Automated Discovery of Structured Process Models From Event Logs: The Discover-and-Structure Approach

Adriano Augusto[a,b,1], Raffaele Conforti[b,1], Marlon Dumas[a], Marcello La Rosa[b,1], Giorgio Bruno[c]

[a] *University of Tartu, Estonia*
[b] *University of Melbourne, Australia*
[c] *Politecnico di Torino, Italy*

Abstract

This article tackles the problem of discovering a process model from an event log recording the execution of tasks in a business process. Previous approaches to this reverse-engineering problem strike different tradeoffs between the accuracy of the discovered models and their structural complexity. With respect to the latter property, empirical studies have demonstrated that block-structured process models are generally more understandable and less error-prone than unstructured ones. Accordingly, several methods for automated process model discovery generate block-structured models only. These methods however intertwine the objective of producing accurate models with that of ensuring their structuredness, and often sacrifice the former in favour of the latter. In this paper we propose an alternative approach that separates these concerns. Instead of directly discovering a structured process model, we first apply a well-known heuristic that discovers accurate but oftentimes unstructured (and even unsound) process models, and then we transform the resulting process model into a structured (and sound) one. An experimental evaluation on synthetic and real-life event logs shows that this discover-and-structure approach consistently outperforms previous approaches with respect to a range of accuracy and complexity measures.

Keywords: Automated process discovery, process mining, structured process model

1. Introduction

Process mining methods enable analysts to extract insights about the performance and conformance of a given business process based on logs recording the execution of

Email addresses: adriano.augusto@ut.ee (Adriano Augusto), raffaele.conforti@unimelb.edu.au (Raffaele Conforti), marlon.dumas@ut.ee (Marlon Dumas), marcello.larosa@unimelb.edu.au (Marcello La Rosa), giorgio.bruno@polito.it (Giorgio Bruno)

[1] Part of the work was conducted while the author was with the Queensland University of Technology

tasks thereof [30]. Among other things, process mining methods allow analysts to generate a process model from an event log, an operation commonly known as *automated process discovery*. In this context, an event log is a set of traces, each consisting of a sequence of events observed within one execution of a process. Such event logs can generally be extracted from enterprise information systems such as Customer Relationship Management (CRM) systems or Enterprise Resource Planning (ERP) systems [30].

A wide range of automated process discovery methods have been proposed over the past two decades, striking various tradeoffs between accuracy and structural complexity [32]. 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 be produced by the process under observation. On the other hand, structural complexity is commonly measured via metrics quantifying either the *size* of the process model, its *branching factor*, or its degree of *structuredness* (the extent to which a model is composed of well-structured single-entry, single-exit components). Also, these metrics have been empirically shown to be proxies for understandability [20].

Inspired by the observation that structured process models are often more understandable than unstructured ones [12], several automated process discovery methods generate structured models by construction [5, 18, 22]. However, these approaches intertwine the concern of accuracy with that of structuredness, sometimes sacrificing the former to achieve the latter. In this paper, we obviate this tradeoff presenting an automated process discovery method that generates structured process models, still achieving a level of fitness, precision and generalization, equal or better than methods that generate unstructured process models. The method follows a "discover-and-structure" approach. First, 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. Next, the discovered model is transformed into a sound and structured model by repeated application of two refactoring operations (push-down and pull-up of gateways) according to an A* search algorithm.

We report on an empirical evaluation of the proposed method against three state-of-the-art automated process discovery methods, based on a dataset consisting of 619 synthetic event logs and twelve real-life ones.

The article is an extended and revised version of a previous conference paper [3]. With respect to the conference version, the main enhancements are:

- An improved search algorithm and a modified version of the push-down and pull-down operators in order to improve the scalability of the block-structuring method.
- A post-processing step that reduces the size of the model by refactoring cloned model fragments introduced during the structuring phase of the method.
- An extended evaluation based on a set of twelve real-life event logs in addition to the evaluation on synthetic logs reported in the conference version.

The rest of the paper is organized as follows. Section 2 introduces existing methods for automated process discovery and for block-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 the associated quality dimensions that we will use to compare automated process discovery methods. Also, we introduce methods for transforming unstructured process models into structured ones, which we 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 the α-algorithm [31], which may produce unstructured models and sometimes even models with disconnected fragments. The *Heuristics Miner* [33] partially addresses the limitations of the α-algorithm and consistently performs well in terms of accuracy and simplicity metrics [32]. 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 [12]. 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* [5, 18]. A process tree is a tree where each leaf is labelled with an activity and each internal node is labelled with a control-flow operator: sequence, exclusive choice, non-exclusive choice, parallelism, or iteration.

The *Inductive miner* [18] 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 to identify cuts. A cut represents 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 is split into portions, a process tree is generated from 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)* [5] 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.

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

Molka et al. [22] 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.

The above methods aim to discover flat process models. Other methods focus on hierarchical process model discovery. For example, *BPMN Miner* [6, 7] employs approximate functional and inclusion dependency discovery techniques to elicit a process hierarchy from the log, which is then used with any flat discovery method to extract a hierarchical BPMN model. Another example is *two-phase mining* [19], which uses sequential patterns mining to identify subprocess boundaries. In this paper we focus on flat process models. As such, our work is complementary to those hierarchical discovery methods such as BPMN Miner that rely on an underlying flat discovery algorithm.

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 our evaluation, we use the fitness measure proposed by Adriansyah et al. [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 our evaluation, we use the precision measure defined by Adriansyah et al. [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 an automated discovery algorithm to capture behavior that is not present in the log but that can be produced by the process under observation. To measure generalization we use 3-fold cross validation [16]: We divide the log into 3 parts, discover a model from 2 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, (structural) *complexity* of a process model can be quantified via several different metrics, which have been shown to be (inversely) related to understandability [20]. As part of our evaluation, we consider *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). We selected these three complexity metrics since they not only assess the structural complexity of a process model but also directly relate to three of the seven process modelling guidelines [21].

2.3. Structuring Techniques

Polyvyanyy et al. [25, 26] propose a technique to transform unstructured process models into behaviourally equivalent structured ones. The approach starts by constructing the Refined Process Structure Tree (RPST) [27] 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. Thus, 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. This technique [26] 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 [23] 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 its matching join gateway. Later, Oulsnam [24] proposed an approach based on two rules to structure these six forms. 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. The second rule deals with an ejection. In this case, the ejection branch is moved after the join gateway, but 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. Figure 1 reports the six forms of unstructuredness detected by Oulsnam [23], and how they can be structured according to the two rules we summarized in this paragraph.

The techniques by Polyvyanyy and Oulsnam are complementary: while the former deals mainly with unstructured acyclic rigids with parallelism, the latter 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 discover maximally structured process models takes as input an event log and operates in four phases: i) discovery, ii) structuring, iii) soundness repair, and iv) clones removal. Figure 2 provides an overview of all the phases of our approach.

