

[51] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search - The Metric Space Approach*. Springer, 2006.

A Computational Issues on Algorithm `BuildTaxonomy`

Lemma 12 *Let n be the number of (basic) activities, and m be an upper bound to the number of activities that must be abstracted by procedure* `getBestAbstraction` *(i.e., the size of the set S it takes in input). Then, every computation of procedure* `getBestAbstraction` *during the restructuring of a process hierarchy for P requires $O(m^2 \times n \times log(n))$ time.*

Proof First, we note that for any workflow schema produced in our approach (in both the mining and the restructuring phases), the number of activities is always $O(n)$. The same upper bound applies to the size the relation $\mathcal{I}sa$, since it encodes a number of trees, whose leaves correspond to the (basic) activities of P, and each of these activities may appear in one tree at most (i.e., it cannot have more than one parents).

Let us now examine the computation of all the functions defined in Section 4.1 and Section 4.2 that are used in the procedure `getBestAbstraction`. Notice that, even though not explicitly specified in Figure 5, we here assume that function $impl^{\mathcal{D}}$ is kept materialized and that suitable indexing structures are employed for directly (i.e., in constant time) retrieving the "children" and "fathers" of each activity according to either $\mathcal{I}sa$ or $\mathcal{P}artOf$, as well as its (basic) implied activities. As a matter of facts, we can compute the values of function $impl^{\mathcal{D}}$ incrementally, by first assigning (at the starting of algorithm `BuildTaxonomy`) an empty set to each activity in the schema—which actually contains basic activities only, while the abstraction dictionary is still empty. Each time two activities m_1 and m_2 are merged into an activity p in the procedure `AbstractActivities`, it is possible to update only the implied activities of p (which might actually coincide with m_1 or m_2) as follows: $impl^{\mathcal{D}}(p) := impl^{\mathcal{D}}(m_1) \cup impl^{\mathcal{D}}(m_1) \cup \{m_1\} \cup \{m_2\}$. Since each of these sets can consists of n (basic) activities at most, $O(n)$ time is enough to perform the above computation, assuming that all of them are kept ordered. Function $msg^{\mathcal{D}}$ can be evaluated in $O(n)$, as it substantially requires a visit over the tree associated with the $\mathcal{I}sa$ relation, whose size is $O(n)$ indeed. A similar fact holds for the similarity function $sim^{\mathcal{D}}_G$, which essentially performs a bottom-up scan over $\mathcal{I}sa$ links. Conversely, the main computational burden involved in the evaluation of function $sim^{\mathcal{D}}$, on two given activities a and b, arises from computing the intersection (actually involved in coefficient β) between the sets of activities implied by a and b, respectively. Since both these sets contain at most n elements, and they are not ensured to be ordered, this computation can be done in $O(n \times log(n))$. Similar considerations apply to the case of function sim^E, which again involves to perform intersection-based pairwise comparisons over a (fixed) number of sets (namely, $\mathcal{P}^E_a$, $\mathcal{P}^E_b$, $\mathcal{S}^E_a$, $\mathcal{S}^E_b$). Since the size of any of these sets is $O(n)$, the computation of sim^E takes $O(n \times log(n))$. Finally, by putting all the above results together, we observe

that the computation of function *score* (Line b1 in Figure 5) for two activities can be done in $O(n \times log(n))$.

In order to compute the overall similarity score for all pairs of activities in the set S (taken as input by `getBestAbstraction`, and containing m elements at most), $O(m^2 \times n \times log(n))$ steps are needed. In fact, this is also the complexity of the whole procedure, since all the remaining operations can be done in at most $O(n \times log(n))$. Indeed, as discussed above, this time is sufficient to evaluate all the functions used there (cf. Lines b7-b11, in Figure 5), as well as to verify whether the activities implied by one activity are a subset of those implied the other one (cf. Lines b9-b10, in the same figure). □

Lemma 13 *Let n be the number of (basic) activities, and m be an upper bound to the number of activities that must be abstracted by procedure* `AbstractActivities` *(i.e., the size of the set S it takes in input). Then,* `abstractActivities` *requires $O(m^3 \times n \times log(n))$ time.*

