Automated Discovery of Structured Process Models: Discover Structured vs. Discover and Structure

Adriano Augusto[1,2], Raffaele Conforti[1], Marlon Dumas[3], Marcello La Rosa[1], and Giorgio Bruno[2]

[1] Queensland University of Technology, Australia
{a.augusto, raffaele.conforti, m.larosa}@qut.edu.au
[2] Politecnico di Torino, Italy
giorgio.bruno@polito.it
[3] University of Tartu, Estonia
marlon.dumas@ut.ee

Abstract. This paper addresses the problem of discovering business process models from event logs. Existing approaches to this problem strike various tradeoffs between accuracy and understandability of the discovered models. With respect to the second criterion, empirical studies have shown that block-structured process models are generally more understandable and less error-prone than unstructured ones. Accordingly, several automated process discovery methods generate block-structured models by construction. These approaches however intertwine the concern of producing accurate models with that of ensuring their structuredness, sometimes sacrificing the former to ensure the latter. In this paper we propose an alternative approach that separates these two concerns. Instead of directly discovering a structured process model, we first apply a well-known heuristic technique that discovers more accurate but sometimes unstructured (and even unsound) process models, and then transform the resulting model into a structured one. An experimental evaluation shows that our "discover and structure" approach outperforms traditional "discover structured" approaches with respect to a range of accuracy and complexity measures.

1 Introduction

Automated process discovery refers to a family of methods that generate a business process model from an event log [18]. An event log in this context is a set of traces, each consisting of a sequence of events observed within one execution of a process.

Existing automated process discovery methods strike various tradeoffs between accuracy and understandability [20]. In this setting, accuracy is commonly declined into three dimensions: (i) *fitness*: to what extent the discovered model is able to "parse" the traces in the log; (ii) *precision*: how much behavior is allowed by the model but not observed in the log; and (iii) *generalization*: to what extent is the model able to parse traces that, despite not being present in the input log, can actually be produced by the process under observation. Understandability on the other hand is commonly measured via size metrics (e.g. number of nodes) and structural complexity metrics. The latter quantify either the *amount of branching* in a process model or its degree of *structuredness* (the extent to which a model is composed of well-structured single-entry, single-exit components), which have been empirically shown to be proxies for understandability [11].

Inspired by the observation that structured process models may be more understandable than unstructured ones [6], several automated process discovery methods generate structured models by construction [10, 3, 12]. These approaches however intertwine the concern of accuracy with that of structuredness, sometimes sacrificing the former to achieve the latter. This paper obviates this tradeoff by presenting an automated process discovery method that generates structured models, yet achieves essentially the same fitness, precision and generalization as methods that generate unstructured models. The method follows a two-phased approach. In the first phase, a model is discovered from the log using a heuristic process discovery method that has been shown to consistently produce accurate, but potentially unstructured or even unsound models. In the second phase, the discovered model is transformed into a sound and structured model by applying two techniques: a technique to maximally block-structure an acyclic process model and an extended version of a technique for block-structuring flowcharts.

The paper reports on an empirical evaluation based on real-life and synthetic event logs that puts into evidence the performance of the proposed method relative to two representative methods that discover structured models by construction.

The rest of the paper is organized as follows. Section 2 introduces existing automated process discovery methods and methods for structuring process models. Section 3 presents the proposed method while Section 4 reports on the empirical evaluation. Finally, Section 5 summarizes the contributions and outlines future work directions.

2 Background and Related Work

In this section we review existing automated process discovery methods and associated quality dimensions. We also introduce methods for transforming unstructured process models into structured ones, which we later use as building blocks for our proposal.

2.1 Automated Process Discovery Algorithms

The bulk of automated process discovery algorithms are not designed to produce structured process models. This includes for example of the α-algorithm [19], which may produce unstructured models and sometimes even models with disconnected fragments. The *Heuristics Miner* [21] partially addresses the limitations of the α-algorithm and consistently performs well in terms of accuracy and simplicity metrics [20]. However, its output may be unstructured and even unsound, i.e. the produced models may contain deadlocks or gateways that do not synchronize all their incoming tokens. *Fodina*[2] is a variant of the Heuristics Miner that partially addresses the latter issue but does not generally produce structured models.

It has been observed that structured process models are generally more understandable than unstructured ones [6]. Moreover, structured process models are sound, provided that the gateways at the entry and exit of each block match. Given these advantages, several algorithms are designed to produce structured process models, represented for example as *process trees* [10, 3]. A process tree is a tree where the each leaf is labelled with an activity and each internal node is labeled with a control-flow operator: sequence, exclusive choice, non-exclusive choice, parallelism, or iteration.

[2] `http://www.processmining.be/fodina`

The *Inductive miner* [10] uses a divide-and-conquer approach to discover process trees. Using the *direct follows dependency* between event types in the log, it first creates a directly-follows graph which is used to identify cuts. A cut represent a specific control-flow dependency along which the log can be bisected. The identification of cuts is repeated recursively, starting from the most representative one until no more cuts can be identified. Once all cuts are identified and the log split into portions, a process tree is generated on top of each portion of the log. The algorithm then applies filters to remove "dangling" directly-follows edges so that the result is purely a process tree.

The *Evolutionary Tree Miner (ETM)* [3] is a genetic algorithm that starts by generating a population of random process trees. At each iteration, it computes an *overall fitness* value for each tree in the population and applies mutations to a subset thereof. A mutation is a tree change operation that adds or modifies nodes. The algorithm iterates until a stop criterion is fulfilled, and returns the tree with highest overall fitness.

Molka et al. [12] proposed another genetic automated process discovery algorithm that produces structured process models. This latter algorithm is similar in its principles to ETM, differing mainly in the set of change operations used to produce mutations.

