BPMN Miner: Automated Discovery of BPMN Process Models with Hierarchical Structure

Raffaele Conforti[a], Marlon Dumas[b], Luciano García-Bañuelos[b], Marcello La Rosa[a,c]

[a] *Queensland University of Technology, Australia*
[b] *University of Tartu, Estonia*
[c] *NICTA Queensland Lab, Australia*

Abstract

Existing techniques for automated discovery of process models from event logs generally produce flat process models. Thus, they fail to exploit the notion of subprocess as well as error handling and repetition constructs provided by contemporary process modeling notations, such as the Business Process Model and Notation (BPMN). This paper presents a technique for automated discovery of hierarchical BPMN models containing interrupting and non-interrupting boundary events and activity markers. The technique employs functional and inclusion dependency discovery techniques in order to elicit a process-subprocess hierarchy from the event log. Given this hierarchy and the projected logs associated to each node in the hierarchy, parent process and subprocess models are then discovered using existing techniques for flat process model discovery. Finally, the resulting models and logs are heuristically analyzed in order to identify boundary events and markers. By employing approximate dependency discovery techniques, it is possible to filter out noise in the event log arising for example from data entry errors or missing events. A validation with one synthetic and two real-life logs shows that process models derived by the proposed technique are more accurate and less complex than those derived with flat process discovery techniques. Meanwhile, a validation on a family of synthetically generated logs shows that the technique is resilient to varying levels of noise.

Key words: Process Mining, Automated Process Discovery, BPMN

1. Introduction

Process mining is a family of techniques to extract knowledge of business processes from event logs [1]. It encompasses, among others, techniques for automated discovery of process models. A range of such techniques exist that strike various trade-offs between accuracy and understandability of discovered models. However, the bulk of these techniques generate flat process models. When contextualized to the standard Business Process Model and Notation (BPMN), they produce flat BPMN models

Email addresses: raffaele.conforti@qut.edu.au (Raffaele Conforti), marlon.dumas@ut.ee (Marlon Dumas), luciano.garcia@ut.ee (Luciano García-Bañuelos), m.larosa@qut.edu.au (Marcello La Rosa)

consisting purely of tasks and gateways. In doing so, they fail to exploit BPMN's constructs for hierarchical modeling, most notably subprocesses and associated markers and boundary events.

To fill the gap, this paper presents an automated process discovery technique – namely BPMN Miner – that generates BPMN models with subprocesses, interrupting and non-interrupting boundary events, event subprocesses, and loop and multi-instance activity markers. An example of a BPMN model discovered by BPMN Miner is shown at the top of Figure 1. At the bottom is shown a flat BPMN model obtained from the Petri net discovered from the same log using the InductiveMiner plugin of the ProM framework [2].

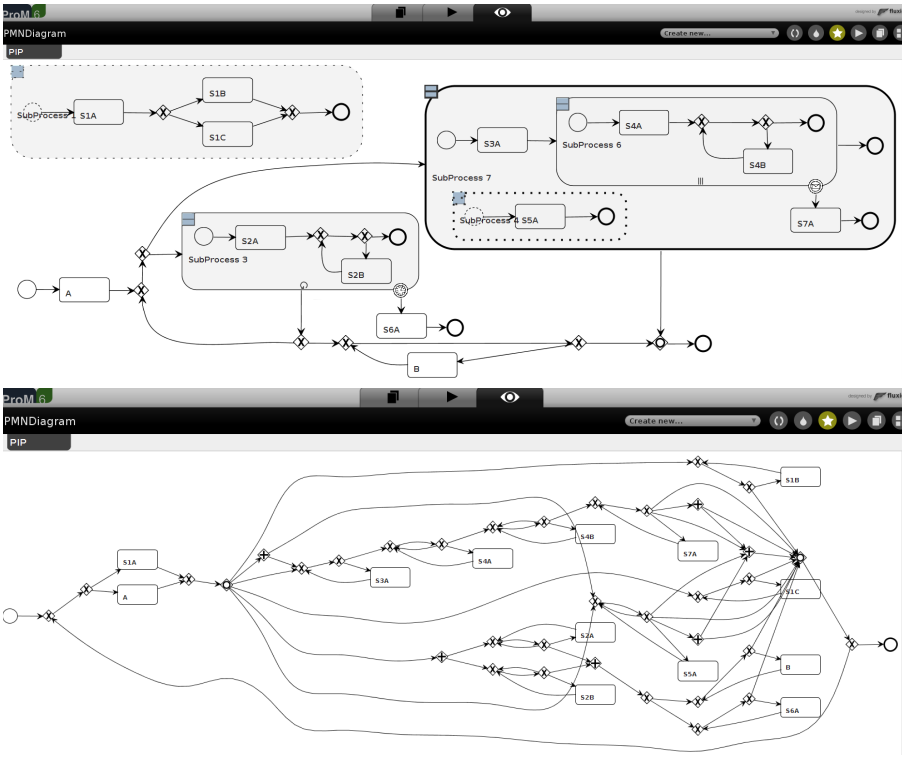


Figure 1: BPMN model obtained with and without applying the proposed technique on a synthetic log of an order-to-cash process (using InductiveMiner to generate flat models).

The technique takes as input a set of event records, each including a timestamp, an event type (indicating the task that generated the event), and a set of attribute-value pairs. Such logs can be extracted from appropriately instrumented information systems [1]. For example, we validated the technique using logs with these characteristics from an insurance claims system and a grant management system, while [3] discusses a log with similar characteristics from an Enterprise Resource Planning (ERP) system.

The proposal exploits three key ideas to address the problem of identifying subprocesses and hierarchical relations: (i) that the set of events of an event type can be seen as a relational table; (ii) that event types sharing a common primary key are likely to belong to the same (sub-)process; and (iii) that foreign keys between event types

are indicators of process-subprocess relations. Accordingly, the proposed technique employs existing functional and inclusion dependency discovery techniques in order to cluster event types into groups corresponding to parent processes and subprocesses. Given the resulting process hierarchy, the technique splits the log into parent process logs and subprocess logs and applies existing process model discovery techniques to each log so as to produce a flat model for each node in the hierarchy. Finally, the resulting models and logs are analyzed heuristically to identify boundary events, event subprocesses and markers.

In order to handle "noise" in the log arising from data quality issues (e.g. data entry errors or incompleteness) or from infrequent behavior, the proposed technique employs approximate dependency discovery techniques, followed by a series of filters designed to remove noise at the lowest possible level of granularity.

The technique has been validated on real-life and synthetic logs. The validation shows that, when combined with existing flat process discovery methods, the technique produces more accurate and less complex models than the corresponding flat models. Moreover, a validation on a family of synthetically generated logs demonstrates the resilience of the proposed technique to varying levels of noise.

The paper is a revised and extended version of a previous conference paper [4]. With respect to the conference version, the main reported extension is the ability to filter out noise in the log that may otherwise prevent the discovery of the process hierarchy, as well as the corresponding validation of the enhanced technique on noisy logs. Other ancillary extensions include a re-implementation of the prototype tool with optimizations leading to gains in execution times, as well as a release of the extended prototype both as a plugin in the ProM framework and as a standalone tool.

The paper is structured as follows. Section 2 discusses techniques for automated process discovery. Section 3 outlines the subprocess identification procedure for the case of perfect logs (no noise) while Section 4 extends this latter technique with the ability to filter out noise. Next, Section 5 presents heuristics to identify boundary events, event subprocesses and markers. Section 6 then presents the implementation of the proposed techniques while Sections 7 and 8 discuss the validation on noise-free and noisy logs respectively. Finally, Section 9 discusses threats to validity of the study while Section 10 concludes and discusses future work.

2. Background and Related Work

This section provides an overview of techniques for discovery of flat and hierarchical process models, and criteria for evaluation of such techniques used later in the paper.

2.1. Automated discovery of flat process models

Various techniques for discovering flat process models from event logs have been proposed [1]. The α-algorithm [5] infers ordering relations between pairs of events in the log (direct follows, causality, conflict and concurrency), from which it constructs a Petri net. The α-algorithm is sensitive to noise (e.g. incorrect or missing event records) and infrequent behavior, and cannot handle complex routing constructs. Weijters et al. [6] propose the Heuristics Miner, which extracts not only dependencies but also the frequency of each dependency. These data are used to construct a graph

of events, where edges are added based on frequency heuristics. Types of splits and joins in the event graph are determined based on the frequency of events associated with those splits and joins. This information can be used to convert the output of the Heuristics Miner into a Petri net. The Heuristics Miner is robust to noise due to the use of frequency thresholds. Van der Werf et al. [7] propose a discovery method where relations observed in the logs are translated to an Integer Linear Programming (ILP) problem. Finally, the InductiveMiner [2] aims at discovering Petri nets that are as block-structured as possible and can reproduce all traces in the log.

Only few techniques discover process models in high-level languages such as BPMN or Event-Driven Process Chains (EPCs). ProM's Heuristics Miner can produce flat EPCs from Heuristic nets, by applying transformation rules similar to those used when transforming a Heuristic net to a Petri net. A similar idea is implemented in the Fodina Heuristics Miner [8], which produces flat BPMN models. Apart from these, the bulk of process discovery methods produce Petri nets. Favre et al. [9] characterize a family of (free-choice) Petri nets that can be bidirectionally transformed into BPMN models. By leveraging this transformation, it is possible to produce flat BPMN models from discovery techniques that produce (free-choice) Petri nets.

Automated process discovery techniques can be evaluated along four dimensions: fitness (recall), appropriateness (precision), generalization and complexity [1]. Fitness measures to what extent the traces in a log can be parsed by a model. Several fitness measures have been proposed. For example, *alignment-based fitness* [10] measures the alignment of events in a trace with activities in the closest execution of the model, while the *continuous parsing measure* counts the number of missing activations when replaying traces against a heuristic net. *Improved Continuous Semantics* (ICS) fitness [11] optimizes the continuous parsing measure by trading off correctness for performance.

Appropriateness (herein called precision) measures the additional behavior allowed by a discovered model not found in the log. A model with low precision is one that parses a proportionally large number of traces that are not in the log. Precision can be measured in different ways. *Negative event precision* [12] works by artificially introducing inexistent (negative) events to enhance the log so that it contains both real (positive) and fake (negative) traces. Precision is defined in terms of the number of negative traces parsed by the model. Alternatively, *ETC* [13] works by generating a prefix automaton from the log and replaying each trace against the process model and the automaton simultaneously. ETC precision is defined in terms of the additional behavior ("escaping" edges) allowed by the model and not by the automaton.

