Beyond Tasks and Gateways: Discovering BPMN Models with Subprocesses, Boundary Events and Activity Markers

Raffaele Conforti[1], Marlon Dumas[2], Luciano García-Bañuelos[2], and Marcello La Rosa[1,3]

[1] Queensland University of Technology, Australia
{raffaele.conforti, m.larosa}@qut.edu.au
[2] University of Tartu, Estonia
{marlon.dumas, luciano.garcia}@ut.ee
[3] 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 BPMN models containing subprocesses, interrupting and non-interrupting boundary events and activity markers. The technique analyzes dependencies between data attributes attached to events in order to identify subprocesses and to extract their associated logs. 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. A validation with one synthetic and two real-life logs shows that process models derived using the proposed technique are more accurate and less complex than those derived with flat process discovery techniques.

1 Introduction

Process mining is a family of techniques to extract knowledge of business processes from event logs [19]. It encompasses, among others, techniques for automated discovery of process models. A range of such techniques exist that strike various tradeoffs 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 BPMN models consisting purely of tasks and gateways. In doing so, they fail to exploit BPMN's constructs for modular modeling, most notably subprocesses and associated markers and boundary events.

This paper presents an automated process discovery technique 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 using the implementation of the proposed technique in the ProM framework 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 [11].

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

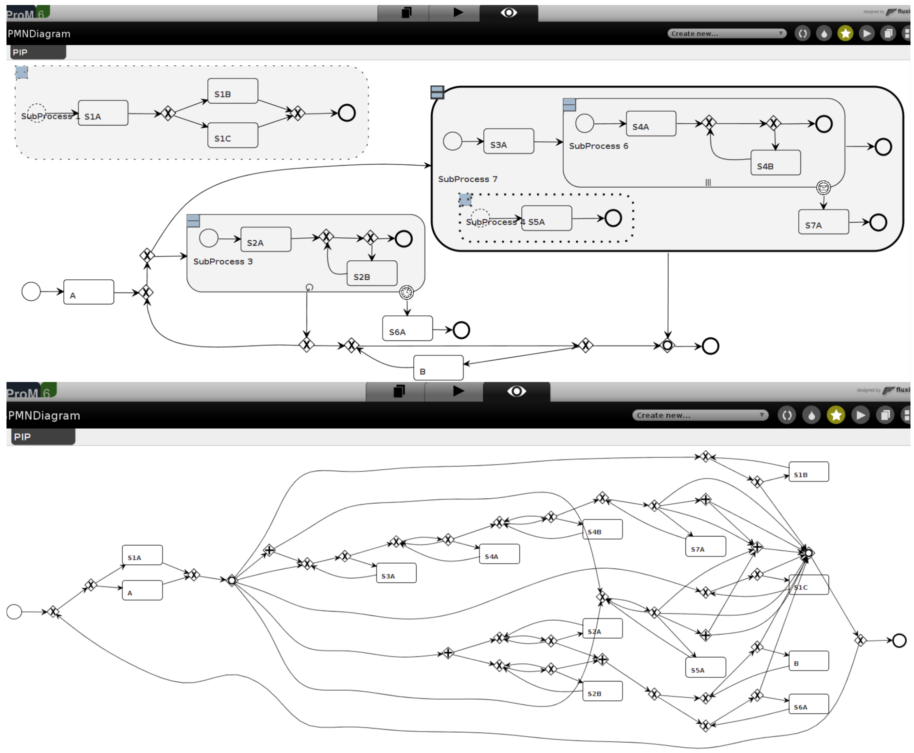


Fig. 1: BPMN model obtained with and without applying the proposed technique on a synthetic log of an order-to-cash process (using InductiveMiner to generate flat models).

pairs. Such logs can be extracted from appropriately instrumented information systems [19]. For example, we validated the technique using logs with these characteristics from an insurance claims system and a grant management system, while [15] discusses a log with similar characteristics from an Enterprise Resource Planning (ERP) system.

The technique analyzes dependencies between event attributes to identify subprocesses. Next, it splits the log into parent and subprocess logs and applies existing discovery techniques to each log to produce flat models. Finally, the resulting models and logs are analyzed heuristically to identify boundary events, event subprocesses and markers.

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.