2.2 Quality Dimensions in Automated Process Discovery

The quality of an automatically discovered process model is generally assessed along four dimensions: *recall* (a.k.a. *fitness*), *precision*, *generalization* and *complexity*.

Fitness is the ability of a model to reproduce the behavior contained in a log. Under trace semantics, a fitness of 1 means that the model can produce every trace in the log. In this paper, we use the fitness measure proposed in [2], which measures the degree to which every trace in the log can be aligned with a trace produced by the model. *Precision* measures the ability of a model to generate only the behavior found in the log. A score of 1 indicates that any trace produced by the model is somehow present in the log. In this paper we use the precision measure defined in [1], which is based on similar principles as the above fitness measure. Recall and precision can be combined into a single F-score, which is the harmonic mean of the two measurements $\left(2 \cdot \frac{Fitness \cdot Precision}{Fitness + Precision}\right)$.

Generalization measures the ability of a discovered model to produce behavior that is not present in the log but that can be produced by the process under observation. To measure generalization we use 10-fold cross validation [9]: We divide the log into 10 parts, discover a model from 9 parts (i.e. we hold-out 1 part), and we measure fitness of the discovered model against the hold-out part. This is repeated for every possible hold-out part. Generalization is the mean of the fitness values obtained for each hold-out part. A generalization of 1 means that the discovered models produce traces in the observed process, even if those traces are not in the log from which the model was discovered.

Finally, *complexity* quantifies how difficult it is to understand a model. Several complexity metrics have been shown to be (inversely) related to understandability [11], including *size* (number of nodes); *Control-Flow Complexity (CFC)* (the amount of branching caused by gateways in the model) and *structuredness* (the percentage of nodes located directly inside a well-structured single-entry single-exit fragment).

2.3 Structuring Techniques

Polyvyanyy et al. [15, 16] propose a technique to transform unstructured process models into behaviourally equivalent structured ones. The approach starts by constructing the Refined Process Structure Tree (RPST) [17] of the input process model. The RPST of a process model is a tree where the nodes are the single-entry single-exit (SESE) fragments of the model and an edge denotes a containment relation between SESE fragments. Specifically, the children of a SESE fragment in the tree are the SESE fragments that it directly contains. Fragments at the same level of the tree are disjoint.

Each SESE fragment is represented by a set of edges. Depending on how these edges are related, a SESE fragment can be of one of four types. A *trivial* fragment consists of a single edge. A *polygon* is a sequence of fragments. A *bond* is a fragment where all child fragments share two common gateways, one being the entry node and the other being the exit node of the bond. In other words, a bond consists of a split gateway with two or more sub-SESE fragments all converging into a join gateway. Any other fragment is a *rigid*. A model that consists only of trivials, polygons and bonds (i.e. no rigids) is fully structured. Thus the goal of a block-structuring technique is to replace rigid fragments in the RPST with combinations of trivials, polygons and bonds.

In the structuring technique by Polyvyanyy et al., each rigid fragment is unfolded and an ordering relation graph is generated. This graph is then parsed to construct a modular decomposition tree leading to a hierarchy of components from which a maximally structured version of the original fragment is derived. The technique in [16] produces a maximally-structured version of any acyclic fragment (and thus of any model), but it does not structure rigid fragments that contain cycles.

The problem of structuring behavioral models has also been studied in the field of programming, specifically for flowcharts: graphs consisting of tasks (instructions), exclusive split and exclusive join gateways. Oulsnam [13] identified six primitive forms of unstructuredness in flowcharts. He observed that unstructuredness is caused by the presence either of an injection (entry point) or an ejection (exit point) in one of the branches connecting a split gateway to a matching join gateway. Later, Oulsnam [14] proposed an approach to structure these six forms. The approach is based on two rules. The first rule deals with an injection, and pushes the injection after the join gateway, duplicating everything that was originally between the injection and the join. On the other hand, when the unstructuredness is caused by an ejection, the ejection is pushed after the join gateway and an additional conditional block is added to prevent the execution of unnecessary instructions. These two rules are recursively applied to the flowchart, starting from the innermost unstructured form, until no more structuring is possible.

Polyvyanyy's and Oulsnam's technique are complementary: while Polyvyanyy's technique deals mainly with unstructured acyclic rigids with parallelism, Oulsnam's one deals with rigid fragments without parallelism (exclusive gateways only). This observation is a centrepiece of the approach presented in the following section.

3 Approach

The proposed approach to discovering structured process models takes as input an event log and operates in two phases: i) discovery & cleaning, and ii) structuring.

3.1 Discovery & Cleaning

In this phase a process model is discovered from an input log using an existing process discovery algorithm. Any process discovery algorithm can be used in this phase. In this paper we use the Heuristics Miner because of its accuracy [20]. In addition to discovering an initial (unstructured) model, this phase fixes model correctness issues such as disconnected nodes (structural issues) and deadlocks (behavioral issues). This is achieved via 3 heuristics. Before presenting them, we formally define a process model.

Definition 1 (Process model). *A* process model *is a connected graph $G = (i, o, A, G^+, G^x, F)$, where A is a non-empty set of activities, i is the start event, o is the end event, G^+ is the set of AND-gateways, G^x is the set of XOR-gateways, and $F \subseteq (\{i\} \cup A \cup G^+ \cup G^x) \times (\{o\} \cup A \cup G^+ \cup G^x)$ is the set of arcs. A* split *gateway is a gateway with one incoming arc and multiple outgoing arcs, while a* join *gateway is a gateway with multiple incoming arcs and one outgoing arc.*

A process model starts with a unique start event, representing the process trigger (e.g. "order received") and concludes with a unique end event, representing the process outcome (e.g. "order fulfilled"). The model may contain activities, which capture actions that are performed during the process (e.g. "check order") and gateways, which are used for branching (split) and merging (join) purposes. Gateways can be of type XOR, to model exclusive decisions (XOR-split) and simple merges (XOR-join), and AND, to model parallelism (AND-split) and synchronization (AND-join).