Proof The initial number of activities to be abstracted is $O(m)$, and progressively decrease at least by 1 at every step of the main loop – indeed, as long as `getBestAbstraction` returns a not null pair of activities, the set *ActuallyAbstracted* contains at least one of them (see Figure 5). At each step, the most expensive computation consists in evaluating function `getBestAbstraction`, which takes $O(m^2 \times n \times log(n))$ (see Lemma 12), since both procedures `deriveConstraints` and `arrangeEdges` only requires $O(n)$ time. As the total number of steps performed in procedure `abstractActivities` is $O(m)$, the total complexity of the procedure is $O(m^3 \times n \times log(n))$. □

Theorem 14 *Let $\mathcal{H}$ be a schema decomposition involving n (basic) activities. Let w be the number of schemas in $\mathcal{H}$ and k be its splitting factor, i.e., the maximum number of children for each non-leaf schema in $\mathcal{H}$. Furthermore, let m be an upper bound to the number of activities that must be abstracted in every computation of procedure* `AbstractActivities`*. Then, algorithm* `BuildTaxonomy` *on $\mathcal{H}$ correctly produces a taxonomy in $O(w \times k \times (n^2 + m^3 \times log(n))$ steps.*

Proof The main cost for computing the output taxonomy arises from applying procedure `generalizeSchemas` to all non-leaf nodes in the hierarchy $\mathcal{H}$, which obviously are $O(w)$. On the other hand, $O(k \times n^2)$ time is enough for performing all the operations in `generalizeSchemas` but the $k + 1$ calls to procedure `abstractActivities`. Therefore, based on the results of Lemma 13, the total cost of `generalizeSchemas` is $O(k \times (n^2 + m^3 \times log(n)))$, and hence the overall cost of algorithm `BuildTaxonomy` is $O(w \times k \times (n^2 + m^3 \times log(n))$. Finally, the correctness of `BuildTaxonomy` follows since, by definition of the procedure `abstractActivities`, each invocation of `generalizeSchemas`$(ChildSchemas, \mathcal{D})$ (cf. Line 5 in Figure 4) returns a workflow schema which generalizes each of the workflows stored in the set $ChildSchemas$. □

Remarks and Extensions. We leave the section by noticing that the running time of *BuildTaxonomy* (cf. Theorem 14) depends on parameters expressing the size of the schema hierarchy being restructured, and, primarily, on the number of activities that compose the process under analysis. It is worthwhile noticing that these parameters are (usually exponentially) lower that the size of the process log taken as input by the mining phase. This is especially true in the case of complex processes—like the ones our approach has been devised for—typically exhibiting a number of execution paths that is combinatorial on the number of activities. As a consequence, the application of algorithm `BuildTaxonomy` is likely not to affect the total computation time, which is rather mostly devoted for the preliminary mining of the schema hierarchy.

Incidentally, a major source of inefficiency in algorithm `BuildTaxonomy` is the way procedure `getBestAbstraction` decides the pair of activities to merge together, as well as the kind of abstraction to apply. In fact, the approach proposed above simply searches the pair of activities in S that achieves the highest similarity value of the function *score*, among all the m^2 pairs of such activities (where $m = |S|$). Notably, each computation of this score takes $O(n \times log(n))$ time. For example, one can think of optimizing such a search by resorting to efficient methods for the evaluation of similarity (self-)join queries (quite an extensive survey on this topic can be found, e.g., in [51]), which would enable us to retrieve pairs of similar enough activities in subquadratic time (w.r.t. m). Actually, many of such approaches mainly rely on the usage of indexing structures, conceived to store and retrieve objects based on some proper features of them that influence the chosen similarity measure. It is particularly interesting to this concern the body of research work done for the case of set-valued attributes (see, e.g., [16,5]), since all of the similarity measures adopted in our approach can be estimated, in an approximated way, by computing set-based similarities (i.e., the coefficient β defined in Section 4.2, a.k.a. Jaccard coefficient in the literature), over a proper set of features (i.e., predecessors/successors function sim^E, sub-activities for sim^D, and ancestors in the case of sim^G function [5]). However, we believe that any further detail on these issues is somewhat beyond the scope of the paper, since any possible improvement gained through such a method will not significantly impact on the performances of the entire approach to the discovery of process taxonomy, in most practical cases, as discussed above.

[5] $\overline{sim^G}(x, y)$ can be approximated roughly by comparing the sets $\{x\} \cup \{q \mid q \uparrow^{\mathcal{D}} x\}$ and $\{y\} \cup \{q \mid q \uparrow^{\mathcal{D}} y\}$ through coefficient $\beta(\cdot, \cdot)$