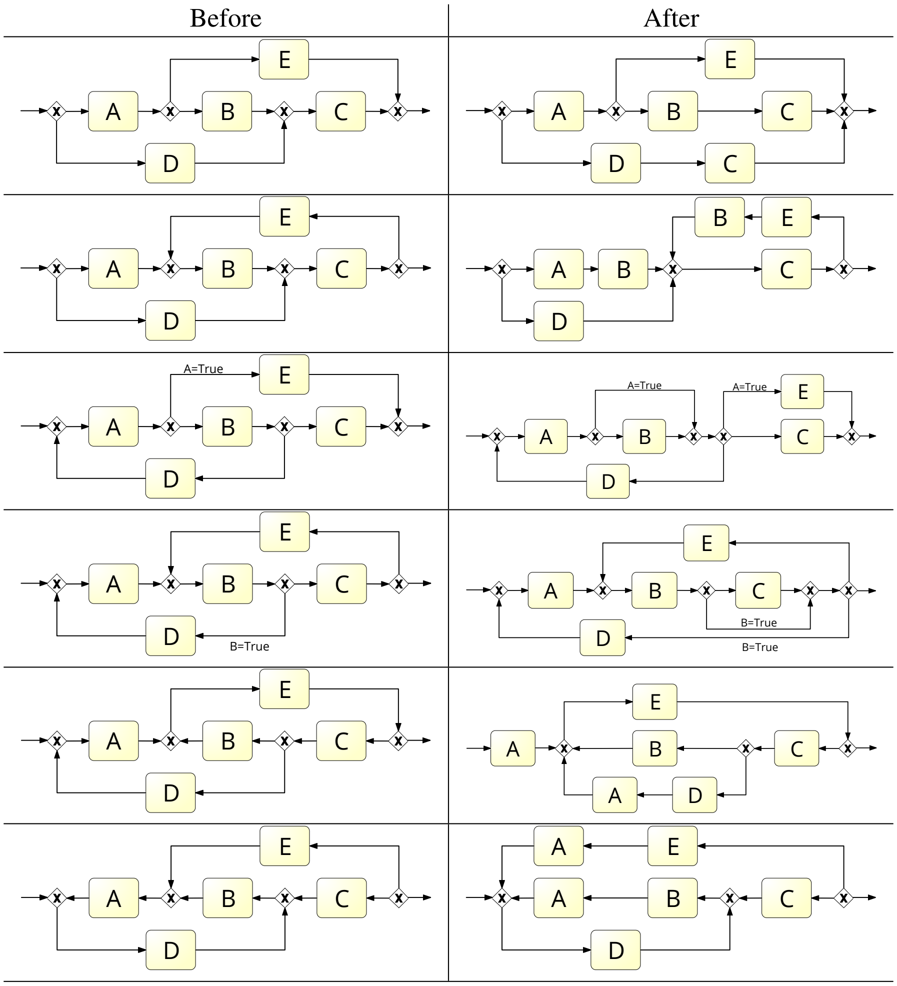


Figure 1: Oulsnam's six forms of unstructuredness, before and after structuring [23, 24].

3.1. Discovery

In this phase, we discover a process model relying on an existing process discovery algorithm. Despite any discovery algorithm can be used, in the context of this work, we rely on the Heuristics Miner as discovery algorithm due to its high accuracy [32]. The output produced in this phase is a *process model*.

Definition 1 (Process Model). *A* process model *is a connected graph* $G = (i, o, A, G^{+}, G^{\times}, F)$*, 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,* $G^{\times}$ *is the set of XOR-gateways, and* $F \subseteq (\{i\} \cup A \cup G^{+} \cup G^{\times}) \times (\{o\} \cup A \cup G^{+} \cup G^{\times})$ *is the set of arcs. Moreover, 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.*

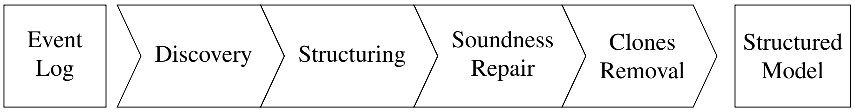


Figure 2: Overview of the proposed approach.

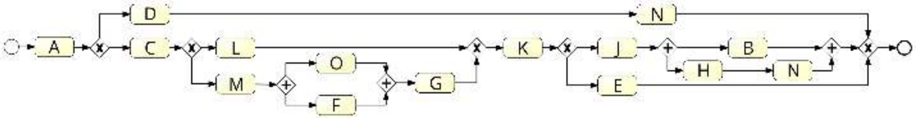


Figure 3: Example of a process model.

A process model produced by the Heuristics Miner 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 activities contained in a model capture actions that are performed during the execution of the process (e.g. "check order"), while gateways are used for branching (split) and merging (join) purposes. Gateways can be of two types: XOR or AND. Gateways of type XOR are used to model exclusive decisions (XOR-Split) and simple merges (XOR-Join), while gateways of type AND are used to model parallelism (AND-Split) and synchronization (AND-Join). Figure 3 shows an example of a process model containing several activities (the rectangles) and gateways (the diamonds).

3.2. Structuring

This phase performs the structuring of a process model via the removal of injections and ejections, i.e. the cause of unstructuredness. Before illustrating this phase in details, we need to define the notions of *activity path*, *injection*, and *ejection*. An activity path is a path between two gateways traversing only activities.

Definition 2 (Activity Path). *Let g_{entry} and g_{exit} be two gateways, and $\sigma = \langle \sigma_1, \sigma_2, \dots, \sigma_n \rangle$ a sequence of activities with $\sigma_i \in A$, $n = |\sigma|$. There is a path from g_{entry} to g_{exit}, i.e. $g_{entry} \rightsquigarrow^{\sigma} g_{exit}$ iff $g_{entry} \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \cdots \rightarrow \sigma_n \rightarrow g_{exit}$, where given two activities $a, b \in A$, $a \rightarrow b$ holds iff $(a,b) \in F$. In the remainder of the paper, given a sequence σ, we refer to σ_n as $\sigma_{|\sigma|}$.*

Moreover, given the set of all paths $P \triangleq \{\sigma \in A^* \mid \exists g_1, g_2 \in G[g_1 \rightsquigarrow^{\sigma} g_2]\}$, [3] the set of incoming paths of a gateway g_x is defined as $\circ g_x = \{\sigma \in P \mid \exists g_1 \in G[g_1 \rightsquigarrow^{\sigma} g_x]\}$. Similarly the set of outgoing paths is defined as $g_x \circ = \{\sigma \in P \mid \exists g_2 \in G[g_x \rightsquigarrow^{\sigma} g_2]\}$.

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 $\exists (\sigma', \sigma'', \sigma''') \in A^* \times A^* \times A^* \mid g_1 \rightsquigarrow^{\sigma'} g_2 \wedge g_2 \rightsquigarrow^{\sigma''}$*

[3] Given a set of symbols V, V^* refers to the set of all finite-length sequences over the symbols in V, including the empty sequence $\langle\rangle$.