The first heuristic (cf. Fig. 1) ensures that a model contains a single start and a single end event, and that every activity in the model is on a path from the start to the end. In case of multiple start or end events, these events are connected via an XOR gateway. In case of activities not on a path from start to end, the heuristic places the activity in parallel with the rest of the process, in such a way that the activity can be skipped and repeated any number of times. The second heuristic ensures that for every bond, the split and the join gateways are of the same type – both AND or both XOR but not mixed (cf. Fig. 1). In the case of an acyclic bond (a bond where all paths go from the entry to the exit gateway), the heuristic matches the exit gateway type with that of entry gateway type. If the bond is cyclic (there is a path from the exit to the entry gateway), the heuristic converts all gateways into XORs. The third heuristic addresses cases of unsoundness related to *quasi-bonds*. A quasi-bond is a bond with an injection via a join gateway or an ejection via a split gateway, along a path connecting the entry and exit gateways of the bond. The heuristic replaces the entry and exit gateways of the quasi-bond as well as the join (split) causing the injection (ejection), with XOR gateways.

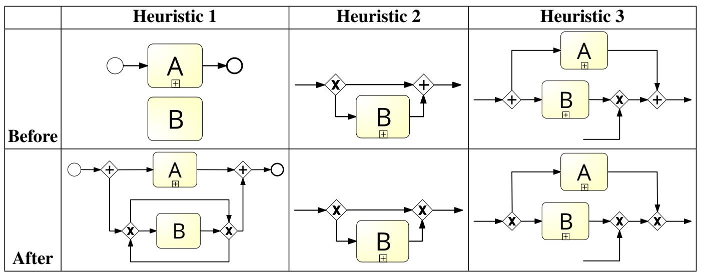


Fig. 1. Examples of application of the three cleaning heuristics.

3.2 Structuring

The second phase of our approach deals with the structuring of the discovered process model by removing injections and ejections. Before discussing this phase, we need to

Algorithm 1: Structuring flow

```
input: RPST rpst
1  Queue Queue := getLeaves(rpst);
2  Set Visited := $\varnothing$;
3  while Queue $\neq \varnothing$ do
4      node := remove(Queue);
5      parent := getParent(node);
6      if isRigid(node) then
7          if isSoundANDHomogeneous(node) OR isSoundHeterogeneous(node) then
8              BPStruct(node);
9          else EOStruct(node);
10     Visited := Visited $\cup$ {node};
11     if parent $\notin$ Visited then insert(Queue, parent);
```

formally define the notions of activity path, injection and ejection. An activity path is a path containing activity nodes only (no gateways), between two gateways.

Definition 2 (Activity Path). *Given two gateways g_{entry} and g_{exit} and a sequence of activities $S = \langle a_1, \ldots, a_n \rangle$, there is a path from g_{entry} to g_{exit}, i.e. $g_{entry} \rightsquigarrow^S g_{exit}$ iff $g_{entry} \rightarrow a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_n \rightarrow g_{exit}$, where $a \rightarrow b$ holds if there is an arc connecting a to b. Using the operator $\rightsquigarrow$ we define the set of all paths of a process model as $P \triangleq \{(g_1, g_2, S) \in G \times G \times A^* \mid g_1 \rightsquigarrow^S g_2\}$. The set of incoming paths of a gateway g_x is defined as $\circ g_x = \{(g_1, g_2, S) \in P \mid g_x = g_2\}$. Similarly the set of outgoing paths is defined as $g_x \circ = \{(g_1, g_2, S) \in P \mid g_x = g_1\}$.*

Definition 3 (Injection). *Given four different gateways g_1, g_2, g_3, g_4, they constitute an injection $i = (g_1, g_2, g_3, g_4)$ iff $\exists (S_1, S_2, S_3) \in A^* \times A^* \times A^* \mid g_1 \rightsquigarrow^{S_1} g_2 \wedge g_2 \rightsquigarrow^{S_2} g_3 \wedge g_4 \rightsquigarrow^{S_3} g_2$ (see "before" column in Table 2).*

Definition 4 (Ejection). *Given four different gateways g_1, g_2, g_3, g_4, they constitue an ejection $e = (g_1, g_2, g_3, g_4)$ iff $\exists (S_1, S_2, S_3) \in A^* \times A^* \times A^* \mid g_1 \rightsquigarrow^{S_1} g_2 \wedge g_2 \rightsquigarrow^{S_2} g_3 \wedge g_2 \rightsquigarrow^{S_3} g_4$ (see "before" column in Table 2)*

According to [17], a rigid is *homogeneous*, if for all injections and ejections in the rigid, the gateways are of the same type, otherwise it is *heterogeneous*.

Moreover, if an injection or ejection is part of a cycle the rigid is *cyclic*, otherwise it is *acyclic*. Now we have all ingredients to describe the structuring phase. In this phase, the RPST of the discovered process model is generated and all its rigids identified. Once all rigids have been identified, the RPST is traversed bottom-up, and each rigid is structured along the way.

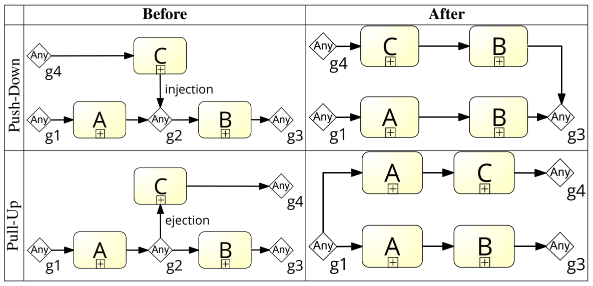


Fig. 2. Structuring of injection and ejection.