The paper is structured as follows. Section 2 discusses techniques for automated process discovery. Section 3 outlines the subprocess identification procedure while Section 4 presents heuristics to identify boundary events, event subprocesses and markers. Section 5 discusses the validation and Section 6 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 [19]. The α-algorithm [20] 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, infrequent or incomplete behavior and cannot handle complex routing constructs. Weijters et al. [25] 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. [21] propose a discovery method where relations observed in the logs are translated to an Integer Linear Programming (ILP) problem. Finally, the InductiveMiner [11] 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 [22], which produces flat BPMN models. Apart from these, the bulk of process discovery methods produce Petri nets. Favre et al. [7] 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 [19]. 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* [1] 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 [4] 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* [23] 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* [14] 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 [13]. Empirical studies [2,13,17] have shown that, in addition to size, the following structural complexity metrics are correlated with understandability and error-proness:

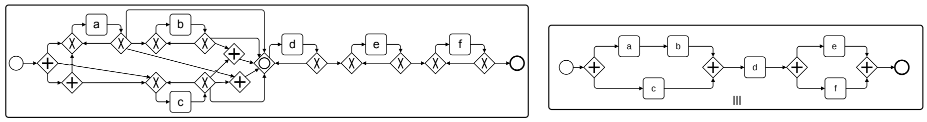
– Avg. Connector Degree (ACD): avg. number of nodes a connector is connected to.
– Control-Flow Complexity (CFC): sum of all connectors weighted by their potential combinations of states after a split.
– Coefficient of Network Connectivity (CNC): ratio between arcs and nodes.
– Density: ratio between 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 [24] 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. 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.

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 [12], 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 4(a), whereas the subprocess model discovered using the Heuristics Miner after segregating the subprocess instances is depicted in Figure 4(b).



(a) Two-phase mining approach

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

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

Another related technique [10] 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 [10] may contain non-free-choice constructs that cannot be mapped to BPMN [7]. Finally, the technique in [10] 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. [8], 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 [6] 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. [16], 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 an 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* $\{A_1,\dots,A_n\}$ *be a set of* attribute names *and* $\{D_1,\dots,D_n\}$ *a set of attribute* domains *where* D_i *is the set of possible values of* A_i *for* $1 \le i \le n$. *An* event $e = (et, \tau, v_1, \dots, v_k)$ *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* $1 \le i \le k$ $v_i = (A_i, d_i)$ *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.
2. In every subprocess instance trace, there is at least an event of a certain type with an attribute (or combination thereof) uniquely identifying the parent process instance.

These conditions match closely the notions of key and foreign key in relational databases. Thus, we use relational algebra concepts [18]. 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 subprocess-parent references (foreign keys). This is achieved by splitting event types into clusters based on keys, linking these clusters hierarchically via foreign keys, extracting one sub-log per node in the hierarchy, and deriving a process hierarchy mirroring the cluster hierarchy (Figure 3). Below we outline each step in turn.

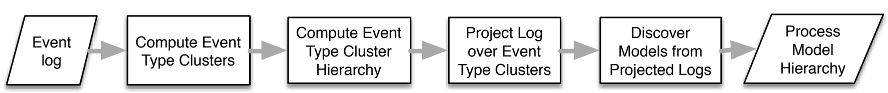


Fig. 3: Procedure to extract a process model hierarchy from an event log.

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 [9] 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, 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.

With scalability in mind, we use the SPIDER algorithm [3] 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

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

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. $\forall tr \in L \forall e_2 \in tr : e_2.et \in ETC_2 \Rightarrow \exists e_1 \in tr : e_1.et \in ETC_1 \wedge e_1[\mathscr{P}] = e_2[\mathscr{F}] \wedge e_1.\tau < e_2.\tau$. *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.*

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, and the set of children cluster of ETC $children(ETC) = \{ETC_1, \ldots ETC_n\}$, the projection of L over ETC is the log $L_{ETC} = \{tr'_1, \ldots tr'_n\}$ where tr'_k is the log obtained by replacing every event 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 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 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 [9]. 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 [3]. 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 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 1, 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 (line 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.

Identify interrupting boundary events Algorithm 2 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.