Generalization captures how well the discovered model generalizes the behavior found in the log. For example, if a model discovered using 90% of traces in the log can parse the remaining 10% of traces in the logs, the model generalizes well the log.

Finally, process model complexity can be measured in terms of size (number of nodes and/or edges) or using structural complexity metrics proposed in the literature [14]. Empirical studies [14, 15, 16] have shown that, in addition to size, the following structural complexity metrics are correlated with understandability and error-proneness:

- Avg. Connector Degree (ACD): avg. number of nodes a connector is connected to.
- Control-Flow Complexity (CFC): sum of all connectors weighted by their poten-

tial combinations of states after a split.

- Coefficient of Network Connectivity (CNC): ratio between arcs and nodes.
- Density: ratio between the actual number of arcs and the maximum possible number of arcs in any model with the same number of nodes.

An extensive experimental evaluation [17] of automated process discovery techniques has shown that the Heuristics Miner provides the most accurate results, where accuracy is computed as the tradeoff between precision and recall (measured by means of *F-score*). Further, this method scales up to large real-life logs. The ILP miner achieves high recall – at the expense of a penalty on precision – but it does not scale to large logs due to memory requirements.

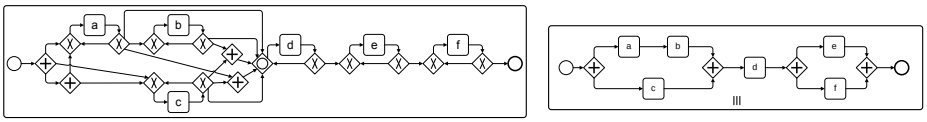
Moreover, these automated process discovery techniques present a substantial drop in accuracy when dealing with logs containing infrequent process behaviour. This problem, despite relevant, in not addressed in this work. In the context of this work noise is caused by data quality issues (e.g. data entry errors). We proposed an algorithm, capable of filtering out noise in the form of infrequent process behaviour, in a separate work [**?**].

2.2. Automated discovery of hierarchical process models

Although the bulk of automated process discovery techniques produce flat models, one exception is the two-phase mining approach [18], which discovers process models decomposed into sub-processes, each subprocess corresponding to a recurrent motif observed in the traces. The two-phase approach starts by applying pattern detection techniques on the event log in order to uncover *tandem arrays* (corresponding to loops) and *maximal repeats* (maximal common subsequence of activities across process instances). The idea is that occurrences of these patterns correspond to "footprints" left in the log by the presence of a subprocess. Once patterns are identified, their significance is measured based on their frequency. The most significant patterns are selected for subprocess extraction. For each selected pattern, all occurrences are extracted to produce subprocess logs. Each occurrence is then replaced by an *abstract activity*, which corresponds to a subprocess invocation in the parent process. This procedure leads to one parent process log and a separate log per subprocess. A process model can then be discovered separately for the parent process and for each subprocess. The procedure can be repeated recursively to produce process-subprocess hierarchies of longer depth.

A shortcoming of the two-phase approach is that it cannot identify subprocesses with (interrupting) boundary events, as these events cause the subprocess execution to be interrupted and thus the subprocess instance traces do not show up neither as tandem arrays nor maximal repeats. Secondly, in case multiple subprocess instances are executed in parallel, the two-phase approach mixes together in the same subprocess trace, events of multiple subprocess instances spawned by a given parent process instance. For example, if a parent process instance spawns three subprocess instances with traces $t_1 = [a_1, b_1, c_1, d_1]$, $t_2 = [a_2, c_2, b_2]$, and $t_3 = [a_3, b_3, c_3]$, the two-phase approach may put all events of t_1, t_2 and t_3 in the same trace, e.g. $[a_1, a_2, b_1, c_1, a_3, c_2, \ldots]$. When the resulting subprocess traces are given as input to a process discovery algorithm, the output is a model where almost every task has a self-loop and concurrency

is confused with loops. For example, given a log of a grant management system introduced later, the two-phase approach combined with Heuristics Miner produces the subprocess model depicted in Figure 2(a), whereas the subprocess model discovered using the Heuristics Miner after segregating the subprocess instances is depicted in Figure 2(b).



(a) Two-phase mining approach

(b) Two-phase mining with manual subprocess instance separation

Figure 2: Sample subprocess model discovered using the two-phase mining approach.

Another related technique [19] discovers Petri nets with *cancellation regions*. A cancellation region is a set P of places, where a given *cancellation* transition may fire, such that this transition firing leads to the removal of all tokens in P. The output is a *reset net*: a Petri net with *reset arcs* that remove tokens from their input place if any token is present. Cancellation regions are akin to BPMN subprocesses with interrupting events. However, generating BPMN models with subprocesses from reset nets is impossible in the general case, as cancellation regions may have arbitrary topologies, whereas BPMN subprocesses have a block-structured topology. Moreover, the reset nets produced by [19] may contain non-free-choice constructs that cannot be mapped to BPMN [9]. Finally, the technique in [19] does not scale up to logs with hundreds or thousands of traces due to the fact that it relies on analysis of the full state space.

Other techniques for discovering hierarchical collections of process models, e.g. [20], are geared towards discovering processes at different levels of generalization. They produce process hierarchies where a parent-child relation indicates that the child process is a more detailed version of the parent process (i.e. *specialization* relations). This body of work is orthogonal to ours, as we seek to discover *part-of* (parent-subprocess) relations.

The SMD technique [21] discovers hierarchies of process models related via specialization but also part-of relations. However, SMD only extracts subprocesses that occur in identical or almost identical form in two different specializations of a process.

Another related work is that of Popova et al. [22], which discovers process models decomposed into artifacts, where an artifact corresponds to the lifecycle of a business object in the process (e.g. a purchase order or invoice). This technique identifies artifacts in the event log by means of functional dependency and inclusion dependency discovery techniques. In this paper, we take this idea as starting point and adapt it to identify process hierarchies and then apply heuristics to identify boundary events and markers.

3. Identifying Subprocesses

In this section we outline a technique to extract a hierarchy of process models from a noise-free event log consisting of a set of traces. Each trace is a sequence of events,

where an event consists of an event type, a timestamp and a number of attribute-value pairs. Formally:

Definition 1 (Event (record)). *Let* $\{A_1,\ldots,A_n\}$ *be a set of* attribute names *and* $\{D_1,\ldots,D_n\}$ *a set of attribute* domains *where* D_i *is the set of possible values of* A_i *for* $1 \leq i \leq n$. *An* event $e = (et, \tau, v_1, \ldots, v_k)$ *consists of*

1. $et \in \Sigma$ *is the* event type *to which e belongs, where* Σ *is the set of all event types*
2. $\tau \in \Omega$ *is the event timestamp, where* Ω *is the set of all timestamps,*
3. *for all* $1 \leq i \leq k$ $v_i = (A_i, d_i)$ *is an attribute-value pair where* A_i *is an attribute name and* $d_i \in D_i$ *is an attribute value.*

Definition 2 (Log). *A* trace $tr = e_1 \ldots e_n$ *is a sequence of events sorted by timestamp. A log L is a set of traces. The set of events* E_L *of L is the union of events in all traces of L.*

The proposed technique is designed to identify logs of subprocesses such that:

1. There is an attribute (or combination of attributes) that uniquely identifies the trace of the subprocess to which each event belongs. In other words, all events in a trace of a discovered subprocess share the same value for the attribute(s) in question. Moreover, in this section we assume that in every given trace of a (sub-)process, there is at most one occurrence of an event of any given event type. This latter assumption is lifted in Section 4.
2. In every subprocess instance trace, there is at least one event of a certain type with an attribute or combination thereof uniquely identifying the parent process instance . In this section we further assume that other all events in the trace of a given sub-process refer to the same parent process instance via said attribute or combination of attributes. This latter assumption is lifted in Section 4.

These conditions match the notions of key (condition 1) and foreign key (condition 2) in relational databases. Thus, we use relational algebra concepts [23]. A *table* $T \subseteq D_1 \times \ldots \times D_m$ is a relation over domains D_i and has a *schema* $\mathscr{S}(T) = (A_1,\ldots,A_m)$ defining for each *column* $1 \leq i \leq m$ an *attribute name* A_i. The domain of an attribute may contain a "null" value $\perp$. The set of timestamps Ω does not contain $\perp$. For a given *tuple* $t = (d_1,\ldots,d_m) \in T$ and column $1 \leq i \leq m$, we write $t.A_i$ to refer to d_i. Given a tuple $t = (d_1,\ldots,d_m) \in T$ and a set of attributes $\{A_{i_1},\ldots,A_{i_k}\} \subseteq \mathscr{S}(T)$, we define $t[A_{i_1},\ldots,A_{i_k}] = (t.A_{i_1},\ldots,t.A_{i_k})$ Given a table T, a key of T is a minimal set of attributes $\{K_1,\ldots K_j\}$ such that $\forall t,t' \in T\, t[K_1,\ldots K_j] \neq t'[K_1,\ldots K_j]$ (no duplicate values on the key). A primary key is a key of a table designated as such. Finally, a foreign key linking table T_1 to T_2 is a pair of sets of attributes $(\{FK_1,\ldots,FK_j\},\{PK_1,\ldots,PK_j\})$ such that $\{FK_1,\ldots,FK_j\} \subseteq \mathscr{S}(T_1)$, $\{PK_1,\ldots,PK_j\}$ is primary key of T_2 and $\forall t \in T_1 \exists t' \in T_2\, t[FK_1,\ldots,FK_j] = t'[PK_1,\ldots,PK_j]$. The latter condition is an *inclusion dependency*.

Given the above, we seek to split a log into sub-logs based on process instance identifiers (keys) and references from subprocess to parent process instances (foreign keys). This is achieved by: (i) splitting the event types observed in the logs into clusters

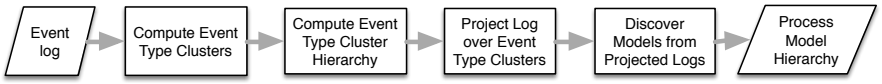


Figure 3: Procedure to extract a process model hierarchy from an event log.

based on keys; (ii) linking these clusters hierarchically via foreign keys; (iii) extracting one sub-log per node in the hierarchy; and (iv) deriving a process hierarchy mirroring the cluster hierarchy (Figure 3). Below we outline each step in turn.