Algorithm 1 shows how the RPST is traversed and each node is structured. The algorithm uses a bottom-up traversal strategy implemented via a queue. First, all leaves

Algorithm 2: Push-Down

```
input: Injection $i = (g_1, g_2, g_3, g_4)$
input: Set of all Paths $P$
input: Set of all Gateways $G$
1 if $g_2\circ \subseteq \circ g_3$ then
2     $g'_2$ := copy($g_2$);
3     $G := G \cup \{g'_2\}$;
4     $P := P \cup \{(g_4, g'_2, S) \in G \times G \times A^* \mid \exists (g_4, g_2, S_x) \in (g_4\circ \cap \circ g_2)[S_x = S]\}$;
5     $P := P \setminus (g_4\circ \cap \circ g_2)$;
6     $P := P \cup \{(g'_2, g_3, S') \in G \times G \times A^* \mid \exists (g_2, g_3, S) \in (g_2\circ \cap \circ g_3)[S' = copy(S)]\}$;
7     if $(|g_2\circ| = 1)$ AND $(|\circ g_2| = 1)$ then $G := G \setminus \{g_2\}$;
8     if $(|g'_2\circ| = 1)$ AND $(|\circ g'_2| = 1)$ then $G := G \setminus \{g'_2\}$;
```

Algorithm 3: Pull-Up

```
input: Ejection $e = (g_1, g_2, g_3, g_4)$
input: Set of all Paths $P$
input: Set of all Gateways $G$
1 if $\circ g_2 \subseteq g_1\circ$ then
2     $g'_2$ := copy($g_2$);
3     $G := G \cup \{g'_2\}$;
4     $P := P \cup \{(g'_2, g_4, S) \in G \times G \times A^* \mid \exists (g_2, g_4, S_x) \in (g_2\circ \cap \circ g_4)[S_x = S]\}$;
5     $P := P \setminus (g_2\circ \cap \circ g_4)$;
6     $P := P \cup \{(g_1, g'_2, S') \in G \times G \times A^* \mid \exists (g_1, g_2, S) \in (g_1\circ \cap \circ g_2)[S' = copy(S)]\}$;
7     if $(|g_2\circ| = 1)$ AND $(|\circ g_2| = 1)$ then $G := G \setminus \{g_2\}$;
8     if $(|g'_2\circ| = 1)$ AND $(|\circ g'_2| = 1)$ then $G := G \setminus \{g'_2\}$;
```

of the RPST are inserted in the queue. At each step a node from the queue is removed, and structured if it is a rigid. The structuring is performed using BPStruct [15] if the rigid is *sound* and consists only of AND gateways (*sound AND-homogeneous*) or a mixture of AND and XOR gateways (*sound heretogeneous*) – cf. line 8. Otherwise the structuring is performed using an extended version of Oulsnam's algorithm [14] (line 9) as discussed later. Then the node is marked as visited and if the parent node has not been visited yet, it is added to the queue (cf. line 11). This is repeated until the queue is empty.

We decided to use two different structuring techniques since BPStruct guarantees optimal results when applied on *sound AND-homogeneous* or *heterogeneous* rigids only, whilst it produces suboptimal results for *acyclic XOR-homogeneous* rigids and it fails in case of *cyclic XOR-homogeneous* or *unsound* rigids. The structuring of these types of rigids is achieved instead using an extended version of Oulsnam's algorithm. Before presenting this latter algorithm, we need to introduce two operators.

The first operator is the *push-down* operator (see Algorithm 2). Given an Injection $i = (g_1, g_2, g_3, g_4)$, Push-Down(i) can be applied if $g_2\circ \subseteq \circ g_3$ (see line 1). The operator removes the input injection in four steps: i) it creates a copy of g_2, namely g'_2; ii) for each path from g_4 to g_2, it changes the end node of the path from g_2 to the new gateway g'_2 (lines 4 and 5); iii) for each path from g_2 to g_3, it duplicates the path, setting g'_2 as

Algorithm 4: EOStruct (Extended Oulsnam)

```
input: Rigid $r$
input: Boolean $pullup$
1  do
2      Set $I$ := detectInjections($r$);
3      Set $E := \varnothing$;
4      if $pullup$ then Set $E$ := detectEjections($r$);
5      if $I \neq \varnothing$ then Injection $i$ := cheapestInjection($I$);
6      if $E \neq \varnothing$ then Ejection $e$ := cheapestEjection($E$);
7      if $(i$ not $\perp)$ OR $(e$ not $\perp)$ then
8          if $((e = \perp)$ OR $((i$ not $\perp)$ AND $(cost(i) \leq cost(e))))$ then Push-Down($i$);
9          else Pull-Up($e$);
10 while $I \neq \varnothing$ OR $E \neq \varnothing$ ;
```

the starting node of the path, instead of g_2 (line 6); and iv) it removes any of g_2 and g'_2 if it is a trivial gateway (see Fig. 2).

The second operator is the *pull-up* operator (see Algorithm 3). Given an Ejection $e = (g_1, g_2, g_3, g_4)$, Pull-Up(e) can be applied if $\circ g_2 \subseteq g_1 \circ$ (see line 1). The operator removes the input ejection in four steps: i) it creates a copy of g_2, namely g'_2; ii) for each path from g_2 to g_4, it changes the starting node of the path from g_2 to the new gateway g'_2 (lines 4 and 5); iii) for each path from g_1 to g_2, it duplicates the path, setting g'_2 as the end node of the path, instead of g_2 (line 6); and iv) it removes any of g_2 and g'_2 if it is a trivial gateway (see Fig. 2).

While the push-down operator is an adaptation of Oulsnam's technique [14], the pull-up operator is a new operator. It can be shown that this pull-up operator preserves trace equivalence but does not preserve weak bisimulation equivalence, because it does not preserve the moment of choice (it may pull a choice to an earlier point). Due to this tradeoff, we make the use of the pull-up operator optional as discussed below.