$g_3 \wedge g_4 \rightsquigarrow^{\sigma'''} g_2$ *(see Figure 4a). Moreover, we refer to* g_2 *as the* injecting gateway *and to* g_3 *as the* exit gateway.

Definition 4 (Ejection). *Given four different gateways* g_1, g_2, g_3, g_4, *they constitute an ejection* $e = (g_1, g_2, g_3, g_4)$ *iff* $\exists(\sigma', \sigma'', \sigma''') \in A^* \times A^* \times A^* \mid g_1 \rightsquigarrow^{\sigma'} g_2 \wedge g_2 \rightsquigarrow^{\sigma''} g_3 \wedge g_2 \rightsquigarrow^{\sigma'''} g_4$ *(see Figure 5a). Moreover, we refer to* g_2 *as the* ejecting gateway *and to* g_1 *as the* entry gateway.

Additionally, to retrieve the subset of injections sharing the same injecting gateway and exit gateway we use the function $\mathscr{I} : G \times G \times 2^I \rightarrow 2^I$. Given a set of injections I, an injecting gateway $\overline{g}_2$, and an exit gateway $\overline{g}_3$, $\mathscr{I}_{I,\overline{g}_2,\overline{g}_3} = \{(g_1, g_2, g_3, g_4) \in I \mid g_2 = \overline{g}_2 \wedge g_3 = \overline{g}_3]\}$ retrieves the subset of injections sharing $\overline{g}_2$ and $\overline{g}_3$. Similarly we define the function $\mathscr{E} : G \times G \times 2^E \rightarrow 2^E$ to retrieved the subset of ejections sharing the same ejecting gateway and entry gateway. Given a set of ejections E, an ejecting gateway $\overline{g}_2$, and an entry gateway $\overline{g}_1$, $\mathscr{E}_{E,\overline{g}_2,\overline{g}_1} = \{(g_1, g_2, g_3, g_4) \in E \mid g_2 = \overline{g}_2 \wedge g_1 = \overline{g}_1]\}$ retrieves the subset of ejections sharing $\overline{g}_2$ and $\overline{g}_1$.

According to the definition of SESE fragments proposed by Polyvyanyy et al. [27], a SESE fragment containing injections or ejections is a rigid. Furthermore, if all the gateways composing the injections and ejections of the rigid are of the same type, the rigid is classified as *homogeneous*, otherwise as *heterogeneous*. Finally, if an injection or an ejection is part of a cycle the rigid is *cyclic*, otherwise it is *acyclic*.

Algorithm 1: Structuring

```
   input: Model m, Boolean pullup
1  $rpst = computeRPST(m)$;
2  $Queue = getLeaves(rpst)$;
3  $Visited = \varnothing$;
4  while $Queue \neq \varnothing$ do
5      $node = remove(Queue)$;
6      $parent = getParent(node)$;
7      if isRigid(node) then
8          if $isSound(node) \wedge \neg isXORHomogeneous(node)$ then BPStruct($node$);
9          else iBPStruct($node, pullup$);
10     $Visited = Visited \cup \{node\}$;
11     if $parent \notin Visited$ then insert($Queue, parent$);
```

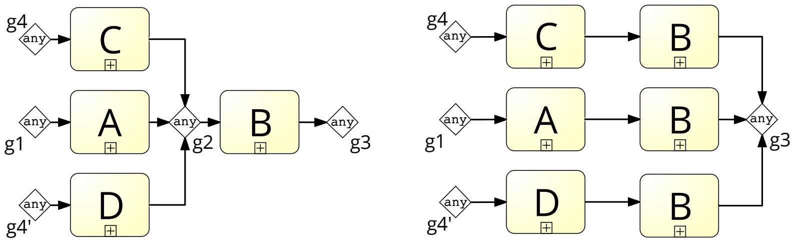
Using these elements we can finally present our structuring algorithm (see Algorithm 1). After computing the RPST of the input model, the algorithm structures the model performing a bottom-up traversal of the RPST. First, all leaves of the RPST are inserted in a queue. At each step, a node from the queue is removed and if the node is a rigid it is structured. For *sound* rigids consisting only of AND gateways (*sound AND-homogeneous*) or a mixture of AND and XOR gateways (*sound heretogeneous*) the structuring is performed using BPStruct [25] (see line 8). For the remaining cases, we use the structuring algorithm presented in Algorithm 4 (see line 9). After marking the node as visited, the parent node is added to the queue if it has not been visited yet (see line 11). These steps are then repeated for all nodes in the queue.

We use two different structuring techniques since BPStruct cannot handle *acyclic XOR-homogeneous*, *cyclic XOR-homogeneous*, and *unsound* rigids. As a matter of fact, BPStruct guarantees optimal results only when applied on top of *sound AND-homogeneous* or *heterogeneous* rigids. For this reason, the structuring algorithm illustrated in Algorithm 4 is only use for the rigids that cannot be handled by BPStruct.

Algorithm 2: Push-Down

```
input: Set of Activities $A$, Set of Flows $F$, Set of Gateways $G$, Set of Injections
       $I$, Injecting Gateway $\overline{g}_2$, Exit Gateway $\overline{g}_3$
1  $I_{\overline{g}_2,\overline{g}_3} = \mathscr{I}_{I,\overline{g}_2,\overline{g}_3}$;
2  $\sigma = \sigma'' \in A^* \mid \sigma'' \in (\overline{g}_2\circ \cap \circ\overline{g}_3)$;
3  while $I_{\overline{g}_2,\overline{g}_3} \neq \varnothing$ do
4      $(g_1, \overline{g}_2, \overline{g}_3, g_4) = any(I_{\overline{g}_2,\overline{g}_3})$;
5      $\overline{g}_2{}' = copy(\overline{g}_2)$;
6      $\sigma' = copy(\sigma)$;
7      $G = G \cup \{\overline{g}_2{}'\}$;
8      $A = A \cup \{a_i \mid a_i \in \sigma'\}$;
9      $F = F \cup \{(a_x, a_y) \in A \times A \mid \exists \sigma'_i \in \sigma'[\sigma'_i = a_x \wedge \sigma'_{i+1} = a_y]\}$;
10     $F = F \cup \{(\overline{g}_2{}', \sigma'_1), (\sigma'_n, \overline{g}_3)\}$;
11     $\Sigma = g_4\circ \cap \circ\overline{g}_2$;
12     $F = F \cup \{(a, g_y) \in A \times G \mid g_y = \overline{g}_2{}' \wedge \exists \sigma'' \in \Sigma[a = \sigma''_n]\}$;
13     $F = F \setminus \{(a, g_y) \in A \times G \mid g_y = \overline{g}_2 \wedge \exists \sigma'' \in \Sigma[a = \sigma''_n]\}$;
14     $I = I \setminus \{(g_1, \overline{g}_2, \overline{g}_3, g_4), (g_4, \overline{g}_2, \overline{g}_3, g_1)\}$;
15     $I_{\overline{g}_2,\overline{g}_3} = \mathscr{I}_{I,\overline{g}_2,\overline{g}_3}$;
16     if $(|\circ\overline{g}_2{}'| = 1 \wedge |\overline{g}_2{}'\circ| = 1)$ then
17         $a_1 = a \in A \mid \exists \sigma'' \in \circ\overline{g}_2{}'[a = \sigma''_n]$;
18         $a_2 = a \in A \mid \exists \sigma'' \in \overline{g}_2{}'\circ[a = \sigma''_1]$;
19         $F = F \cup \{(a_1, a_2)\} \setminus \{(a_1, \overline{g}_2{}'), (\overline{g}_2{}', a_2)\}$;
20         $G = G \setminus \{\overline{g}_2{}'\}$;
21 if $(|\circ\overline{g}_2| = 1 \wedge |\overline{g}_2\circ| = 1)$ then
22     $a_1 = a \in A \mid \exists \sigma'' \in \circ\overline{g}_2[a = \sigma''_n]$;
23     $a_2 = a \in A \mid \exists \sigma'' \in \overline{g}_2\circ[a = \sigma''_1]$;
24     $F = F \cup \{(a_1, a_2)\} \setminus \{(a_1, \overline{g}_2), (\overline{g}_2, a_2)\}$;
25     $G = G \setminus \{\overline{g}_2\}$;
26 return $(A, F, G, I)$;
```