Compute event type clusters. We start by splitting the event types appearing in the log into clusters such that all event types in a cluster (seen as tables consisting of event records) share a common key K. The intuition of the technique is that the key K shared by all event types in a cluster is an identifying attribute for all events in a subprocess. In other words, the set of instances of event types in a cluster that have a given value for K (e.g. $K = v$ for a fixed v), will form one trace of the (sub-)process in question. For example, in an order-to-cash process, all event types that have POID (Purchase Order Identifier) as primary key, will form the event type cluster corresponding to the root process. A given trace of this root process will consist of instances of event types in this cluster that share a given POID value (e.g. all events with POID = 122 for a trace). Meanwhile, event types that share LIID (Line Item Identifier) as primary key will form the event type cluster corresponding to a subprocess dealing with individual line items (say a "Handle Line Item" subprocess). A trace of this subprocess will consist of events of a trace of the parent process that share a given value of LIID (e.g. LIID = "122-3").[1]

To find keys of an event type *et*, we build a table consisting of all events of type *et*. The columns are attributes appearing in the attribute-value pairs of events of type *et*.

Definition 3 (Event type table). *Let et be an event type and* $\{e_1,\ldots,e_n\}$ *the set of events of type et in log L, i.e.* $e_i = (et, \tau_i, v_{i_1}, \ldots, v_{i_m})$ *where* $v_{i_j} = (A_j, d_{i_j})$ *and* A_j *is an attribute for* e_i*. The* event type table *for et in L is a table* $ET \subseteq (D_1 \cup \{\perp\}) \times \ldots \times (D_m \cup \{\perp\})$ *with* schema $\mathscr{S}(ET) = (A_1,\ldots,A_k)$ *s.t. there exists an entry* $t = (d_1,\ldots,d_m) \in ET$ *iff there exists an event* $e \in ET$ *where* $e = (et, \tau, (A_1,d_1),\ldots,(A_k,d_k))$ *s.t.* $d_i \in D_i \cup \{\perp\}$.

Events of a type *et* may have different attributes. Thus, the schema of the event type table consists of the union of all attributes that appear in events of this type in the log. Therefore there may be null values for some attributes of some events.

For each event type table, we seek to identify its key(s), meaning the attributes that may identify to which process instance a given event belongs to. To detect keys in event type tables, we use the TANE [24] algorithm for discovery of functional dependencies from tables. This algorithm finds all candidate keys, including composite keys. Given that an event type may have multiple keys, we need to select a primary one. Two options are available. The first is based on user input: The user is given the set of candidate keys discovered for each event type and designates one as primary – and in doing so chooses the subprocesses to be extracted. Alternatively, for full automation,

[1] It may happen alternatively that the key of the "Handle Line Item" subprocess is $(POID, LIID)$.

the lexicographically smallest candidate key of an event type is selected as the primary key $pk(ET)$, which may lead to event types not being grouped the way a user would have done so.

All event tables sharing a common primary key are grouped into an event type cluster. In other words, an event type cluster ETC is a maximal set of event types $ETC = \{ET_1, \ldots, ET_k\}$ such that $pk(ET_1) = pk(ET_2) = pk(ET_k)$.

Compute event type cluster hierarchy. We now seek to relate pairs of event clusters via foreign keys. The idea is that if an event type ET_2 has a foreign key pointing to a primary key of ET_1, every instance of an event type in ET_2 can be uniquely related to one instance of each event type in ET_1, in the same way that every subprocess instance can be uniquely related to one parent process instance.

We use the well-known SPIDER algorithm [25] to discover inclusion dependencies across event type tables. SPIDER identifies all inclusion dependencies between a set of tables, while we specifically seek dependencies corresponding to foreign keys relating one event type cluster to another. Thus we only retain dependencies involving the primary key of an event type table in a cluster corresponding to a parent process, and attributes in tables of a second cluster corresponding to a subprocess. The output is a set of candidate parent process-subprocess relations as follows.

Definition 4 (Candidate process-subprocess relation between clusters). *Given a log L, and two event type clusters ETC_1 and ETC_2, a tuple $(ETC_1, \mathscr{P}, ETC_2, \mathscr{F})$ is a candidate parent-subprocess relation if and only if:*

1. *$\mathscr{P} = pk(ETC_1)$ and $\forall ET_2 \in ETC_2, \exists ET_1 \in ETC_1 : ET_2[\mathscr{F}] \subseteq ET_1[\mathscr{P}]$ where $ET_1[\mathscr{P}]$ is the relational algebra projection of ET_1 over attributes in $\mathscr{P}$ and similar for $ET_2[\mathscr{F}]$. In other words, ETC_1 and ETC_2 are related, if every table in ETC_2 has an inclusion dependency to the primary key of a table in ETC_1 so that every tuple in ETC_2 is related to a tuple in ETC_1.*
2. *$\forall tr \in L \, \forall e_2 \in tr : e_2.et \in ETC_2 \Rightarrow \exists e_1 \in tr : e_1.et \in ETC_1 \wedge e_1[\mathscr{P}] = e_2[\mathscr{F}] \wedge (e_1.\tau < e_2.\tau \vee e_1 = e_2)$. This condition ensures that the direction of the relation is from the parent process to the subprocess by exploiting the fact that the first event of a subprocess instance must be preceded by at least one event of the parent process instance, or the first event of a subprocess is also the first event of the parent process instance.*

The candidate process-subprocess relations between clusters induces a directed acyclic graph. We extract a directed minimum spanning forest of this graph by extracting a directed minimum spanning tree from each weakly connected component of the graph. We turn the forest into a tree by merging all root clusters in the forest into a single root cluster. This leads us to a hierarchy of event clusters. The root cluster in this hierarchy consists of event types of the root process. The children of the root are event type clusters of second-level (sub-)processes, and so on.

Project logs over event type clusters. We now seek to produce a set of logs related hierarchically so that each log corresponds to a process in the envisaged process hierarchy. The log hierarchy will reflect one by one the event cluster hierarchy, meaning that each event type cluster is mapped to log. Thus, all we have to do is to define a function that maps each event type cluster to a log. This function is called log projection.

Given an event type cluster ETC, we project the log on this cluster by abstracting every trace in such a way that all events that are not instances of types in ETC are deleted, and markers are introduced to denote the first and last event of the log of a child cluster of ETC. Each of these child clusters corresponds to a subprocess and thus the markers denote the start and the end of a subprocess invocation.

Definition 5 (Projection of a trace over an event type cluster). *Given a log $L = \{tr_1, \ldots tr_n\}$, an event cluster ETC, the set of children cluster of ETC $children(ETC) = \{ETC_1, \ldots ETC_n\}$, and v a value of the key of the event cluster ETC, the projection of L over ETC_v is the log $L_{ETC_v} = \{tr'_1, \ldots tr'_n\}$ where tr'_k is the log obtained by replacing every event containing the key value v in tr_k that is also first event of a trace in the projected child log L_{ETC_i} by an identical event but with type $Start_{ETC_i}$ (start of cluster ETC_i), replacing every event containing the key value v in tr_k that is also last event of a trace in the projected child log L_{ETC_i} by an identical event but with type End_{ETC_i} (end of cluster ETC_i), and then removing from tr_k all other events not containing the key value v or of a type not in ETC.*

This recursive definition has a fix-point because the relation between clusters is a tree. We can thus first compute the projection of logs over the leaves of this tree and then move upwards in the tree to compute projected logs of parent trace clusters.

Generate process model hierarchy. Given the hierarchy of projected logs, we generate a hierarchy of process models isomorphic to the hierarchy of logs, by applying a process discovery algorithm to each log. For this step we can use any process discovery method that produces a flat process model (e.g. the Heuristics Miner). In the case of a process with subprocesses, the resulting process model will contain tasks corresponding to the subprocess start and end markers introduced in Definition 5.

Complexity. The complexity of the first step of the procedure is determined by that of TANE, which is in the size of the relation times a factor exponential on the number of attributes [24]. This translates to $O(|E_L| \cdot 2^a)$ where a is the number of attributes and $|E_L|$ is the number of events in the log. The second step's complexity is dominated by that of SPIDER, which is $O(a \cdot mlogm)$ where m is the maximum number of distinct values of any attribute [25]. If we upper-bound m by $|E_L|$, this becomes $O(a \cdot |E_L| log |E_L|)$. In this step, we also determine the direction of each primary-foreign key dependency. This requires one pass through the log for each discovered dependency, thus a complexity in $O(|E_L| \cdot k)$ where k is the number of discovered dependencies. If we define N as the number of event type clusters, $k < N^2$, this complexity becomes $O(|E_L| \cdot N^2)$. The third step requires one pass through the log for each event type cluster, hence $O(|E_L| \cdot N)$, which is dominated by the previous step's complexity. The final step is that of process discovery. The complexity here depends on the chosen process discovery method and we thus leave it out of this analysis. Hence, the complexity of subprocess identification is $O(|E_L| \cdot 2^a + a \cdot |E_L| log |E_L| + |E_L| \cdot N^2)$, not counting the process discovery step.

4. Robust subprocess discovery

A recurrent issue in the context of automated process discovery is to account for the effects of noise in the input log that may affect the quality of the discovered model,

measured in terms of accuracy and complexity metrics. Possible sources of noise are data quality issues arising from data entry errors or incompleteness – whether missing events or missing attribute values (e.g. "null" values). For example an invoice may wrongly point to a purchase order that is not recorded in the log. Another source of noise is exceptional behavior. For example, in an order-to-cash process, it is generally the case that an invoice is raised with reference to a purchase order. However, in exceptional cases it may be that an invoice is raised outside the context of a purchase order, for example to correct errors or miscommunications. In this case, there may be a set of invoice-related events in the log that are not related to the top-level order-to-cash process. Such noise, even when highly infrequent, would result in the technique presented in the previous section not discovering the parent-child relation between the order-to-cash process and the invoicing subprocess.

In this section we discuss the implications of noise on the technique presented in the previous section.