Algorithm 4 (Extended Oulsnam) shows how the two operators are used to structure a rigid fragment. The inputs of the algorithm are an unstructured rigid and a boolean value to indicate whether the pull-up operator is to be used. First, the algorithm detects every injection on top of which the push-down operator can be applied (see line 2), and if the pull-up is enabled, every ejection on top of which the pull-up can be applied (line 4). Second, it selects the cheapest injection and the cheapest ejection (lines 5 and 6). The cheapest injection (ejection) is the injection (ejection) generating the minimum number of duplicates after a push-down (pull-up). Third, the cheapest among these two is then chosen (line 8) and the corresponding operator is applied. The algorithm iterates over these three steps until no more ejections or injections can be removed, which results in a fully structured or maximally structured rigid. Selecting the cheapest injection or ejection at each step does not ensure that the final model will have the minimum number of duplicates. In order to achieve the latter property, we embed the Extended Oulsnam algorithm inside an A^* search [8], where each state in the search tree is a transformed version of the initial rigid fragment, and the cost function associated with each state is defined as $f_{(s)} = g_{(s)} + h_{(s)}$ with $g_{(s)} = \#duplicates$ and $h_{(s)} = 0$. We set function $h_{(s)}$ to zero since it is not possible to predict how many duplicates are needed in order to structure a rigid.

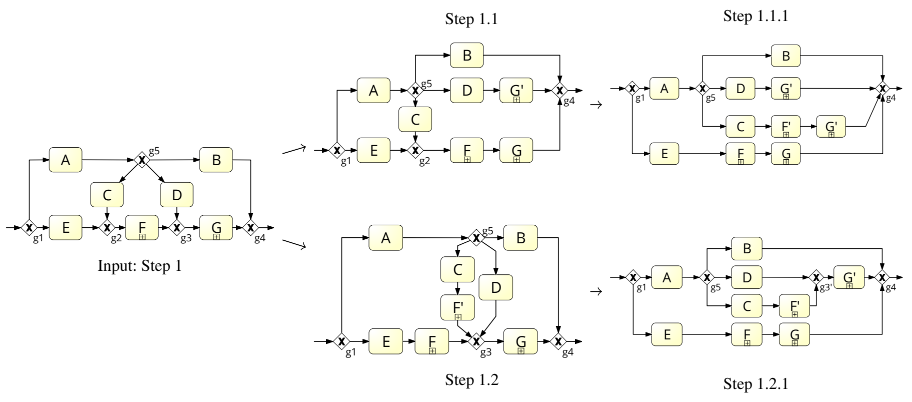


Fig. 3. An example application of the A^* search tree with our structuring method.

Figure 3 illustrates an example where a rigid is structured using Algorithm 4 within an A^* search. In this example, the rigid has two injections, i.e. $i_1 = (g_1, g_2, g_3, g_5)$ and $i_2 = (g_2, g_3, g_4, g_5)$. Assuming i_2 is the cheapest of the two injections (i.e. the size of subprocess G is smaller than the size of subprocess F), if we first remove i_2 and then i_1 (see Step 1.1 and Step 1.1.1) we will have to duplicate sub-process G twice. This would not happen if we first removed i_1 and then i_2 (see Step 1.2 and Step 1.2.1). The use of an A^* search helps us avoid these situations since it takes care of exploring the search tree and selecting the sequence of removals of injections and ejections, that leads to the minimum number of duplicated elements.

For unsound rigids, we only apply the push-down operator in order to preserve the moment of choice of the split gateways of the quasi-bonds that will be turned into bonds when structuring the rigid (not shown in Algorithm 4 for brevity). After the structuring procedure has been completed, we match the type of the join gateways of the acyclic bonds with the type of their corresponding split gateways (e.g. if the split is an AND gateway the join will be turned into an AND gateway). In case of cyclic bonds, we turn both split and join gateways into XOR to avoid soundness issues. If multiple bonds share the same join gateway, this is replaced with a chain of gateways, one for each bond, maintaining the original bonds hierarchy. Finally, since we disable the use of the pull-up operator on unsound rigids, we cannot guarantee that these will be fully structured, hence we cannot guarantee that they will be turned into sound fragments.

Complexity The complexity of the push-down and pull-up operators is linear on the number of activity paths to be duplicated when structuring an injection or ejection, i.e. $O(|g_2\circ \cap \circ g_3|)$. This is bounded by $O(n^2)$, where n is the number of nodes in the model. The complexity of the Extended Oulsnam algorithm is linear on the number of injections and ejections, which is $O\left(\binom{g}{4}\right)$ where g is the number of gateways, which is bounded by the number of nodes n. Hence, $O\left(\binom{n}{4}\right) + O(n^2) \approx O(n^4)$. Finally, the complexity of A^* is $O(b^q)$ where b is the branching factor and q is the depth of the solution. In our case the branching factor is the number of injections and ejections, and so is the depth of the solution. Hence the complexity of our method is $O(n^{4^{n^4}}) \cdot O(n^4) \approx O(n^n)$. This does not include the complexity of the baseline discovery method.

4 Evaluation

We implemented our method as a standalone tool as well as a ProM plugin, namely the *Structured Miner* (hereafter SM).[3] The tool takes a log in MXML or XES format (currently it supports Heuristics Miner (HM) and Fodina (FM) as baseline discovery algorithms), and returns a maximally structured process model in BPMN format.

Using this tool, we conducted a series of experiments to evaluate the accuracy of our discovery approach compared to that of methods that structure the model during discovery. We selected two representative methods: Inductive Miner (IM) and Evolutionary Tree Miner (ETM), and compared the results with our approach on top of HM and FM. As the results obtained with FM were consistently similar to those obtained with HM and due to space reasons, this section only reports the results using HM.