Identify interrupting boundary timer events Algorithm 3 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

Algorithm 1: 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}

foreach *Activity a in p* **do**
- **if** *there exists a process s in PS such that* $label(a) = Start_s$ **then**
  - s := updateModel(s, PS, L, LS, tv_{int}, tv_{timer}, pv_{timer}, pv_{MI});
  - L_p := getLog(p, LS);
  - **if** *s is in a self loop* **then** mark s as Loop;
  - **if** *isInterruptingEvent(a, p,* L_p*,* tv_{int}*)* **then**
    - set s as exception flow of p via new interrupting event e_i;
    - **if** *isTimerInterruptingEvent(a,* L_p*,* tv_{timer}*,* pv_{timer}*)* **then** mark e_i as Timer;
    - **else** mark e_i as Message;
  - **else if** *isEventSubprocess(a, p)* **then** mark s as EventSubprocess of p;
  - **if** *isMultiInstance(s, L,* pv_{MI}*)* **then**
    - mark s as MI;
    - s_{LB} := discoverMILowerBound(s, L);
    - s_{UB} := discoverMIUpperBound(s, L);
- **else**
  - **if** *a is in a self loop* **then** mark a as Loop;
  - **if** *isMultiInstance(a, L,* pv_{MI}*)* **then**
    - mark s as MI;
    - a_{LB} := discoverMILowerBound(a, L);
    - a_{UB} := discoverMIUpperBound(a, L);

return p

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.

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 4 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 2: isInterruptingEvent

input: Activity a_s, process model p, log L_p, tolerance tv_{int}
if *there exists a path in p from* a_s *to an end event of p without activities and AND gateways* **then**
  $\#BoundaryEvents := 0$;
  $\#Traces := 0$;
  foreach *trace tr in* L_p **do**
    if *there exists an event* e_1 *in tr such that* $e_1.et = label(a_s)$ **then**
      if *there not exists an event* e_2 *in tr such that* $e_2.et \neq label(a_s)$ *and* $e_2.\tau \geq e_1.\tau$ **then** $\#BoundaryEvents := \#BoundaryEvents + 1$;
      $\#Traces := \#Traces + 1$;
  if $\#BoundaryEvents \geq \#Traces \cdot (1 - tv_{int})$ **then return** *true*
return *false*

Algorithm 3: isTimerInterruptingEvent

input: Activity a_s, log L_p, tolerance tv_{timer}, percentage pv_{timer}
$\#TimerEvents := 0$;
$timeDiff_{tot} := 0$;
$timeDifferences := \varnothing$;
foreach *trace tr in* L_p **do**
  if *there exists an event* e_1 *in tr such that* $e_1.et = label(a_s)$ **then**
    e_2 := first event of *tr*;
    $timeDiff_{tot} := timeDiff_{tot} + (e_1.\tau - e_2.\tau)$;
    $timeDifferences := timeDifferences \cup \{(e_1.\tau - e_2.\tau)\}$;
$timeDiff_{avg} := timeDiff_{tot} / |timeDifferences|$;
foreach $diff \in timeDifferences$ **do**
  if $timeDiff_{avg} - timeDiff_{avg} \cdot tv_{timer} \leq diff \leq timeDiff_{avg} + timeDiff_{avg} \cdot tv_{timer}$ **then** $\#TimerEvents := \#TimerEvents + 1$;
return $\#TimerEvents \geq |timeDifferences| \cdot pv_{timer}$

Complexity Each heuristic used in Algorithm 1 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 1 is $O(p \cdot |E_L|)$, where p is the number of process models. This complexity is dominated by that of subprocess identification.

5 Validation

We implemented the technique as a ProM plugin called *BPMNMiner*. We also implemented utility plugins to: (i) measure model complexity; (ii) convert Petri nets to BPMN to compare models produced by flat discovery methods with those produced by BPMN Miner (adapted from the Petri Net to EPCs converter in ProM 5.2); (iii) convert BPMN models to Petri nets to compute accuracy (based on [5]); and (iv) simplify the final BPMN model by removing trivial gateways and turning single-activity subprocesses