4.1. Types and implications of noise

The technique for extraction of process hierarchies presented in the previous section relies on the following assumptions:

1. *Perfect key assumption:* For every given trace of a subprocess to be discovered, there is an attribute or combination thereof (the "key") such that no two events of a given type belonging to the same subprocess trace share the same value on these attribute(s).

2. *Perfect and consistent foreign key assumption:* There is an event type belonging to a subprocess, such that every event of this type refers to exactly one parent process instance via a given attribute or combination thereof (the "foreign key").

Below we analyze the implications of these assumptions being violated and corresponding relaxations of these assumptions to address these implications.

Perfect key assumption. The first assumption ensures the existence of a key that can be used to determine to which (sub-)process instance a given event belongs to. However, it may happen that there is no perfect key identifying all events that belong to a given subprocess, due for example to exceptional behavior. Concretely, consider the case where of an order-to-cash process where the purchaser sends a PO to the supplier and the supplier sends back one "PO response" but in some cases, the supplier may send back multiple "PO responses" (e.g. because the first response was found to be incomplete). In this case, we will see in the log that there are in some cases multiple events of type "PO response" that refer to the same PO identifier, yet they are clearly all part of the same instance of the PO submission subprocess. The presence of multiple "PO response" events with the same PO identifier would mean that the PO identifier is not a key of the event type table "PO response" and thus the PO response would not be placed in the same event type cluster as other events related to the submission of the PO. Instead, the "PO response" would be placed in a separate event type cluster – possibly together with other event types related to the PO submission that lack the uniqueness property on attribute "PO identifier". This would lead to a "catch all" subprocess being created during the construction of the subprocess hierarchy, where for all such events would be put together.

This potential issue can be addressed within the framework of the TANE algorithm itself. Specifically, TANE can be configured (via a tolerance threshold) to discover keys that do not strictly satisfy the uniqueness property – a problem known as *approximate primary key discovery*. With reference to the previous example, this means we can set this tolerance threshold in TANE so that it classifies the PO identifier as a candidate key of event type "PO response" even though it is not a perfect key. The trade-off here is that TANE may start producing more candidate keys as the tolerance threshold is increased and thus more user intervention may be needed to select which candidate key should be made a primary key for each event type table.

Once an approximate primary key has been discovered for a given event type, the event log can be processed using the same techniques presented in the previous section (i.e. in the same way as if the primary key was an exact one). Event types with duplicate primary key values in an event type table will show up as "repeated tasks" and possibly embedded inside loops when the automated process discovery technique is applied to the projected log derived from the corresponding event type cluster.

Perfect and consistent foreign key assumption. The second assumption can be broken down into two sub-assumptions:

Perfect foreign key. That each event type ET in the cluster corresponding to a subprocess has a foreign key that links every event of ET to an event of an event type in the cluster corresponding to the parent process. This means that for every event type, there is a perfect inclusion dependency between this foreign key and the primary key of an event type in the parent process.

Consistent foreign keys. That these foreign keys are such that all events that belong to the same subprocess instance to the same parent process instance, as otherwise there is an ambiguity regarding the parent process instance to which a given subprocess instance is attached to.

The presence of noise – whether missing events in a parent process or incorrect or missing foreign key values – can break the perfect foreign key assumption. Accordingly, we relax this assumption by assuming that the inclusion dependency between the foreign key and the parent's primary key is imperfect. The corresponding data mining problem is known under the name of *approximate foreign key*. The SPIDER algorithm is unfortunately designed to discover exact foreign keys only. To discover approximate foreign keys, we employ the approximate foreign key discovery technique of Zhang et al. [26]. In addition to its relative scalability, this technique has the advantage over alternative ones: (i) it discovers both single-column and multi-column foreign keys; and (ii) it embeds a mechanism to reduce the number of "false positives", specifically fk/pk (foreign key–primary key) relations that are due simply to the domain of one column being included in the domain of another column.

In a nutshell, Zhang et al.'s method relies on a measure of *randomness* as an indicator of the appropriateness of a candidate single or multi-column foreign key. The intuition behind the method is that a foreign key should follow a probability distribution similar to that of the corresponding primary key, e.g., the values on the foreign key are uniformly taken from the values on the corresponding primary key. This property is tested for candidate fk/pk pairs by means of non-parametric statistical tests, tailored for handling multiple random variables (cf. table attributes) from different domains.

The output of the method is a set of candidate fk/pk relations with high levels of appropriateness, from which the user can select those that they consider meaningful from a domain perspective, which in our case means those that are likely to correspond to a process-subprocess relation. These pk/fk relations can then be used to construct the process hierarchy as outlined in Section 3.

The use of an approximate foreign key detection technique instead of SPIDER partly addresses the implications of the perfect key assumption, insofar as it allows us to detect process-subprocess relations even in the presence of noise. However, it is still necessary to filter out events (or entire subprocess traces) that violate inclusion dependencies between fk/pk pairs linking subprocesses to their parent processes. Moreover, the detection of approximate foreign keys does not ensure that the second sub-assumption above is fulfilled. It may still be that two events that share the same value for the primary key of a subprocess (i.e. they belong to the same subprocess instance) refer to different parent process instances. These two remaining concerns are addressed in the following sub-section.

4.2. Noise filtering

To handle noise in the log arising from imperfect or inconsistent fk/pk relations, we post-process each projected log in the hierarchy in order to remove noisy events at the lowest possible level of granularity. With reference to the procedure outlined in Figure 3, we introduce a noise filtering step between steps "Project Log over Event Type Clusters" and "Discover Models from Projected Logs". The noise filtering step consists of three filters:

1. A filter to remove traces containing at least an event referring to a parent process instance which does not exist.

2. A filter to remove subprocess traces containing at least an event referring to a parent process instance which does not exist.

3. A filter to remove events in a subprocess trace that refer to a different parent process instance than other events in the same subprocess trace.

The first filter (Algorithm 1) is applied to traces containing at least an event referring to a parent process which does not exist. This filter removes process instances containing noise, (i.e. a foreign key value that does not match any primary key value in the trace). For each trace t in a log, the algorithm scans sequentially each event in the trace. Whenever an event e_1 contains an attribute representing a foreign key fk, the algorithm checks if in the trace containing e_1 there exists an other event e_2 having an attribute representing a primary key pk which value is equal to the value of fk. If such event cannot be found the entire trace is discarded. To prevent the removal of a large number of process instances, the algorithm requires a threshold value (between 0 and 1). If the percentage of process instances removed exceeds the given threshold the algorithm returns the original log.

The second filter (Algorithm 2) removes subprocess instances containing noise in the form of events that do not refer to any parent process instance (i.e. a foreign key value that does not match any primary key value). The algorithm takes as input a subprocess log, its parent process log, and the primary key-foreign key pair relating the subprocess to the parent process. For each trace in the log, the algorithm scans

Algorithm 1: RemoveNoisyTraces

```
   input: original log L, tolerance value tv_noise, and minimum spanning tree tree
 1 L_new := {};
 2 removed := 0;
 3 while Noden in tree do
 4     FK := n.getFK();
 5     PK := n.getPK();
 6     PK_parent := n.getParent().getPK();
 7     foreach Trace t in L do
 8         PKs := {};
 9         L_new := L_new ∪ {t};
10         foreach Event e in t do
11             if there not exists an attribute A_j ∈ e such that A_j ∈ PK then
12                 if there exists an attribute A_k ∈ e such that A_k ∈ PK_parent then
13                     PKs := PKs ∪ {e.d_k};
14         foreach Event e in t do
15             if there exists an attribute A_k ∈ e such that e.d_k ∈ PKs then
16                 L_new := L_new \ {t};
17                 removed := removed + 1;
18 if removed > |L| · tv_noise then return L;
19 return L_new
```

sequentially each event in the trace of the subprocess. Whenever it finds an event containing the foreign key given in input, the algorithm checks if in the parent process log there exists an event containing the primary key given in input and if the value of these two attributes is equal. If an event in the parent process log cannot be found the subprocess instance is removed.

Finally, the third filter (Algorithm 3) removes, from a grandparent process instance, events referring to a parent process instance which does not exist. For each trace t of subprocess p_1, the algorithm checks if the trace contains an event referring to a parent subprocess p_2, child of p_1, which does not exist. If such an event exists the event is removed from the trace. Note that in the algorithm $PKs_{children}$ is treated as the function mapping PrimaryKeys to the sets of their possible values. Note that in Algorithm 3 $PKs_{children}$ is treated as the function mapping PrimaryKeys to the sets of their possible values.

Complexity. The complexity of the noise-resilient technique is the same as that of the base technique of Section 3 except for two points. First the term corresponding to the SPIDER algorithm – $O(a \cdot |E_L| log |E_L|)$ – is replaced by the complexity of the approximate foreign key detection algorithm. When applied to a given pair of attributes (a candidate pk/fk), the complexity of this latter algorithm is cubic on the size of the database, i.e. $O(|E_L|^3)$ in our case. If we only seek foreign keys of length one, the complexity of the approximate foreign key discovery step is thus $O(a \cdot |E_L|^3)$. If we seek composite foreign keys as well, the factor a is replaced by a binomial factor, which quickly makes the method impractical.

The second difference with respect to Section 3 is the addition of the three filters. These filters require one pass each through the (partitioned) log and thus add a linear term $O(|E_L|))$ to the overall complexity.