Before presenting the algorithm, we need to introduce the *push-down* operator and the *pull-up* operator. The *push-down* operator (see Algorithm 2), inspired by Oulsnam's push-down [24], removes at once all injections contained in a set of shared injections. To achieve this the operator cycles over each injection $(g_1, \overline{g}_2, \overline{g}_3, g_4) \in I_{\overline{g}_2,\overline{g}_3}$ and removes it through the following four steps:

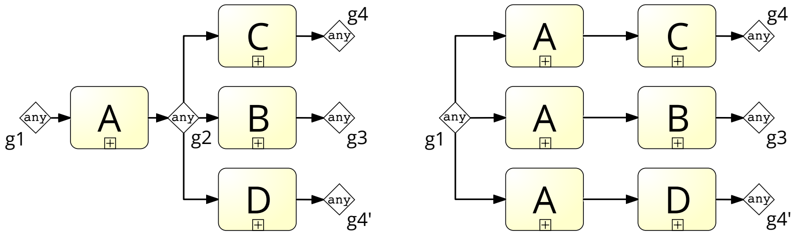


(a) Injection.

(b) Injection removal.

Figure 4: Example of push-down operator.

- create a copy of $\overline{g}_2$, namely $\overline{g}_2{}'$;
- duplicate the path from $\overline{g}_2$ to $\overline{g}_3$, and connect $\overline{g}_2{}'$ to the path (see lines 8-10);
- replace $\overline{g}_2$ with $\overline{g}_2{}'$ for each path going from g_4 to $\overline{g}_2$ (see lines 11-13);
- remove $\overline{g}_2{}'$, if it is a trivial gateway (see lines 17-20).



(a) Ejection.

(b) Ejection removal.

Figure 5: Example of pull-up operator.

On the other hand, the *pull-up* operator (see Algorithm 3) removes at once all ejections contained in a set of shared ejections applying the following steps over each ejection $(\overline{g}_1, \overline{g}_2, g_3, g_4) \in E_{\overline{g}_2, \overline{g}_1}$.

- create a copy of $\overline{g}_2$, namely $\overline{g}_2{}'$;
- duplicate the path from $\overline{g}_1$ to $\overline{g}_2$, and connect the path to $\overline{g}_2{}'$ (see lines 8-10);
- replace $\overline{g}_2$ with $\overline{g}_2{}'$ for each path connecting $\overline{g}_2$ to g_4 (lines 11-13);
- remove $\overline{g}_2{}'$, if it is a trivial gateway (see lines 17-20).

Figure 4b and Figure 5b show the application of the push-down operator and pull-up operator respectively. Despite providing additional structuring power, the pull-up operator does not preserve weak bisimulation equivalence, since by pulling-up a gateway we are moving the moment of choice to an earlier point in the model. To address this possible limitation, the use of this operator is left to the user to decide.

These two operators are used in the context of Algorithm 4. After retrieving the set of activities, flows, and gateways contained in the rigid (see lines 1-3), the algorithm

Algorithm 3: Pull-Up

```
input: Set of Activities $A$, Set of Flows $F$, Set of Gateways $G$, Set of Ejections
       $E$, Ejecting Gateway $\overline{g}_2$, Entry Gateway $\overline{g}_1$

$E_{\overline{g}_2,\overline{g}_1} = \mathscr{E}_{E,\overline{g}_2,\overline{g}_1}$;
$\sigma = \sigma'' \in A^* \mid \sigma'' \in (\overline{g}_1\circ \cap \circ\overline{g}_2)$;
while $E_{\overline{g}_2,\overline{g}_1} \neq \varnothing$ do
    $(\overline{g}_1,\overline{g}_2,g_3,g_4) = any(E_{\overline{g}_2,\overline{g}_1})$;
    $\overline{g}_2{}' = copy(\overline{g}_2)$;
    $\sigma' = copy(\sigma)$;
    $G = G \cup \{\overline{g}_2{}'\}$;
    $A = A \cup \{a_i \mid a_i \in \sigma'\}$;
    $F = F \cup \{(a_x,a_y) \in A \times A \mid \exists \sigma'_i \in \sigma'[\sigma'_i = a_x \wedge \sigma'_{i+1} = a_y]\}$;
    $F = F \cup \{(\overline{g}_1,\sigma'_1),(\sigma'_n,\overline{g}_2{}')\}$;
    $\Sigma = \overline{g}_2\circ \cap \circ g_4$;
    $F = F \cup \{(g_y,a) \in G \times A \mid g_y = \overline{g}_2{}' \wedge \exists \sigma'' \in \Sigma[a = \sigma''_1]\}$;
    $F = F \setminus \{(g_y,a) \in G \times A \mid g_y = \overline{g}_2 \wedge \exists \sigma'' \in \Sigma[a = \sigma''_1]\}$;
    $E = E \setminus \{(\overline{g}_1,\overline{g}_2,g_3,g_4),(\overline{g}_1,\overline{g}_2,g_4,g_3)\}$;
    $E_{\overline{g}_2,\overline{g}_1} = \mathscr{E}_{E,\overline{g}_2,\overline{g}_1}$;
    if $(|\circ\overline{g}_2{}'| = 1 \wedge |\overline{g}_2{}'\circ| = 1)$ then
        $a_1 = a \in A \mid \exists \sigma'' \in \circ\overline{g}_2{}'[a = \sigma''_n]$;
        $a_2 = a \in A \mid \exists \sigma'' \in \overline{g}_2{}'\circ[a = \sigma''_1]$;
        $F = F \cup \{(a_1,a_2)\} \setminus \{(a_1,\overline{g}_2{}'),(\overline{g}_2{}',a_2)\}$;
        $G = G \setminus \{\overline{g}_2{}'\}$;
if $(|\circ\overline{g}_2| = 1 \wedge |\overline{g}_2\circ| = 1)$ then
    $a_1 = a \in A \mid \exists \sigma'' \in \circ\overline{g}_2[a = \sigma''_n]$;
    $a_2 = a \in A \mid \exists \sigma'' \in \overline{g}_2\circ[a = \sigma''_1]$;
    $F = F \cup \{(a_1,a_2)\} \setminus \{(a_1,\overline{g}_2),(\overline{g}_2,a_2)\}$;
    $G = G \setminus \{\overline{g}_2\}$;
return $(A,F,G,E)$;
```