Algorithm 4: isMultiInstance

input: Subprocess s, original log L, percentage pv_{MI}
if *s is* Loop **then**
  #Traces$_{MI}$:= 0;
  #Traces := 0;
  foreach *trace tr in L* **do**
    if *there exists an event e in tr such that* $e.et = Start_s$ **then**
      if *there exist two events* e_1, e_2 *in t such that* $e_1.et = Start_s$, $e_2.et = Start_s$, $e_1[PK] \neq e_2[PK]$ *and* $e_1[FK] = e_2[FK]$ **then**
        if *there exists an event* e_3 *in tr such that* $e_3.et = End_s$, $e_3[PK] = e_1[PK]$, $e_3[FK] = e_1[FK]$, $e_1.\tau \leq e_2.\tau < e_3.\tau$ **then**
          #Traces$_{MI}$:= #Traces$_{MI}$ + 1;
      #Traces := #Traces + 1;
  return *#Traces$_{MI}$* $\geq$ *#Traces* $\cdot pv_{MI}$;
return *false*

into tasks.[2] Using this implementation, we conducted tests to assess the benefits of the technique in terms of accuracy and complexity of discovered process models.

5.1 Datasets

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,[3] based on a 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).

5.2 Setup

We measured accuracy and complexity of the models produced by BPMN Miner on top of five process discovery methods, and compared them to the same measures on

[2] All plugins, the artificial log and the experimental results are in the BPMN Miner package of the ProM 6 nightly-build – `http://processmining.org`

[3] `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 event logs used for the validation.

the corresponding model produced by the flat discovery method alone. We selected the following flat discovery methods: Heuristics Miner (abbreviated as H) and ILP (I) as they provide the best results in terms of accuracy according to [24]; the InductiveMiner (N) 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 [24].

Following [24], 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.[4] 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 [5] 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 1 to zero.

5.3 Results

Table 2 shows the results of the measurements. 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 baseline 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 baseline methods. This is attributable to the fact that, after subprocess extraction, the discovery of ordering relations between events is done on smaller sets of event types (those within the boundaries of a subprocess). In doing so, behavioral errors also tend to get fixed.

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

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

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.

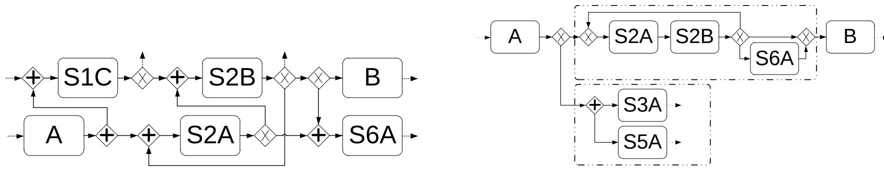
easy sound (i.e. did not have an execution sequence that completes by marking the end event with one token). An example of a fragment of such a model discovered by the Heuristics Miner alone is given in Figure 4(a). In these cases, the use of BPMN Miner resulted in simpler models without dead transitions (cf. Figure 4(b)).

We also remark that, while density is inversely correlated with size (smaller models tend to be denser) [13], 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 baseline 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.

We conducted our tests on an Intel Xeon 2.93GHz with 16GB RAM, running Windows Server 2008R2 and JVM 7 with 10GB of heap space. Time performance ranged from a few seconds for small logs with few subprocesses (e.g., 4sec for $BPMN_A$ on

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



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

Fig. 4: Behavioral error in a discovered flat model not present in the hierarchical one.

FRIS) to several minutes for the large log (max. 34.8min for BPMN$_H$ on Artificial while H on Artificial took 14.2sec). The bulk of time is spent in subprocesses identification, while the time required for identifying boundary events and markers is negligible.

6 Conclusion

We have shown that the proposed technique leads to process models that are not only more modular, but also more accurate and less complex than those obtained with traditional flat process discovery techniques. This is a step forward towards the development of methods for discovery of modular and rich business process models from event logs. Naturally, the proposal has its limitations. First, it requires logs with data attributes, such that the set of attributes includes keys to identify (sub)process instances, and foreign keys to identify relations between parent and child processes. One can think of subprocesses where this condition does not hold, for example when subprocesses are used not to encapsulate activities pertaining to a 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. Thus, a potential avenue to enhance the technique is to combine it with the two-phase mining approach [12] and shared subprocess extraction techniques as in SMD [6].

Secondly, it is assumed that data is of sufficient quality to discover the relevant functional and inclusion dependencies. In this respect, more noise-tolerant techniques for functional and inclusion dependency discovery could be employed, but the extent of required noise-tolerance needs to be evaluated against relevant datasets.

A direction for future work is to apply the technique on larger collections of logs, for example logs extracted from ERP systems, where there may be multiple keys for every entity associated with a process and associations may be more complex. A validation of the produced process models with actual users is also needed to assess usefulness.