Algorithm 2: RemoveNoisySubtraces

input: original log L, log L_p, primary key PK, foreign key FK

$L_{new} := \{\}$;
foreach *Trace* t_p *in* L_p **do**
  $FKs := \{\}$;
  foreach *Event* e_p *in* t_p **do**
    if *there exists an attribute* $A_k \in e_p$ *such that* $A_k \in FK$ **then**
      $FKs := FKs \cup \{e_p.d_k\}$;
  if $|FKs| = 1$ **then**
    foreach *Trace* t *in* L **do**
      if $t_p \subset t$ **then**
        foreach *Event* e *in* t **do**
          if *there exists an attribute* $A_j \in e$ *such that* $A_j \in PK$ *and* $e.d_j \in FKs$ **then**
            $L_{new} := L_{new} \cup \{t_p\}$;
return L_{new}

Algorithm 3: RemoveNoisyEvents

input: log L_p, node of the minimum spanning tree related with the subprocess *node*

$L_{new} := \{\}$;
$PKs_{children} := \{\}$;
$PKsValues_{children} := \{\}$;
foreach *Node n in node.getChildren*() **do**
  $PKs_{children} := PKs_{children} \cup \{n.getPK()\}$;
  $PKsValues_{children} := PKsValues_{children} \cup \{(n.getPK(), n.getPKValues())\}$;
foreach *Trace* t_p *in* L_p **do**
  $t_{new} := \langle\rangle$;
  foreach *Event* e_p *in* t_p **do**
    $hasKey := false$;
    $add := false$;
    if *there exists an attribute* $A_k \in e_p$ *such that* $A_k \in PKs_{children}$ **then**
      $hasKey := true$;
      if $d_k \notin PKsValues_{children}(A_k)$ **then** $add := true$;
    if $\neg hasKey$ *or* add **then** $t_{new} := t_{new}^{\frown}\langle e_p\rangle$;
  if $t_{new} \neq \langle\rangle$ **then** $L_{new} := L_{new} \cup \{t_{new}\}$;
return L_{new}

5. Identifying Boundary Events, Event Subprocesses and Markers

This section presents heuristics to refactor a BPMN model by i) identifying interrupting boundary events, ii) assigning these events a type, iii) extracting event subprocesses, and iv) assigning loop and multi-instance markers to subprocesses and tasks. The overall refactoring procedure is given in Algorithm 4, which recursively traverses the process models hierarchy starting from the root model. This algorithm requires the root model, the set of all models *PS*, the original log L and the logs for all process models *LS*, plus parameters to set the tolerance of the heuristics as discussed later.

For each activity a of p that invokes a subprocess s (line 2), we check if the subprocess is in a self loop and if so we mark s with the appropriate marker and remove the loop structure (line 5 – refactoring operations are omitted for simplicity). We then check if the subprocess is triggered by an interrupting boundary event (line 6), in which case the subprocess is an exception flow of the parent process. If so, we attach an interrupting boundary event to the border of the parent process and connect the boundary event to the subprocess via an exception flow. Then we identify the type of boundary event, which can either be timer or message (line 8). Next, we check if the subprocess is an event subprocess (line 10). Finally, we check if the subprocess is multi-instance (lines 11 and 17), in which case we discover from the log the minimum and maximum number of instances. If activity a does not point to a subprocess (i.e. it is a task), we check if this is a loop (line 16) or multi-instance task (line 17), so that this task can be marked accordingly. Each of these constructs is identified via a dedicated heuristic.

Algorithm 4: UpdateModel

```
   input: Process model p, set of all process models PS, original log L, set of all process logs LS,
          tolerance values tv_int and tv_timer, percentages pv_timer and pv_MI
 1 foreach Activity a in p do
 2     if there exists a process s in PS such that label(a) = Start_s then
 3         s := updateModel(s, PS, L, LS, tv_int, tv_timer, pv_timer, pv_MI);
 4         L_p := getLog(p, LS);
 5         if s is in a self loop then mark s as Loop;
 6         if isInterruptingEvent(a, p, L_p, tv_int) then
 7             set s as exception flow of p via new interrupting event e_i;
 8             if isTimerInterruptingEvent(a, L_p, tv_timer, pv_timer) then mark e_i as Timer;
 9             else mark e_i as Message;
10         else if isEventSubprocess(a, p) then mark s as EventSubprocess of p;
11         if isMultiInstance(s, L, pv_MI) then
12             mark s as MI;
13             s_LB := discoverMILowerBound(s, L);
14             s_UB := discoverMIUpperBound(s, L);
15     else
16         if a is in a self loop then mark a as Loop;
17         if isMultiInstance(a, L, pv_MI) then
18             mark s as MI;
19             a_LB := discoverMILowerBound(a, L);
20             a_UB := discoverMIUpperBound(a, L);
21 return p
```

Identify interrupting boundary events. Algorithm 5 checks if subprocess s of p is triggered by an interrupting event. It takes as input an activity a_s corresponding to the

invocation of subprocess s. We check that there exists a path in p from a_s to an end event of p without traversing any activity or AND gateway (line 1). We count the number of traces in the log of p where there is an occurrence of a_s (line 5), and the number of those traces where a_s is the last event. If the latter number is at least equal to the former, we tag the subprocess as "triggered by an interrupting event" (line 8). The heuristic uses threshold tv_{int}. If $tv_{int} = 0$, we require all traces containing a_s to finish with a_s to tag s as triggered by an interrupting event, while if $tv_{int} = 1$, the path condition is sufficient.

Algorithm 5: isInterruptingEvent

```
   input: Activity a_s, process model p, log L_p, tolerance tv_int
1  if there exists a path in p from a_s to an end event of p without activities and AND gateways then
2      #BoundaryEvents := 0;
3      #Traces := 0;
4      foreach trace tr in L_p do
5          if there exists an event e_1 in tr such that e_1.et = label(a_s) then
6              if there not exists an event e_2 in tr such that e_2.et ≠ label(a_s) and e_2.τ ≥ e_1.τ then
                   #BoundaryEvents := #BoundaryEvents + 1;
7              #Traces := #Traces + 1;
8      if #BoundaryEvents ≥ #Traces · (1 − tv_int) then return true
9  return false
```

Identify interrupting boundary timer events. Algorithm 6 detects if a subprocess s of p is triggered by a timer boundary event. We first extract from the log of p all traces t containing executions of a_s (line 5). For each of these traces we compute the average time difference between the occurrence of a_s and that of the first event of the trace (lines 4-9). We then count the number of traces where this difference is equal to the average difference, modulo an error determined by the product of the average difference and tolerance value tv_{timer} (line 11). If the number of traces that satisfy this condition is greater than or equal to the number of traces containing an execution of a_s, we tag subprocess s as triggered by an interrupting boundary timer event (line 12). The heuristic can be adjusted using a percentage threshold pv_{timer} to allow for noise.

Algorithm 6: isTimerInterruptingEvent

```
   input: Activity a_s, log L_p, tolerance tv_timer, percentage pv_timer
1  #TimerEvents := 0;
2  timeDiff_tot := 0;
3  timeDifferences := ∅;
4  foreach trace tr in L_p do
5      if there exists an event e_1 in tr such that e_1.et = label(a_s) then
6          e_2 := first event of tr;
7          timeDiff_tot := timeDiff_tot + (e_1.τ − e_2.τ);
8          timeDifferences := timeDifferences ∪ {(e_1.τ − e_2.τ)};
9  timeDiff_avg := timeDiff_tot / |timeDifferences|;
10 foreach diff ∈ timeDifferences do
11     if timeDiff_avg − timeDiff_avg · tv_timer ≤ diff ≤ timeDiff_avg + timeDiff_avg · tv_timer then
           #TimerEvents := #TimerEvents + 1;
12 return #TimerEvents ≥ |timeDifferences| · pv_timer
```

Identify event subprocesses. A subprocess s of p is identified as an event subprocess if it satisfies two requirements: i) it needs to be repeatable (i.e. it has either been marked with a loop marker, or it is part of a while-do construct), and ii) can be executed in parallel with the rest of the parent process (either via an OR or an AND block).

Identify multi-instance activities. Algorithm 7 checks if a subprocess s of p is multi-instance. We start by retrieving all traces of p that contain invocations to subprocess s (line 5). Among them, we identify those where there are at least two instances of subprocess s executed in parallel (lines 6-7). As per Def. 5, an instance of s is delimited by events of types $Start_s$ and End_s sharing the same (PK, FK). Two instances of s are in parallel if they share the same FK and overlap in the log. If the number of traces with parallel instances is at least equal to a predefined percentage pv_{MI} of the total number of traces containing an instance of s, we tag s as multi-instance. Finally, we set the lower (upper) bound of instances of a multi-instance subprocess to be equal to the minimum (maximum) number of instances that are executed among all traces containing at least one invocation to s. Note that $e[PK]$ is the projection of event e over the primary key of $e.et$ and $e[FK]$ is the projection of e over the event type of the parent cluster of $e.et$.

Algorithm 7: isMultiInstance

```
   input: Subprocess s, original log L, percentage pv_MI
 1 if s is Loop then
 2     #Traces_MI := 0;
 3     #Traces := 0;
 4     foreach trace tr in L do
 5         if there exists an event e in tr such that e.et = Start_s then
 6             if there exist two events e1, e2 in t such that e1.et = Start_s, e2.et = Start_s,
               e1[PK] ≠ e2[PK] and e1[FK] = e2[FK] then
 7                 if there exists an event e3 in tr such that e3.et = End_s, e3[PK] = e1[PK],
                   e3[FK] = e1[FK], e1.τ ≤ e2.τ < e3.τ then
 8                     #Traces_MI := #Traces_MI + 1;
 9             #Traces := #Traces + 1;
10     return #Traces_MI ≥ #Traces · pv_MI;
11 return false
```

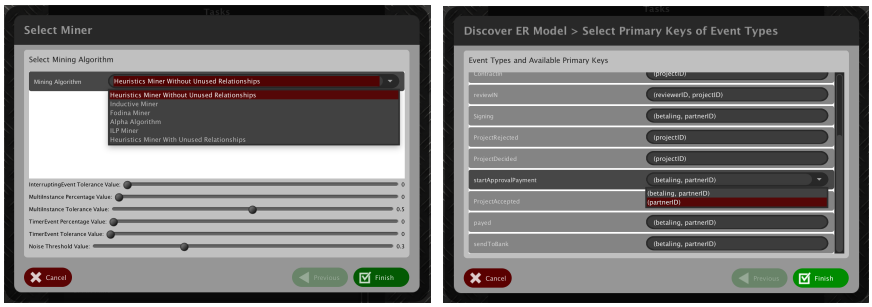
Complexity. Each heuristic used in Algorithm 4 requires one pass through the log and for each trace, one scan through the trace, hence a complexity in $O(|E_L|)$. The heuristics are invoked for each process model, thus the complexity of Algorithm 4 is $O(p \cdot |E_L|)$, where p is the number of process models. This complexity is dominated by that of subprocess identification.