Algorithm 4: iBPStruct

```
   input: Rigid $r$, Boolean $pullup$
1  $A_r$ = getActivities($r$);
2  $F_r$ = getFlows($r$);
3  $G_r$ = getGateways($r$);
4  $I$ = getInjections($r$);
5  $E$ = $\varnothing$;
6  if $pullup$ then $E$ = getEjections($r$);
7  while $I \neq \varnothing \vee E \neq \varnothing$ do
8      $(\overline{g}_2, \overline{g}_y) = selectBestUsingA^*(r)$;
9      if $\overline{g}_2 \neq \perp$ then
10         if $\mathscr{I}_{I,\overline{g}_2,\overline{g}_y} \neq \varnothing$ then
11             $(A', F', G', I')$ = Push-Down($A_r, F_r, G_r, I, \overline{g}_2, \overline{g}_y$);
12             $A_r = A'$, $F_r = F'$, $G_r = G'$, $I = I'$;
13         else
14             $(A', F', G', E')$ = Pull-Up($A_r, F_r, G_r, E, \overline{g}_2, \overline{g}_y$);
15             $A_r = A'$, $F_r = F'$, $G_r = G'$, $E = E'$;
16 return $(A_r, F_r, G_r)$;
```

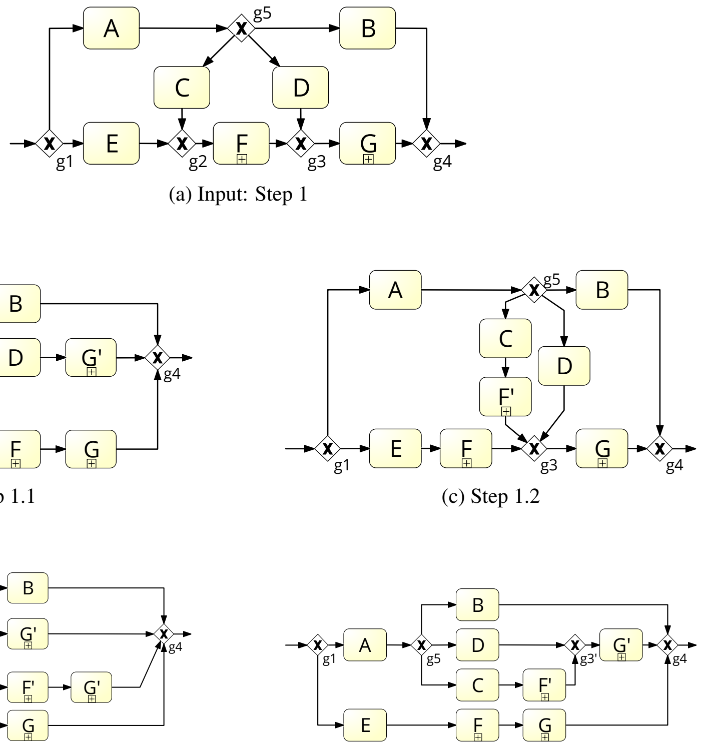
detects all the injections contained in the rigid (see line 4), and if the pull-up rule is enabled, all the ejections (see line 6). It then selects the shared injections (or ejections if enabled) which will introduce the least number duplicates (see line 8), and will remove them applying the corresponding operator, i.e. push-down or pull-up (see lines 10-15). These steps are repeated until no more injections or ejections can be removed, resulting in a fully or maximally structured rigid.

To obtain a structured model with the minimum number of duplicates, we select the shared injections (or ejections) to be removed through an A^* search [15]. In this scenario, the cost function required by the A^* search and associated with each node of the search tree is defined as $f(s) = g(s) + h(s)$, where the current cost function $g(s)$ is defined as $g(s) = \#duplicates$ and the future cost function $h(s)$ is defined as $h(s) = 0$. We set $h(s) = 0$ since it is not possible to predict how many duplicates would be required to structure a rigid.

Figure 6 illustrates an example where a rigid is structured using Algorithm 4. In this example, the rigid has two sets of shared injections, I_1 and I_2. Set I_1 contains injections $i_1 = (g_1, g_2, g_3, g_5)$ and $i_2 = (g_5, g_2, g_3, g_1)$, where g_2 is the injecting gateway and g_3 the exit gateway. Set I_2 contains injections $i_3 = (g_2, g_3, g_4, g_5)$ and $i_4 = (g_5, g_3, g_4, g_2)$, where g_3 is the injecting gateway and g_4 the exit gateway.

Let us assume that I_2 is the cheapest set of shared injections. If we apply a greedy selection we will first remove I_2 and then I_1 (see Step 1.1 and Step 1.1.1), which will result in duplicating sub-process G twice. On the other hand, this would not happen if we first remove I_1 and then I_2 (see Step 1.2 and Step 1.2.1).

While it is clear why we should perform the removal of injections (ejections) at



(a) Input: Step 1

(b) Step 1.1

(c) Step 1.2

(d) Step 1.1.1

(e) Step 1.2.1

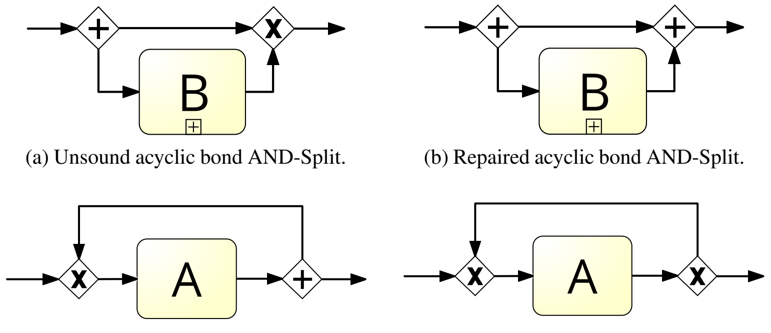
Figure 6: An example application of the A^* search tree with our structuring method.

a global level, an A^* search could be quite detrimental for the performance of our approach. To avoid this, we implemented two additional versions of our algorithm, one using a time-bound A^* search [4] and the other using a memory-bound A^* [28].