Acknowledgments We thank Anna Kalenkova for her BPMN ProM interface 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. A. Adriansyah, B.F. van Dongen, and W.M.P. van der Aalst. Conformance checking using cost-based fitness analysis. In *Proc. of EDOC*. IEEE, 2011.
2. E. Rolón Aguilar, J. Cardoso, F. García, F. Ruiz, and M. Piattini. Analysis and validation of control-flow complexity measures with BPMN process models. In *Proc. of BPMDS and EMMSAD Workshops*, Springer, 2009.

3. J. Bauckmann, U. Leser, and F Naumann. Efficient and exact computation of inclusion dependencies for data integration. Technical Report 34, Hasso-Plattner-Institute, 2010.
4. A.K. Alves de Medeiros. *Genetic Process Mining*. PhD thesis, Eindhoven University of Technology, 2006.
5. R.M. Dijkman, M. Dumas, and C. Ouyang. Semantics and analysis of business process models in bpmn. *Information & Software Technology*, 50(12), 2008.
6. C.C. Ekanayake, M. Dumas, L. García-Bañuelos, and M. La Rosa. Slice, mine and dice: Complexity-aware automated discovery of business process models. In *Proc. of BPM*, Springer, 2013.
7. C. Favre, D. Fahland, and H. Völzer. The relationship between workflow graphs and free-choice workflow nets. *Information Systems*, 2014. In press.
8. G. Greco, A. Guzzo, and L. Pontieri. Mining taxonomies of process models. *Data Knowl. Eng.*, 67(1), 2008.
9. Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *Computer Journal*, 42(2), 1999.
10. A. Kalenkova and I.A. Lomazova. Discovery of cancellation regions within process mining techniques. In *Proc. of CS&P Workshop*, CEUR-WS.org, 2013.
11. S.J.J. Leemans, D. Fahland, and W.M.P. van der Aalst. Discovering block-structured process models from event logs – a constructive approach. In *Proc. of PETRI NETS*, Springer, 2013.
12. J. Li, R.P. Bose, and W.M.P. van der Aalst. Mining context-dependent and interactive business process maps using execution patterns. In *Proc. of BPM Workshops*, Springer, 2011.
13. J. Mendling, H.A. Reijers, and J. Cardoso. What Makes Process Models Understandable? In *Proc. of BPM*, Springer, 2007.
14. J. Munoz-Gama and J. Carmona. A fresh look at precision in process conformance. In *Proc. of BPM*, Springer, 2010.
15. E.H.J. Nooijen, B.F. van Dongen, and D. Fahland. Automatic discovery of data-centric and artifact-centric processes. In *Proc. of BPM Workshops*. Springer, 2013.
16. V. Popova, D. Fahland, and M. Dumas. Artifact lifecycle discovery. *CoRR abs/1303.2554*, 2013.
17. H.A. Reijers and J. Mendling. A study into the factors that influence the understandability of business process models. *IEEE T. Syst. Man Cy. A*, 41(3), 2011.
18. A. Silberschatz, H.F. Korth, and S. Sudarshan. *Database System Concepts, 4th Edition*. McGraw-Hill Book Company, 2001.
19. W.M.P. van der Aalst. *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.
20. W.M.P. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. *IEEE Trans. Knowl. Data Eng.*, 16(9), 2004.
21. J.M.E.M. van der Werf, B.F. van Dongen, C.A.J. Hurkens, and A. Serebrenik. Process discovery using integer linear programming. *Fundam. Inform.*, 94(3-4), 2009.
22. S.K.L.M. vanden Broucke, J. De Weerdt, J. Vanthienen, and B. Baesens. Fodina: a robust and flexible heuristic process discovery technique. `http://www.processmining.be/fodina/`. Last accessed: 03/27/2014.
23. S.K.L.M. vanden Broucke, J. De Weerdt, B. Baesens, and J. Vanthienen. Improved artificial negative event generation to enhance process event logs. In *Proc. of CAiSE*, Springer, 2012.
24. J. De Weerdt, M. De Backer, J. Vanthienen, and 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.
25. A.J.M.M. Weijters and J.T.S. Ribeiro. Flexible Heuristics Miner (FHM). In *Proc. of CIDM*, IEEE, 2011.