6. Implementation

We implemented the proposed technique as a ProM plugin called *BPMN Miner*. This plugin implements the procedure for extracting a process model hierarchy from an event log outlined in Figure 3, and then utilizes the heuristics presented in Section 5 for identifying boundary events, event subprocesses and activity markers. While in principle our technique supports multi-rooted logs, i.e. logs that capture the events of

multiple business processes, the implementation is restricted to a log with a single-rooted hierarchy, i.e. it assumes that the log records the events of a single business process. Thus, the tool generates a single hierarchical BPMN process model.

Before starting the discovery procedure, the user is requested to choose an existing flat process model discovery method upon which the technique will be applied, and configure the parameters of the various heuristics described in Section 5, e.g. the tolerance level for interrupting events and multi-instance activities (see Figure 4(a)).



(a) Selecting flat discovery method (b) Overwriting a primary key

Figure 4: Configuration dialogs of the BPMN Miner plugin for ProM.

In order to compute the event type clusters the tool identifies a list of candidate event attributes for primary key detection.These are the event attributes that remain after filtering out those known not to be valid primary keys in a process log, e.g. activity name, timestamp, assigned resource and data in/out. The tool assigns to each activity the most likely event attribute. This assignment can then be modified by the user, which, prompted with the list of candidate event attributes for primary key detection,can remove false positives, if any, among the attributes identified. Upon completion, the TANE algorithm (cf. Section 3) is invoked to discover functional dependencies. The identified candidate primary keys are then shown to the user who can overwrite the selection in case of multiple candidates for a primary key (see Figure 4). Afterwards, the tool automatically performs the remaining steps of the procedure for subprocess extraction and applies the discovery heuristics. To increase performance, the log projection over event type clusters has been multi-threaded (one thread per cluster) as well as the removal of noisy subtraces using Algorithm 2 (one thread per subprocess log). An example of the result produced by BPMN Miner is the top model of Figure 1.

BPMN Miner supports the following flat process model discovery methods: Heuristics Miner and ILP as they provide the best results in terms of accuracy according to [17]; the InductiveMiner as an example of a method intended to discover block-structured models with high fitness; Fodina Heuristics Miner, which generates flat BPMN models natively; and the α-algorithm, as an example of a method suffering from low accuracy, according to [17].

As part of this work, we also implemented a number of utility plugins to:

- measure the complexity of a BPMN model;
- convert Petri nets to BPMN models in order to compare models produced by flat

discovery methods with those produced by BPMN Miner. For this, we adapted the Petri Net to EPCs converter available in ProM 5.2;

- convert BPMN models to Petri nets in order to compute accuracy. For this, we implemented the algorithm in [27];
- convert Heuristic Nets to BPMN models, in order to use the Heuristics Miner as a flat process model discovery method within our plugin. For this, we adapted the Heuristics Nets to EPCs converter available in ProM 5.2;
- simplify the final BPMN model by removing trivial gateways and turning single-activity subprocesses into tasks.

All plugins, together with the experimental results reported in this paper and the artificial logs used for the tests, are made available in the BPMN Miner package of the ProM 6 nightly-build.[2] Additionally, BPMN Miner is available as a standalone tool.[3]

7. Evaluation without noise

Using BPMNMiner, we conducted a number of tests to assess the benefits of our technique in terms of accuracy and complexity of the discovered process models. In this section we report the results of these tests on logs without noise; in the next section we discuss the results in the presence of different levels of noise.

7.1. Datasets and Setup

For the tests without noise, we used two real-life logs and one artificial log. The first log comes from a system for handling project applications in the Belgian research funding agency IWT (hereafter called FRIS), specifically for the applied biomedical research funding program (2009-12). This process exhibits two multi-instance subprocesses, one for handling reviews (each proposal is reviewed by at least five reviewers), the other for handling the disbursement of the grant, which is divided into installments. The second log (called Commercial) comes from a large Australian insurance company and records an extract of the instances of a commercial insurance claims handling process executed in 2012. This process contains a non-interrupting event subprocess to handle customer inquires, since these can arrive at any time while handling a claim, and three loop tasks to receive incoming correspondence, to process additional information, and to provide updates to the customer. Finally, the third log (called Artificial) is generated synthetically using CPN Tools,[4] based on the model of an order-to-cash process that has one example of each BPMN construct supported by our technique (loop marker, multi-instance marker, interrupting and non-interrupting boundary event and event subprocess). Table 1 shows the characteristics of the datasets, which differ widely in terms of number of traces, events and duplication ratio (i.e. the ratio between events and event types).

We measured accuracy and complexity of the models produced by BPMN Miner on top of the five process discovery methods supported by BPMN Miner (see Section 6

[2] `http://processmining.org`
[3] `http://apromore.org/platform/tools`
[4] `http://cpntools.org`

Log	Traces	Events	Event types	Duplication ratio
FRIS	121	1,472	13	113
Commercial	896	12,437	9	1,382
Artificial	3,000	32,896	13	2,530

Table 1: Characteristics of the event logs used for the validation.

for a rationale for the choice of these methods, we remind the reader that among these five process discovery methods only Fodina natively support BPMN models, while the other support Petri nets or Heuristics nets), and compared them to the same measures on the corresponding model produced by the flat discovery method alone.

Following [17], we measured accuracy in terms of *F-score* – the harmonic mean of recall (fitness – f) and precision (appropriateness – a), i.e. $2\frac{f \cdot a}{f+a}$. We measured complexity using size, CFC, ACD, CNC and density, as justified in Section 2.

We computed fitness using ProM's Alignment-based Conformance Analysis plugin, and appropriateness using the Negative event precision measure in the CoBeFra tool.[5] The choice of these two particular measures is purely based on the scalability of the respective implementations. These measures operate on a Petri net. We used the mapping in [27] to convert the BPMN models produced by BPMN Miner and by Fodina to Petri nets. For this conversion, we treated BPMN multi-instance activities as loop activities, since based on our tests, the alignment-based plugin could not handle the combinatorial explosion resulting from expanding all possible states of the multi-instance activities.We set all tolerance parameters of Algorithm 4 to zero.

7.2. Accuracy and Complexity

Table 2 shows the results of the measurements. Here and in the following tables we use the following abbreviations: H for Heuristics Miner, I for ILP, N for InductiveMiner, F for Fodina Heuristics Miner and A for the α-algorithm. When using BPMN Miner on top of these methods, we abbreviate it as BPMN$_X$ where X is the abbreviation of the method chosen.

From the results we observe that BPMN Miner consistency produces BPMN models that are more accurate and less complex than the corresponding flat models. The only exception is made by BPMN$_I$ on the artificial log. This model has a lower F-score than the one produced by the flat version of the ILP, despite improving on complexity. This is attributable to the fact that the artificial log exhibits a high number of concurrent events, which ILP turns into interleaving transitions in the discovered model (one for each concurrent event in the log). After subprocess identification, BPMN Miner replaces this structure with a set of interleaving subprocesses (each grouping two or more events), which penalizes both fitness and appropriateness.

In spite of the α-algorithm generally producing the least accurate models, we observe that BPMN$_A$ produces results comparable to those achieved using BPMN Miner on top of other discovery methods. In other words, BPMN Miner thins off differences between the flat discovery methods. This is attributable to the fact that, after subprocess extraction, the discovery of ordering relations between events is done on smaller sets

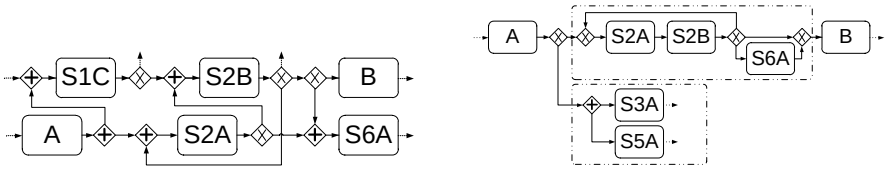
[5] `http://processmining.be/cobefra`

of event types (those within the boundaries of a subprocess). In doing so, behavioral errors also tend to get fixed.

Log	Method	Accuracy			Complexity				
		Fitness	Appropr.	F-score	Size	CFC	ACD	CNC	Density
FRIS	A	0.855	0.129	0.224	33	25	3.888	1.484	0.046
	$BPMN_A$	**0.917**	**0.523**	**0.666**	**32**	**21**	**3.4**	**1.25**	**0.040**
	F	**0.929**	0.354	0.512	35	85	8.5	2.828	0.083
	$BPMN_F$	0.917	**0.644**	**0.756**	**26**	**10**	**3.142**	**1.115**	**0.044**
	I	0.919	0.364	0.521	47	48	4.312	1.765	0.038
	$BPMN_I$	**0.987**	**0.426**	**0.595**	**42**	**34**	**3.652**	**1.428**	**0.034**
	H	0.567	0.569	0.567	31	26	3.25	1.290	**0.043**
	$BPMN_H$	**0.960**	**0.658**	**0.780**	**24**	**7**	**3.2**	**1.083**	0.047
	N	**1**	0.442	0.613	45	81	3.866	1.6	0.036
	$BPMN_N$	0.977	**0.525**	**0.682**	**39**	**28**	**3**	**1.230**	**0.032**
Commercial	A	0.703[6]	0.285	0.405	**19**	16	3.5	1.263	0.070
	$BPMN_A$	**1**	**0.382**	**0.552**	23	**11**	3.5	**1.173**	**0.053**
	F	0.928	0.398	0.557	**26**	**29**	4	**1.538**	0.061
	$BPMN_F$	**0.982**	**0.407**	**0.575**	37	35	**3.909**	1.540	**0.042**
	I	**1**	0.221	0.361	41	54	5.133	2.121	0.053
	$BPMN_I$	0.913	**0.264**	**0.409**	**34**	**31**	**4.105**	**1.558**	**0.047**
	H	0.399[6]	0.349	0.372	35	32	**3.083**	1.342	**0.039**
	$BPMN_H$	**0.935**	**0.425**	**0.584**	**17**	**2**	4	**1**	0.062
	N	1	0.448	0.618	25	21	4.571	1.680	0.070
	$BPMN_N$	1	**0.466**	**0.635**	**23**	**14**	**4**	**1.260**	**0.057**
Artificial	A	na	0.208	na	38	47	3.636	1.447	0.039
	$BPMN_A$	0.654	**0.222**	0.331	**33**	**11**	**3**	**1**	**0.031**
	F	na	0.295	na	**46**	53	3.677	1.543	0.034
	$BPMN_F$	0.813	**0.413**	0.548	47	**31**	**3.3**	**1.212**	**0.026**
	I	**0.969**	**0.331**	**0.493**	74	130	7.068	2.982	0.040
	$BPMN_I$	0.870	0.160	0.270	**37**	**21**	**4.2**	**1.216**	**0.033**
	H	na	0.290	na	49	47	3.17	1.387	0.028
	$BPMN_H$	0.908	**0.470**	0.619	**33**	**6**	**3**	**0.909**	0.028
	N	1	0.182	0.307	50	120	3.828	1.62	0.033
	$BPMN_N$	1	**0.362**	**0.531**	**45**	**18**	**3**	**1.022**	**0.023**

Table 2: Models' accuracy and complexity before and after applying BPMN Miner.