We measured accuracy using the fitness, precision, F-score and generalization metrics and model complexity via size, CFC and structuredness as defined in Section 2.2. The experiments were done on an Intel dual-core i5-3337U 1.80Ghz with 12GB of RAM running JVM 7 with 8GB of heap, except for the experiments using ETM, which were done on a 6-core Xeon E5-1650 3.50Ghz with 128GB of RAM running JVM 7 with 40GB of heap, time-bounded to 30min as the ETM algorithm is computationally very expensive and can otherwise take several hours per log.

4.1 Datasets

We generated three sets of logs using the ProM plugin "Generate Event Log from Petri Net". This plugin takes as input a process model in PNML format and generates a distinct log trace for each possible execution sequence in the model. The first set (591 Petri nets) was obtained from the SAP R/3 collection, SAP's reference model to customize their R/3 ERP product [4]. The log-generator plugin was only able to parse 545 out of 591 models, running into out-of-memory exceptions for the others. The second set (54 Workflow nets[4]) was obtained from a collection of sound and unstructured models extracted from the IBM BIT collection [6]. The BIT collection is a publicly-available set of process models in financial services, telecommunication and other domains, gathered from IBMs consultancy practice [7]. The third set contains 20 artificial models, which we created to test our method with more complex forms of unstructuredness, not observed in the two real-life collections. These are: i) rigids containing AND-gateway bonds, iii) rigids containing a large number of XOR gateways (> 5); iii) rigids containing rigids and iv) rigids being the root node of the model. Out of these 619 logs we only selected those for which HM

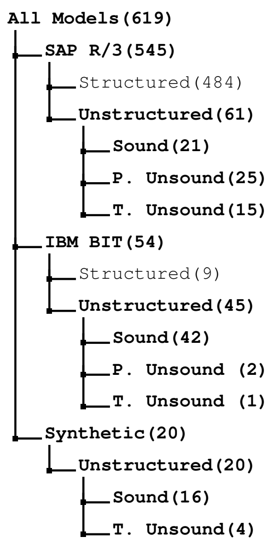


Fig. 4. Taxonomy of models discovered by HM from the logs (P. = partially, T. = totally).

[3] Available from `http://apromore.org/platform/tools`.

[4] This collection originally counted 59 models, but we discarded five duplicates.

produced an unstructured model, as our approach does not add value if the resulting model is already structured. This resulted in 126 logs, of which 61 came from SAP, 45 from IBM and 20 were synthetic. These logs range from 4,111 to 201,758 total events (avg. 49,580) with 3 to 4,235 distinct traces (avg. 137). From the models discovered with HM, we identified 79 sound models, 31 partially unsound models, i.e. models for which there is at least one complete trace, and 16 totally unsound models, i.e. models whose traces always deadlock. A taxonomy of the datasets used is shown in Fig. 4.

4.2 Results

Tables 1 and 2 report the average value and standard deviation for each quality measure across all discovery algorithms, for the models mined from the real-life data, respectively, artificial data. When HM generates sound models its output is already of high quality along fitness, precision and generalization, with a marginal standard deviation. In this case, our approach only improves the structuredness of the models, at the cost of a minor increase in size and CFC, due to the duplication introduced by the structuring. IM instead, despite having similarly high values of fitness and generalization, loses in precision with an average of 0.69 with high standard deviation, meaning that the actual precision may be much better or worse depending on the specific log used. As expected, these models are structured by construction, but CFC still remains higher than that of HM (and its structured variant SM) due to IM's tendency to generate flower models (which is also the cause for low precision). Finally, the quality of the models discovered by ETM ranks in-between that of IM and HM both in terms of accuracy and complexity, at the price of sensibly longer execution times.

Log Class (Class Size)	**Discovery Method**	**Accuracy**				**Complexity**		
		Fitness	**Precision**	**F-score**	**Gen.(10-fold)**	**Size**	**CFC**	**Struct.**
Sound (63)	IM	**1.00 ± 0.01**	0.69 ± 0.31	0.77 ± 0.26	**1.00 ± 0.01**	**23.8 ± 7.9**	11.2 ± 5.0	**1.00 ± 0.00**
	ETM	0.91 ± 0.08	0.93 ± 0.06	0.92 ± 0.06	0.90 ± 0.06	26.4 ± 8.6	**8.6 ± 4.3**	**1.00 ± 0.00**
	HM	**1.00 ± 0.00**	**0.99 ± 0.05**	0.99 ± 0.03	**1.00 ± 0.01**	25.0 ± 7.7	8.7 ± 4.2	0.50 ± 0.16
	SM_{HM}	**1.00 ± 0.00**	**0.99 ± 0.02**	**1.00 ± 0.01**	**1.00 ± 0.00**	29.7 ± 13.3	10.2 ± 6.5	0.90 ± 0.21
P. Unsound (27)	IM	**0.98 ± 0.03**	0.73 ± 0.27	0.80 ± 0.22	**0.98 ± 0.03**	22.1 ± 5.9	11.6 ± 5.0	**1.00 ± 0.00**
	ETM	0.90 ± 0.09	0.86 ± 0.11	0.87 ± 0.07	0.89 ± 0.06	**21.7 ± 7.8**	**7.5 ± 5.2**	**1.00 ± 0.00**
	HM	0.69 ± 0.21	0.85 ± 0.10	0.75 ± 0.16	0.66 ± 0.21	21.9 ± 7.3	9.0 ± 5.1	0.53 ± 0.21
	SM_{HM}	0.97 ± 0.04	**0.93 ± 0.11**	**0.95 ± 0.08**	0.97 ± 0.04	24.6 ± 10.5	10.0 ± 6.7	0.97 ± 0.15
T. Unsound (16)	IM	**0.99 ± 0.03**	0.82 ± 0.21	0.88 ± 0.14	**0.99 ± 0.03**	24.1 ± 12.0	9.6 ± 6.7	**1.00 ± 0.00**
	ETM	0.90 ± 0.10	0.87 ± 0.09	0.88 ± 0.07	0.89 ± 0.09	25.0 ± 4.2	9.2 ± 0.7	**1.00 ± 0.00**
	HM	-	-	-	-	**22.3 ± 9.4**	7.8 ± 3.6	0.72 ± 0.19
	SM_{HM}	0.96 ± 0.06	**0.92 ± 0.14**	**0.93 ± 0.11**	0.96 ± 0.06	23.2 ± 10.4	**7.7 ± 3.3**	**1.00 ± 0.00**

Table 1. Quality of models discovered from real-life data.