3.3. Soundness Repair

As a matter of facts a structured model resulting from a model that was originally unsound will still be unsound. This phase tries to address this issue through the application of three heuristics. The first heuristic is used in case of acyclic bonds. Whenever we encounter an acyclic unsound bond we match the type of the join gateway of the bond with the type of its corresponding split gateway, e.g. if the split is an AND gateway the join will be turned into an AND gateway. Figures 7a and 7b show the repair of an unsound acyclic bond. The second heuristic deals with unsound cyclic bonds. In this case we replace the split and join gateways of the bond with XOR gateways (see Figures 7c and 7d).

Finally, the third heuristic deals with nested bonds sharing the same join gateway. Whenever we encounter bonds sharing the same join gateway, we replace the gateway with a chain of gateways, one for each bond. In doing so, we maintain the original bonds hierarchy. This heuristic enables the application of the first heuristic, which



(a) Unsound acyclic bond AND-Split.

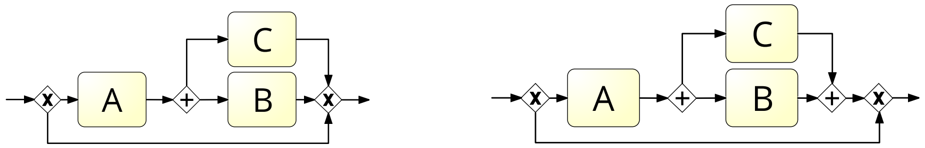
(b) Repaired acyclic bond AND-Split.

(c) Unsound cyclic bond AND-Split.

(d) Repaired cyclic bond AND-Split.

Figure 7: Repair of unsound bonds.

otherwise would not be able correctly identify the matching split. Figure 8 shows the application of this third heuristic.



(a) Unsound nested bonds.

(b) Repaired nested bonds.

Figure 8: Repair of unsound nested bonds sharing the join gateway.

3.4. Clone Refactoring

As result of the application of push-downs and pull-ups, a structured model may contain several duplicates which negatively affect the quality of the model. To address this issue, the last phase of our approach removes duplicate activities which are not required to maintain maximal structuredness.

Before describing the algorithm, we need to highlight some of the properties of a bond (i.e. a node of an RPST) which allows us to remove clones:

- A bond has always at least two children, and they are both RPST nodes.
- If a child of a bond is a trivial, the trivial is an edge connecting the entry gateway of the bond with the exit gateway of the bond.
- If a child of a bond is a rigid, the entry and the exit gateways of the rigid match the entry and the exit gateways of the bond.
- If a child of a bond is a polygon (i.e. a sequence of RPST nodes), the entry gateway of the first RPST node of the polygon is the entry gateway of the bond, and the exit gateway of the last RPST node of the polygon is the exit gateway of the bond.

Algorithm 5: Clone Removal

```
input: Process Model $G$
$G' = \perp$;
do
    $G' = G$;
    $RPST = computeRPST(G)$;
    $computeCanonicalCodes(RPST)$;
    $Bonds = getRPSTBonds(RPST)$;
    foreach $bond \in Bonds$ do
        $\mathscr{A} = \varnothing, \mathscr{O} = \varnothing$;
        foreach $child \in getChildren(bond)$ do
            if $isPolygon(child)$ then
                $\mathscr{A} = \mathscr{A} \cup \{getFirstNode(child)\}$;
                $\mathscr{O} = \mathscr{O} \cup \{getLastNode(child)\}$;
            else if $isRigid(child)$ then
                $\mathscr{A} = \mathscr{A} \cup \{child\}$;
                $\mathscr{O} = \mathscr{O} \cup \{child\}$;
        while $\mathscr{A} \neq \varnothing$ do
            $node = any(\mathscr{A})$;
            $\mathscr{S} = \{node' \in \mathscr{A} \mid getCode(node') = getCode(node)\}$;
            if $|\mathscr{S}| > 1$ then $G = removeClonesSharingEntry(G, \mathscr{S})$;
            $\mathscr{A} = \mathscr{A} \setminus \mathscr{S}$;
        if $G' \neq G$ then break;
        while $\mathscr{O} \neq \varnothing$ do
            $node = any(\mathscr{O})$;
            $\mathscr{S} = \{node' \in \mathscr{O} \mid getCode(node') = getCode(node)\}$;
            if $|\mathscr{S}| > 1$ then $G = removeClonesSharingExit(G, \mathscr{S})$;
            $\mathscr{O} = \mathscr{O} \setminus \mathscr{S}$;
        if $G' \neq G$ then break;
while $G' \neq G$ ;
return $G$;
```

- A child of a bond cannot be a bond.[4]

The removal of duplicates activities is achieve by Algorithm 5 through the following steps. First, a *canonical code* [11, 29] is assigned to each node of the RPST (see line 5). Then all bonds contained in the model are retrieved (see line 6). Successively, the children of each bond sharing the same entry and/or the exit are identified (see lines 8-15). In particular, if the child is a polygon we consider the first RPST node of the polygon for the entry and the last RPST node of the polygon for the exit (see lines 11 and 12), while if the child is a rigid (i.e. in case fully structuring was not achieved) the node is considered for both entry and exit (see lines 14-15).

In the next step, clones among children sharing the same entry are detected (see line 18). This is achieved comparing the *canonical code* of each child, (two RPST nodes are clones if their canonical code is the same [11, 29]). Clones children sharing the same entry or exit are then removed and the model updated (see line 19 and line 25). Finally, after updating the model, the RPST is recalculated and a new iteration begins (see line 21 and line 27). These steps are repeated until no further clones are removed.