This is the case in three instances reported in our tests (A, F and H on Artificial which have "na" for fitness in Table 2), where the alignment-based fitness could not be computed because these flat models contained dead (unreachable) tasks and were not *easy sound* (i.e. did not have an execution sequence that completes by marking the end event with one token). An example of a fragment of such a model discovered by the Heuristics Miner alone is given in Figure 5(a). In these cases, the use of BPMN Miner resulted in simpler models without dead transitions (cf. Figure 5(b)).



(a) Heuristics Miner (b) BPMN Miner after Heuristics Miner

Figure 5: Behavioral error in a discovered flat model not present in the hierarchical one.

We also remark that, while density is inversely correlated with size (smaller models

[6]Over-approximation, as the fitness can only be computed on a fraction of the traces in the log

Prob.	Time [sec]		
	FRIS	Commercial	Artificial
A	0.01	0.04	0.16
$BPMN_A$	16.56	21.53	184.65
F	0.13	0.16	3.99
$BPMN_F$	16.61	21.71	276.54
I	1.20	4.02	11.67
$BPMN_I$	16.85	32.29	209.93
H	0.17	1.34	3.26
$BPMN_H$	17.16	23.88	311.40
N	0.09	0.06	3.36
$BPMN_N$	16.69	21.61	290.46

Table 3: Time performance in the absence of noise.

tend to be denser) [14], BPMN Miner produces smaller and less dense process models than those obtained by the flat process discovery methods. This is because it replaces gateway structures with subprocesses leading to less arcs, as evidenced by smaller ACD.

In summary, we obtained the best BPMN models using Heuristics Miner as the flat discovery method across all three logs. $BPMN_H$ achieved the highest accuracy and lowest complexity on FRIS and Artificial, while on Commercial it achieved the second highest accuracy (with the highest being $BPMN_N$) and the lowest complexity.

7.3. Time performance

We conducted our tests on an Intel Core i7 2.5 GHz with 16GB RAM, running MacOSX 10.10 and JVM 7 with 6GB of heap space. Table 3 shows the results. While timings increase substantially compared to those obtained by the flat discovery methods, the technique is still quite efficient. In fact, time performance ranged from a few seconds for small logs with few subprocesses (e.g. 17 seconds for $BPMN_I$ on FRIS) to a maximum of several minutes for the larger logs (e.g. 5 minutes for $BPMN_H$ on the Artificial log – on the same log the Heuristics Miner took 3.26 seconds). The bulk of time was spent on subprocesses identification, while the time required for identifying boundary events, event subprocesses and activity markers is negligible.

8. Evaluation on noisy logs

In this section we discuss the results of testing the technique on logs with noise.

8.1. Datasets and Setup

For the tests with noise, we created three log sets by injecting three different types of noise at various levels into the Artificial log, using CPN Tools. Specifically, in the first set (indicated with L_1) the noise introduced was in the form of foreign key attributes pointing to wrong primary keys. In the second set (L_2), foreign keys pointed to wrong or non-existing primary keys. Finally, in the third set (L_3) only the first event of each subprocess contained a foreign key, which pointed to a wrong or non-existing primary key in case of noise.

Each log set contained seven logs obtained by varying the likelihood of noise when generating the events of each subprocess. The likelihood ranged from 10% to 70% in

increments of 10%. The process model captured by the Artificial log counts a total of seven subprocesses. These subprocesses can be combined in different ways obtaining up to three levels of nesting and three subprocesses running in parallel.

We conducted the measurements using the Heuristics Miner as the flat discovery method, since this provides the best results in terms of model accuracy and complexity among the supported methods, as emerged from the tests reported in Section 7. We set all tolerance parameters of Algorithm 4 to zero, and the noise tolerance of Algorithm 1 to 70%. This means that Algorithm 1 is utilized to remove entire noisy traces unless the total number of traces being removed exceeds 70%, in which case Algorithm 2 is triggered to remove noisy subtraces.

8.2. Accuracy and complexity

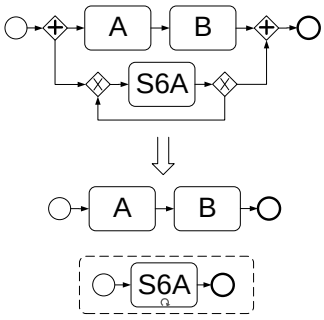
Table 4 shows the results of the measurements. For each log, the table indicates the noise probability used to generate that log, the percentage of events removed as a result of noise elimination, and the accuracy and complexity of the discovered BPMN models. The values for accuracy and complexity are averaged over three logs generated with the same noise probability. For reference, the first row of the table reports the results obtained on the log without noise. All experiments reported in Table 4 rely on the Heuristics Miner as the flat process model discovery method.

Log	**Prob.**	**% events filtered**	**Accuracy**			**Complexity**				
			Fitness	**Appropr.**	**F-score**	**Size**	**CFC**	**ACD**	**CNC**	**Density**
Artificial	0.0	-	0.908	0.470	0.619	33	6	3	0.909	0.028
L_1	0.1	54	0.998	0.531	0.693	33.667	8.333	3.083	0.949	0.029
	0.2	65	0.997	0.534	0.695	33	6	3	0.909	0.028
	0.3	70	0.997	0.522	0.685	35	13.667	3.233	1.048	0.031
	0.4	74	0.997	0.546	0.705	35	13.667	3.233	1.048	0.031
	0.5	76	0.992	0.586	0.737	35.667	15.333	3.067	1.075	0.031
	0.6	77	0.994	0.597	0.746	35	13.333	3	1.038	0.030
	0.7	78	0.995	0.564	0.719	35	13.333	3.083	1.116	0.030
L_2	0.1	52	0.997	0.533	0.695	33	6	3	0.909	0.028
	0.2	64	0.994	0.561	0.717	33	6.667	3	0.929	0.029
	0.3	71	0.997	0.521	0.684	35	13.667	3.233	1.048	0.031
	0.4	74	0.995	0.530	0.692	35	14.333	3.217	1.067	0.031
	0.5	77	0.996	0.573	0.726	35	13	3.067	1.029	0.030
	0.6	78	0.994	0.592	0.741	35	14.333	3.067	1.048	0.031
	0.7	79	0.995	0.560	0.717	37.667	19	3.143	1.094	0.030
L_3	0.1	75	0.997	0.615	0.760	33	6	3	0.909	0.028
	0.2	80	0.994	0.618	0.762	33	7.333	3	0.950	0.029
	0.3	84	0.997	0.653	0.788	35	13.667	3.233	1.048	0.031
	0.4	87	0.998	0.621	0.765	35	13	3.250	1.029	0.030
	0.5	89	0.994	0.665	0.797	35	14	3.150	1.057	0.031
	0.6	90	0.993	0.697	0.819	35	14	3	1.057	0.031
	0.7	92	0.994	0.671	0.800	35	13.667	3.067	1.048	0.031

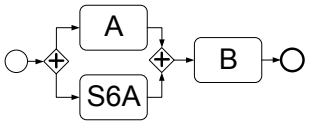
Table 4: Models' accuracy and complexity for varying noise probabilities

The results show that our technique can handle noise well: accuracy remains very high across all logs, despite model complexity tends to increase. More precisely, we observe an increase in fitness and appropriateness. The slight increase in fitness (from 0.909 to 0.99) is due to the fact that after removing noise the log has less distinct traces than the original log, which renders the heuristics used by the Heuristics Miner more effective. On the other hand, the increase in appropriateness is more pronounced (e.g. from 0.470 to 0.671 in the case of L_3). This is not only due to the reduced number of the traces of the log, but also to the fact that as noise increases, the heuristics employed

by our technique are less effective in discovering event subprocesses. This in turn leads to restricting the behavior of the model. Let us consider an example. Figure 7 shows an excerpt of the model discovered from the Artificial log without noise (a) and with noise (b). In Figure 6(a) subprocess S6A is correctly identified as a repeatable subprocess in parallel to tasks A and B. This structure will be replaced by an event subprocess. Figure 6(b) shows the model discovered after injecting noise into the log. In this model S6A is no longer identified as an event subprocess since the parallel relation between S6A and B is now less frequent. Thus, a trace where S6A follows B can no longer be replayed by the model.



(a) Without noise: S6A is correctly identified as an event subprocess.



(b) With noise: S6A is identified as a parallel subprocess to task A only.

Figure 6: The effects of noise on model appropriateness.

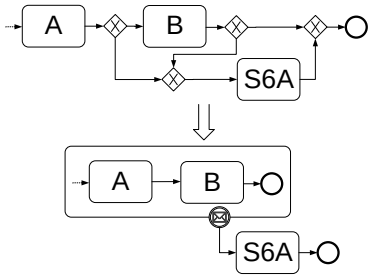
The increase in complexity is due to the heuristic for identifying boundary events. Similar to the case of event subprocesses, with the increase of noise this heuristic becomes less effective. When a boundary event cannot be identified, the model will contain more arcs and XOR gateways, and thus become larger and denser. Figure **??** illustrates this situation using an excerpt of the model discovered from the Artificial log before (a) and after injecting noise (b).

These side effects can be mitigated by increasing the tolerance parameters of Algorithm 4. However this will eventually impact on the accuracy of the model (lower precision).

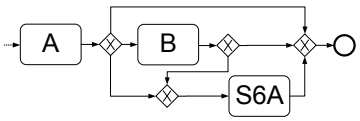
8.3. Time performance

Table 5 shows the execution times of the noise-resilient discovery technique. A comparison of this latter table with Table 3 (Artificial log) shows that noise removal leads to a visible effect on execution times, lifting them to up to 11 minutes. This effect can be explained by the fact that the complexity of the approximate foreign key discovery technique is higher than that of its exact counter-part. The execution times should be appreciated in the light of the size of the log under analysis (3,000 cases, 32,896 events) and the structural complexity of the model discovered (up to three levels of nesting and three parallel subprocesses). We did observe a correlation between increase in noise probability and decrease in performance.