Log Class (Class Size)	**Discovery Method**	**Accuracy**				**Complexity**		
		Fitness	**Precision**	**F-score**	**Gen.(10-fold)**	**Size**	**CFC**	**Struct.**
Sound (16)	IM	**1.00 ± 0.01**	0.53 ± 0.31	0.64 ± 0.26	**1.00 ± 0.01**	**18.7 ± 4.5**	10.7 ± 3.7	**1.00 ± 0.00**
	ETM	0.89 ± 0.07	0.96 ± 0.05	0.92 ± 0.04	0.89 ± 0.05	22.1 ± 7.7	**7.3 ± 3.2**	**1.00 ± 0.00**
	HM	**1.00 ± 0.00**	**1.00 ± 0.00**	**1.00 ± 0.00**	**1.00 ± 0.00**	21.6 ± 5.2	8.2 ± 3.1	0.32 ± 0.17
	SM_{HM}	**1.00 ± 0.00**	**1.00 ± 0.00**	**1.00 ± 0.00**	**1.00 ± 0.00**	25.1 ± 7.7	9.1 ± 3.5	**1.00 ± 0.00**
P. Unsound (4)	IM	**1.00 ± 0.00**	0.44 ± 0.27	0.56 ± 0.22	**1.00 ± 0.00**	**23.5 ± 10.4**	11.5 ± 1.1	**1.00 ± 0.00**
	ETM	0.83 ± 0.12	0.88 ± 0.09	0.84 ± 0.07	0.78 ± 0.15	25.5 ± 1.5	10.0 ± 1.0	**1.00 ± 0.00**
	HM	0.61 ± 0.16	0.84 ± 0.06	0.69 ± 0.14	0.61 ± 0.16	27.8 ± 9.1	**8.8 ± 1.5**	0.30 ± 0.15
	SM_{HM}	0.89 ± 0.13	**0.98 ± 0.02**	**0.93 ± 0.07**	0.89 ± 0.13	30.0 ± 12.3	11.0 ± 3.3	**1.00 ± 0.00**

Table 2. Quality of models discovered from artificial data.

The improvement of our method on top of HM is more evident when the latter discovers unsound models. Here HM's accuracy dramatically worsen compared to IM and

ETM. For example, in the case of partially unsound models, on average fitness is 0.69 for HM vs. 0.98 for IM on real-life data, and 0.61 vs. 1 on artificial data, while for totally unsound models, fitness and precision for HM cannot even be measured. Our approach does not only notably increases structuredness (e.g. 0.53 vs. 0.97), but it also repairs the soundness issues and recovers the accuracy lost by HM, significantly outperforming both IM and ETM in terms of precision and F-score without compromising fitness and generalization, which get very close to those obtained by IM, e.g. fitness increases from 0.69 to 0.97, as opposed to 0.98 for IM, with an F-score of 0.95 instead of 0.80 in the case of partially unsound models discovered from real-life data. In the case of "sound models", ETM strikes a better tradeoff between accuracy and complexity compared to IM, but again, at the price of long execution times.

To illustrate when our approach outperforms IM, Fig. 5 shows the BPMN model generated by IM, HM and SM from one of the SAP R/3 logs and the corresponding quality measures.[5] In this example, the precision of the model produced by IM is low due to the presence of a large "flower-like" structure, which causes overgeneralization. Precision is higher with HM, though fitness and generalization suffer from the model being unsound. By structuring and fixing the behavioral issues of this model, SM improves on all metrics, scoring a perfect 1 for both F-score and generalization.

The negative effects of overgeneralization brought by IM are higher when the models used for generating the logs exhibit complex unstructured patterns, such as those introduced in the artificial data (cf. Table 2). For example, the precision of IM is 0.53 for sound models (with a high standard deviation), as opposed to 1 with HM. In these cases, SM consistently outperforms IM and ETM, while significantly improving over HM in terms of structuredness (0.3 vs. 1).

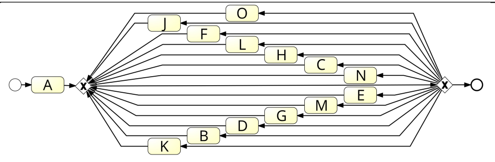
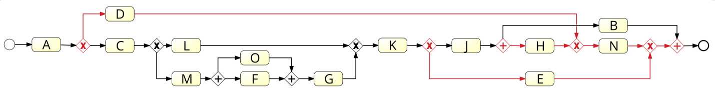
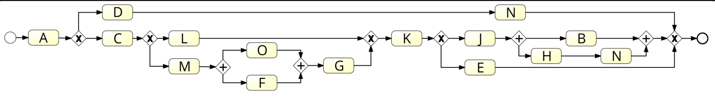
Method	Accuracy	Discovered model
IM	fitness: 1.00 precision: 0.20 F-score: 0.33 generaliz.: 1.00	
HM	fitness: 0.60 precision: 0.88 F-score: 0.72 generaliz.: 0.60	
SM	fitness: 1.00 precision: 1.00 F-score: 1.00 generaliz.: 1.00	

Fig. 5. A model from the SAP R/3 logs, discovered by IM, HM and SM (injections and gateways causing unsoundness in the HM model are highlighted).