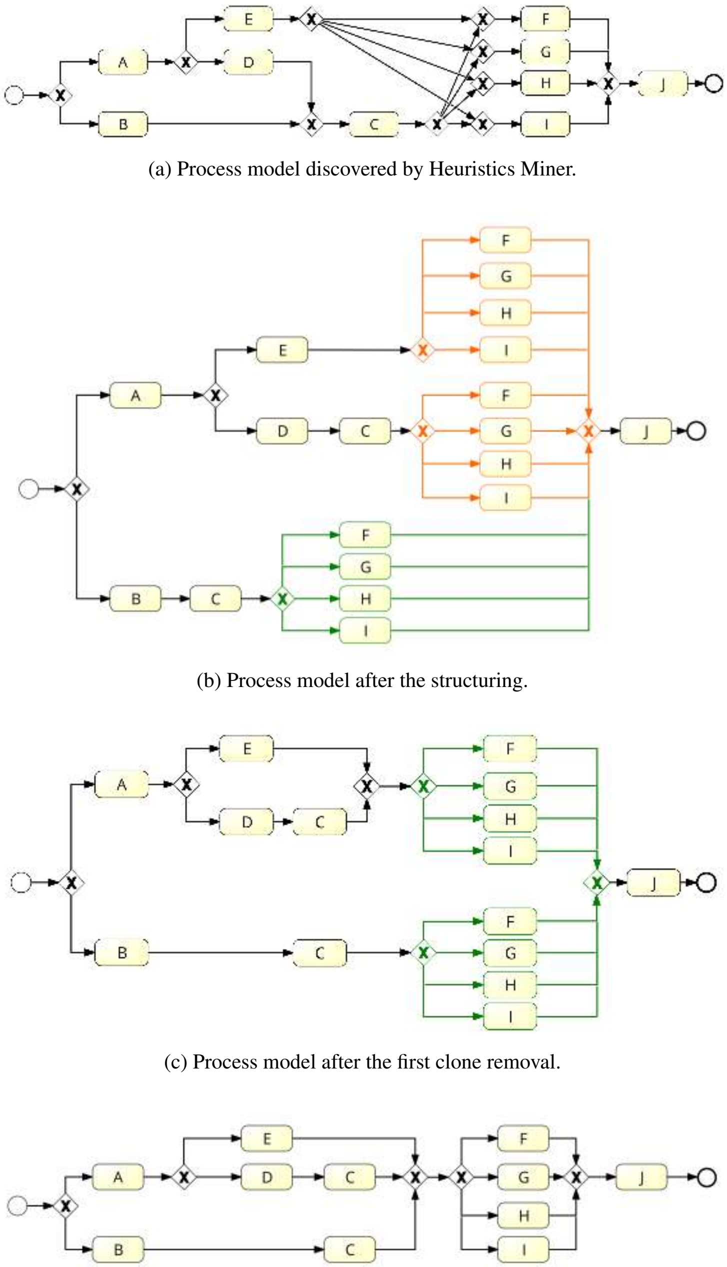
Figure 9 shows an example, from one of the logs used in the evaluation, where we applied structuring and clone removal. Figure 9a shows the model discovered using the Heuristics Miner which is provided in input to our structuring algorithm. After the structuring phase, the resulting model contains three bonds that are exact clones (highlighted in orange and green in Figure 9b). The two colors are used to differentiate the parent bond to which each clone belongs to. The two bonds in orange are the end nodes of two polygons children of the bond starting after activity A, while the bond in green is the end node of the polygon children of the bond starting after the start event.

After a first iteration, our algorithm only removes the two orange bonds (see Figure 9c). It is important to notice that the bond in green is not removed since it belongs to a different parent bond. After removing the bonds in orange, we perform a second iteration of our algorithm. This time the two bonds in green are part of the same parent bond, hence our algorithm proceeds with their removal producing as final result the model shown in Figure 9d. Finally, the algorithm will perform a third iteration which will cause the algorithm to terminate not being able to find additional clones to remove.

3.5. Complexity Analysis

The complexity of our methods depends on the complexity of the model structuring, the soundness repair, and the final clone removal. For the structuring, we have to consider the complexity of the push-down and pull-up operators. Both contain a loop on the number of injections or ejections (having the same injecting or ejecting gateway) to be removed. This operation is in the worst case $O\left(\binom{g}{2}\right)$, since we may have to loop over all possible sets of shared injections (or ejections) that exist between a given injecting gateway $\overline{g}_2$ and exit gateway $\overline{g}_3$ (or ejecting gateway $\overline{g}_2$ and entry gateway $\overline{g}_1$ for the pull-up). Since the number of gateways is bounded by the number of nodes, $O\left(\binom{g}{2}\right)$ is bounded by $O\left(\binom{n}{2}\right)$. The complexity of our structuring algorithm is

[4] Otherwise the child bond would share the entry and the exit gateways of the parent bond, and the children of the child bond would be children of the parent bond.



(a) Process model discovered by Heuristics Miner.

(b) Process model after the structuring.

(c) Process model after the first clone removal.

(d) Final output after the second clone removal.

Figure 9: Example of structuring and clone removal.

linear on the number of injections and ejections, which is $O\left(\binom{g}{4}\right)$, assuming that in the worst case we have an injection or ejection each four gateways. This operation is also bounded by $O\left(\binom{n}{4}\right)$. Hence, $O\left(\binom{n}{2}\right)+O\left(\binom{n}{4}\right) \approx O\left(\binom{n}{4}\right)$. 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 the structuring is $O\left(\binom{n}{4}^{\binom{n}{4}}\right) \cdot O\left(\binom{n}{4}\right) \approx O\left(\binom{n}{4}^{\binom{n}{4}}\right)$.

The complexity of repairing the soundness of a model is linear on the number of bonds contained in the model, which is bounded by the number of nodes. Additionally, since we have to compute the RPST of a model, which is linear on the number of edges and nodes contained in the model, the complexity of repairing the soundness is $O(n)+O(v+n) \approx O(n)+O(n^2+n) \approx O(n^2)$.

As for clone removal, the complexity of computing the RPST is $O(v+n) \approx O(n^2+n) \approx O(n^2)$. The complexity of computing the canonical code is linear on the numbers of nodes and it is computed for each RPST node (bounded by the number of nodes of the model). Thus, we obtain: $n * O(n) \approx O(n^2)$. Given a bond, identifying the children that share an entry or exit is linear on the number of children (bounded by the number of nodes of the input model), hence $O(n)$. The removal of clones which share an entry or exit is $n*(O(n)+O(n)) \approx O(n^2)$. This is because identifying the clones of a given child is $O(n)$, and the functions *removeClonesSharingEntry* and *removeClonesSharingExit* are linear on the size of the input set (bounded by the number of nodes). Hence given a bond, the removal of clones within this bond is $O(n)+O(n^2) \approx O(n^2)$. Additionally, we have to repeat this operation for each bond, hence $n \cdot O(n^2) \approx O(n^3)$. Finally, since we have to repeat the removal until no more clones can be found, and the number of clones is bounded by the number of nodes, we have a final complexity for this operation of $n*(O(n^2)+O(n^2)+O(n^3)) \approx O(n^4)$.

The overall complexity is thus $O\left(\binom{n}{4}^{\binom{n}{4}}\right)+O(n^2)+O(n^4) \approx O\left(\binom{n}{4}^{\binom{n}{4}}\right)$.

4. Evaluation

We implemented our method as a standalone Java application called *Structured Miner*,[5] and embedded it into the BPMN Miner plugin of the Apromore online process analytics platform [17].[6] This tool supports Heuristics Miner version 5.2 and 6, and Fodina as the base methods for unstructured process model discovery. It takes a log in MXML or XES format as input, and returns a process model in BPMN format.

Using this tool, we conducted a two-pronged evaluation. First, we assessed the accuracy and complexity of the models on a large set of synthetic logs. Next, we repeated the measurements on a battery of publicly-available real-life logs. In addition to execution times, we measured accuracy using fitness, precision and their F-score, generalization using 3-fold fitness, and model complexity via size, CFC and structuredness as defined in Section 2.2.

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

[6] `http://apromore.org`

We tested our method on top of the three base discovery algorithms supported by the tool implementation. In the following we only report the results on top of Heuristics Miner 6 (hereafter referred to as S-HM$_6$), since the use of our method on top of this discovery algorithm led to the best results. We compared these results with those obtained by two representative methods for structured process model discovery: Inductive Miner (IM) and Evolutionary Tree Miner (ETM). The required model conversions (e.g. from BPMN to Petri nets to measure accuracy and generalization) were done with ProM's *BPMN Miner* package.[7] This package introduces the BPMN loop marker to replace self-loop activities, leading to small savings in size. For IM, ETM and HM$_6$, we used the corresponding implementation in ProM with default parameters. For our tool we used a standard A^* for the structuring phase, time-bounded at two minutes, followed by a memory-bound A^* with 10 children per parent node, with the latter operation also bounded at two minutes.