The timings for L_3 are smaller than those for the other two logs, and they decrease when the noise probability increases (we remind the reader that from probability 0.3 Algorithm 2 is used with consequent drop in performance). This is because in this log,



(a) Without noise: S6A is correctly identified as the subprocess triggered by a boundary event.



(b) With noise: the boundary event is not identified, leading to a more complex model.

Figure 7: The effects of noise on model complexity.

Prob.	**Time [min]**		
	L_1	L_2	L_3
0.1	3.09	2.63	0.74
0.2	3.30	3.00	0.46
0.3	3.50	3.18	1.36
0.4	4.22	4.51	1.29
0.5	4.36	5.05	1.25
0.6	4.53	5.43	1.20
0.7	4.86	5.92	1.16

Table 5: Time performance in the presence of noise (Heuristics Miner used as flat discovery method).

the foreign key attribute is only present in the first event of each subprocess. Thus, all other events are removed from their parent process logs, which yields sensibly smaller logs to work with (cf. number of noisy events removed from L_3 vs. those removed from the other two log sets in Table 4).

9. Threats to validity

A potential threat to internal validity is the use process model complexity metrics as proxies for assessing understandability of the produced process models, as opposed to direct human judgement. However, as discussed in Section 2, the five chosen process model metrics have been empirically shown to be correlated with perceived understandability and error-proneness. Additionally, the observed reductions in complexity when applying our technique are very significant, which strengthens the likelihood that the process models produced by our technique would be perceived to be more understandable than the models generated by the corresponding flat process discovery algorithms applied to the entire log.

Another potential threat to internal validity comes from the fact that each time we generate one fresh log from the original log for each level of noise probability, as opposed to adding noise incrementally – e.g. to obtain a log with 30% noise probability we would add more noise to the log obtained with a probability of 20%. This choice may undermine the reliability of the results since it only tests the case where noise is rather uniformly distributed in the log, as opposed to cases where noise is concentrated

for example on a subset of subprocesses. To mitigate this effect, we generated three logs (from scratch) for each level of noise probability, and averaged the accuracy and complexity metrics over the three logs. We chose not to add noise incrementally and in a way that is concentrated on certain subprocesses, because in order to add noise to the right place (e.g. in a subprocess that is already affected by noise) we would need to use our own technique to identify the boundaries of subprocesses, and this would bias the experiments as we would be testing a technique against a result obtained by the same technique.

A threat to external validity is given by the small number of logs upon which we conducted our tests. While we chose only three logs, two of these are real-life logs and originate from two distinct domains (project management and insurance), while the artificial log was generated specifically to test all BPMN constructs supported by the proposed technique. While tests on further logs would be desirable – especially in conjunction with user validation – the three logs used in this study are representative of different sizes (number of traces and number of events), different duplication ratios and different types of processes.

10. Conclusion

We have shown that existing techniques for functional and inclusion dependency discovery provide a suitable basis for discovering hierarchical (BPMN) process models from event logs, both in the case of noise-free and noisy logs. We have also shown that given a process hierarchy and the corresponding logs for each (sub-)process in the hierarchy, it becomes possible to discover models containing advanced BPMN constructs, specifically interrupting and non-interrupting boundary events, loop and multi-instance markers. The empirical evaluation on two real-life and a family of synthetic logs shows that the proposed techniques lead to process models that are not only more modular, but also significantly more accurate and less complex than those obtained with traditional flat process discovery techniques.

A fundamental limitation of the proposed approach is that it requires the events in the log to include data attributes. Moreover, the set of attributes should include a primary key for every (sub-)process to be discovered as well as a foreign key for every process-subprocess relation. This is a limitation in two respects. First, some logs might not include said attributes. Second, one can think of subprocesses where these conditions do not hold, for example when subprocesses are used not to encapsulate activities related to a common business entity (with its own key) but rather to refactor block-structured fragments with loops – without there being a key associated to the loop body – or to refactor fragments shared across multiple process models (shared subprocesses). Thus, a potential avenue to enhance the technique is to combine it with the two-phase mining approach [18] and shared subprocess extraction as in SMD [21].

A direction for future work is to apply the technique on case studies involving larger real-life logs extracted for example from ERP systems with more complex data schemas than those found in the datasets used in this study. A related future work direction is to validate the understandability and usefulness of the produced process models with users.

Acknowledgments We thank Anna Kalenkova for her support with the ProM interface for BPMN and Pieter De Leenheer for enabling access to the FRIS dataset. This work

is partly funded by the EU FP7 Program (ACSI Project) and the Estonian Research Council. NICTA is funded by the Australian Department of Broadband, Communications and the Digital Economy and the Australian Research Council via the ICT Centre of Excellence program.

References

[1] W. van der Aalst, Process Mining - Discovery, Conformance and Enhancement of Business Processes, Springer, 2011.

[2] S. Leemans, D. Fahland, W. van der Aalst, Discovering block-structured process models from event logs – a constructive approach, in: Proc. of PETRI NETS, Vol. 7927 of LNCS, Springer, 2013, pp. 311–329.

[3] E. Nooijen, B. van Dongen, D. Fahland, Automatic discovery of data-centric and artifact-centric processes, in: Proc. of BPM Workshops, Springer, 2013, pp. 316–327.

[4] R. Conforti, M. Dumas, L. García-Bañuelos, M. L. Rosa, Beyond tasks and gateways: Discovering BPMN models with subprocesses, boundary events and activity markers, in: Proc. of BPM, Springer, 2014, pp. 101–117.

[5] W. van der Aalst, T. Weijters, L. Maruster, Workflow mining: Discovering process models from event logs, IEEE Trans. Knowl. Data Eng. 16 (9) (2004) 1128–1142.

[6] A. Weijters, J. Ribeiro, Flexible Heuristics Miner (FHM), in: Proc. of CIDM, IEEE, 2011, pp. 310–317.

[7] J. van der Werf, B. van Dongen, C. Hurkens, A. Serebrenik, Process discovery using integer linear programming, Fundam. Inform. 94 (3-4) (2009) 368–387.

[8] S. vanden Broucke, J. De Weerdt, J. Vanthienen, B. Baesens, Fodina: a robust and flexible heuristic process discovery technique, `http://www.processmining.be/fodina/`, last accessed: 03/27/2014 (2013).

[9] C. Favre, D. Fahland, H. Völzer, The relationship between workflow graphs and free-choice workflow nets, Information Systems 47 (0) (2014) 197–219, in press.

[10] A. Adriansyah, B. van Dongen, W. van der Aalst, Conformance checking using cost-based fitness analysis, in: Proc. of EDOC, IEEE, 2011, pp. 55–64.

[11] A. A. de Medeiros, Genetic process mining, Ph.D. thesis, Eindhoven University of Technology (2006).

[12] S. vanden Broucke, J. D. Weerdt, B. Baesens, J. Vanthienen, Improved artificial negative event generation to enhance process event logs, in: Proc. of CAiSE, Vol. 7328 of LNCS, Springer, 2012, pp. 254–269.

[13] J. Munoz-Gama, J. Carmona, A fresh look at precision in process conformance, in: Proc. of BPM, Vol. 6336 of LNCS, Springer, 2010, pp. 211–226.

[14] J. Mendling, H. Reijers, J. Cardoso, What Makes Process Models Understandable?, in: Proc. of BPM, Vol. 4714 of LNCS, Springer, 2007, pp. 48–63.

[15] E. Rolón Aguilar, J. Cardoso, F. García, F. Ruiz, M. Piattini, Analysis and validation of control-flow complexity measures with BPMN process models, in: Proc. of BPMDS and EMMSAD Workshops, Vol. 29 of LNBIP, Springer, 2009, pp. 58–70.

[16] H. Reijers, J. Mendling, A study into the factors that influence the understandability of business process models, IEEE T. Syst. Man Cy. A 41 (3) (2011) 449–462.

[17] J. D. Weerdt, M. D. Backer, J. Vanthienen, B. Baesens, A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs, Inf. Syst. 37 (7) (2012) 654–676.

[18] J. Li, R. Bose, W. van der Aalst, Mining context-dependent and interactive business process maps using execution patterns, in: Proc. of BPM Workshops, Vol. 66 of LNBIP, Springer, 2011, pp. 109–121.

[19] A. Kalenkova, I. Lomazova, Discovery of cancellation regions within process mining techniques, in: Proc. of CS&P Workshop, Vol. 1032 of CEUR Workshop Proceedings, CEUR-WS.org, 2013, pp. 232–244.

[20] G. Greco, A. Guzzo, L. Pontieri, Mining taxonomies of process models, Data Knowl. Eng. 67 (1) (2008) 74–102.

[21] C. Ekanayake, M. Dumas, L. García-Bañuelos, M. La Rosa, Slice, mine and dice: Complexity-aware automated discovery of business process models, in: Proc. of BPM, Vol. 8094 of LNCS, Springer, 2013, pp. 49–64.

[22] V. Popova, D. Fahland, M. Dumas, Artifact lifecycle discovery, CoRR abs/1303.2554, 2013.

[23] A. Silberschatz, H. Korth, S. Sudarshan, Database System Concepts, 4th Edition, McGraw-Hill Book Company, 2001.

[24] Y. Huhtala, J. Kärkkäinen, P. Porkka, H. Toivonen, TANE: An efficient algorithm for discovering functional and approximate dependencies, Computer Journal 42 (2) (1999) 100–111.

[25] J. Bauckmann, U. Leser, F. Naumann, Efficient and exact computation of inclusion dependencies for data integration, Technical Report 34, Hasso-Plattner-Institute (2010).

[26] M. Zhang, M. Hadjieleftheriou, B.-C. Ooi, C. Procopiuc, D. Srivastava, On multi-column foreign key discovery, PVLDB 3 (1) (2010) 805–814.

[27] R. Dijkman, M. Dumas, C. Ouyang, Semantics and analysis of business process models in bpmn, Information & Software Technology 50 (12) (2008) 1281–1294.