In these experiments we disabled the pull-up operator of our method in order to ensure weak bisimulation equivalence between the model discovered by HM and the structured one. As a result, we could not fully structure 15 models from real-life data, which explains a value of structuredness less than 1 for SM in Table 1. When we enable

[5] The original labels are replaced with letters for the sake of compactness.

the pull-up operator, all the discovered models are fully structured, at the price of losing weak bisimilarity.

Time performance. Despite having exponential complexity in the worst case scenario, the time SM took to structure the models used in this evaluation was well within acceptable bounds, taking on average less than one second per model (avg = 894ms, min = 2ms, max = 109s, 95% percentile = 47.65ms).

4.3 Threats to Validity

A potential threat to internal validity is the use of process model complexity metrics as proxies for assessing the understandability of the discovered process models, as opposed to direct human judgement. However, the three chosen complexity metrics (size, CFC and structuredness) have been empirically shown to be highly correlated with perceived understandability and error-proneness [11, 6]. Further, while the process models obtained with our method are affected by the individual accuracy (fitness, precision and generalization) of the baseline algorithm used (HM or FM), Structured Miner is independent of these algorithms, and our experiments show that the method significantly improves on structuredness while keeping the same levels of accuracy. In addition, the method can often fix issues related to model correctness.

The evaluation reported above is based on two real-life datasets. This poses a threat to external validity. It should be noted though that these two datasets collect models from a variety of domains, including finance, sales, accounting, logistics, communication and human resources, and that the resulting logs are representative of different characteristics (number of events and number of distinct traces). Moreover, the use of an additional dataset artificially generated allowed us to evaluate our method against a large variety of unstructured model topologies, including some complex ones not observed in the two real-life datasets.

5 Conclusion

We presented a two-phased method to extract a structured process model from an event log wherein a process model is first extracted without any structural restriction, and then transformed into a structured one if needed. The experimental results show that this two-phased method leads to higher F-score than existing methods that discover a structured process model by design. In addition, the proposed method is more modular, insofar as different discovery and block-structuring methods can be plugged into it.

In this paper, we used the Heuristics Miner and Fodina for the first phase. In future work, we will experiment with alternative methods for discovering (unstructured) process models to explore alternative tradeoffs between model quality metrics. In the second phase, we employed a structuring method that preserves weak bisimilarity (if the pull-up operator is disabled). A direction for future work is to explore the option of partially sacrificing weak bisimilarity (while still keeping trace equivalence) to obtain models with higher structuredness. Another direction for future work is to use process model clone detection techniques [5] to refactor duplicates introduced by the structuring phase. This may allow us to strike better tradeoffs between size and structuredness.

Acknowledgments. This research is partly funded by the Australian Research Council (grant DP150103356) and the Estonian Research Council (grant IUT20-55).

References

1. A. Adriansyah, J. Munoz-Gama, J. Carmona, B. F. van Dongen, and W. M. P. van der Aalst. Alignment based precision checking. In *Proc. of BPM Workshops*, volume 132 of *LNBIP*, pages 137–149. Springer, 2012.
2. 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.
3. J. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. On the role of fitness, precision, generalization and simplicity in process discovery. In *Proc. of CoopIS*, volume 7565 of *LNCS*, pages 305–322. Springer, 2012.
4. T. Curran and G. Keller. *SAP R/3 Business Blueprint: Understanding the Business Process Reference Model*. Upper Saddle River, 1997.
5. M. Dumas, L. García-Bañuelos, M. La Rosa, and Reina Uba. Fast detection of exact clones in business process model repositories. *Inf. Syst.*, 38(4):619–633, 2013.
6. M. Dumas, M. La Rosa, J. Mendling, R. Mäesalu, H.A. Reijers, and N. Semenenko. Understanding business process models: the costs and benefits of structuredness. In *Proc. of CAiSE*, volume 7328 of *LNCS*, pages 31–46. Springer, 2012.
7. Dirk Fahland, Cédric Favre, Jana Koehler, Niels Lohmann, Hagen Völzer, and Karsten Wolf. Analysis on demand: Instantaneous soundness checking of industrial business process models. *Data Knowl. Eng.*, 70(5):448–466, 2011.
8. P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Tran. Syst. Sci. Cybern.*, 4(2):100–107, 1968.
9. R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proc. of IJCAI*, pages 1137–1145. Morgan Kaufmann, 1995.
10. 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*, volume 7927 of *LNCS*. Springer, 2013.
11. J. Mendling. *Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness*. Springer, 2008.
12. T. Molka, D. Redlich, W. Gilani, X.-J. Zeng, and M. Drobek. Evolutionary computation based discovery of hierarchical business process models. In *Proc. of BIS*, volume 208 of *LNBIP*, pages 191–204. Springer, 2015.
13. G. Oulsnam. Unravelling unstructured programs. *Comput. J.*, 25(3):379–387, 1982.
14. G. Oulsnam. The algorithmic transformation of schemas to structured form. *Comput. J.*, 30(1):43–51, 1987.
15. A. Polyvyanyy, L. García-Bañuelos, and M. Dumas. Structuring acyclic process models. *Inf. Syst.*, 37(6):518–538, 2012.
16. A. Polyvyanyy, L. García-Bañuelos, D. Fahland, and M. Weske. Maximal structuring of acyclic process models. *Comput. J.*, 57(1):12–35, 2014.
17. A. Polyvyanyy, J. Vanhatalo, and H. Völzer. Simplified computation and generalization of the refined process structure tree. In *Proc. of WS-FM*, pages 25–41, 2010.
18. W.M.P. van der Aalst. *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.
19. 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.
20. 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.
21. A.J.M.M. Weijters and J.T.S. Ribeiro. Flexible Heuristics Miner (FHM). In *Proc. of CIDM*. IEEE, 2011.