The experiments were conducted on a 6-core Xeon E5-1650 3.50Ghz with 128GB of RAM running JVM 8 with 16GB of heap space. Each discovery operation was timed out at 30 minutes for each log in the synthetic dataset, and at one hour for each log in the real-life dataset.

4.1. Datasets

For the first experiment, we generated three sets of synthetic logs using the ProM plugin "Generate Event Log from Petri Net".[8] 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 used to customize their R/3 ERP product [9]. 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[9]) was obtained from a collection of sound and unstructured models extracted from the IBM BIT collection [12]. The BIT collection is a publicly-available set of process models in financial services, telecommunication and other domains, gathered from IBMs consultancy practice [13]. 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.

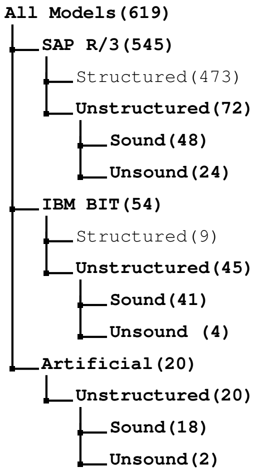


Figure 10: Taxonomy of models discovered by HM$_6$ from the synthetic logs.

These are: i) rigids containing AND-gateway bonds, ii) rigids containing a large number of XOR gateways (> 5); iii) rigids containing rigids and iv) rigids being the

[7] `http://www.promtools.org`

[8] `http://processmining.be/loggenerator`

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

Log	Total	Distinct	Total	Distinct	Trace Length		
Name	**Traces**	**Traces (%)**	**Events**	**Events**	**min**	**avg**	**max**
BPIC12	13087	33.4	262200	36	3	20	175
BPIC13$_{\text{cp}}$	1487	12.3	6660	7	1	4	35
BPIC13$_{\text{inc}}$	7554	20.0	65533	13	1	9	123
BPIC14$_{\text{f}}$	41353	36.1	369485	9	3	9	167
BPIC15$_{\text{1f}}$	902	32.7	21656	70	5	24	50
BPIC15$_{\text{2f}}$	681	61.7	24678	82	4	36	63
BPIC15$_{\text{3f}}$	1369	60.3	43786	62	4	32	54
BPIC15$_{\text{4f}}$	860	52.4	29403	65	5	34	54
BPIC15$_{\text{5f}}$	975	45.7	30030	74	4	31	61
BPIC17$_{\text{f}}$	21861	40.1	714198	41	11	33	113
RTFMP	150370	0.2	561470	11	2	4	20
SEPSIS	1050	80.6	15214	16	3	14	185

Table 1: Descriptive statistics of the real-life logs.

root node of the model. Out of these 619 logs we only selected those for which HM$_6$ produced an unstructured model, as our method does not add value if the resulting model is already structured. This resulted in 137 logs, of which 72 came from SAP, 45 from IBM and 20 were artificial. These logs range from 4,111 to 201,758 total events (avg. 50340) with 3 to 4,235 distinct traces (avg. 132). From the models discovered with HM$_6$, we identified 107 sound models and 30 unsound models, i.e. models whose traces deadlock. A taxonomy of the synthetic dataset is shown in Fig. 10.

The dataset used for the second experiment contains twelve real-life logs publicly available in the "4TU Centre for Research Data".[10] From the collection present in this website, we included the *BPI Challenge* (BPIC) logs, except those that do not explicitly capture a business process (i.e. the BPIC 2011 and 2016 logs), the *Road Traffic Fines Management Process* (RTFMP) and the *SEPSIS Cases* logs, and left out those logs that are already contained in other logs (e.g. the *Environmental permit application process* log). The twelve selected logs record executions of business processes in different domains, including healthcare, finance, government and IT service management. In three logs (BPIC14, BPIC15 and BPIC17), we applied the filtering technique in [8] to remove infrequent behavior. This was necessary since otherwise, all the models discovered by the methods tested had very poor accuracy (F-score close to 0 or not computable), making the comparison useless.

Table 1 reports the characteristics of these logs. We can observe that the collection is widely heterogeneous ranging from simple to very complex logs. The log size ranges from 681 traces (for the BPIC15$_{\text{2f}}$ log) to 150,370 traces (for the RTFMP log). Similar differences can be observed in the percentage of distinct traces, ranging from 0.2% to 80.6%, and in the number of event classes (i.e. activities executed within the process), ranging from 7 to 82. The length of a trace also varies from very short traces, counting one event only, to very long ones, counting 185 events.

[10] `https://data.4tu.nl/repository/collection:event_logs_real`

Discovery	**Accuracy**			**Gen.**
Method	**Fitness**	**Precision**	**F-score**	**(3-fold)**
IM	**1.00 ± 0.01**	0.73 ± 0.30	0.80 ± 0.24	**1.00 ± 0.01**
ETM	0.90 ± 0.08	0.92 ± 0.09	0.91 ± 0.07	0.90 ± 0.07
HM$_6$	**1.00 ± 0.01**	0.98 ± 0.04	**0.99 ± 0.03**	**1.00 ± 0.01**
S-HM$_6$	**1.00 ± 0.01**	**0.99 ± 0.04**	**0.99 ± 0.03**	**1.00 ± 0.01**
IM	**0.98 ± 0.04**	0.71 ± 0.28	0.78 ± 0.23	**0.98 ± 0.04**
ETM	0.88 ± 0.09	0.87 ± 0.09	0.87 ± 0.06	0.87 ± 0.07
HM$_6$	-	-	-	-
S-HM$_6$	0.98 ± 0.05	**0.95 ± 0.12**	**0.96 ± 0.09**	0.98 ± 0.05

Table 2: Accuracy results on synthetic logs.

4.2. Results

Tables 2 and 3 report the average value and standard deviation for each quality measure across all discovery algorithms, on the synthetic dataset. Since the measurements of accuracy and generalization on unsound models are unreliable, we divided the results in two groups. The upper part of Tables 2 and 3 shows the results for those models for which HM$_6$ did discover a sound model, while the lower part shows the results for those models for which HM$_6$ returned an unsound model. In this latter case, we did not report the measurements for accuracy and generalization for HM$_6$.

When HM$_6$ generates sound models, its output already has high accuracy 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, due to the duplication introduced by the structuring. IM, despite having similarly high values of fitness and generalization, loses in precision with an average of 0.73 and a high standard deviation, meaning that the actual precision may be much better or worse depending on the specific input log. 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. Generalization is on the other hand lower than that obtained by all other methods.

As expected, the models discovered by IM and ETM are structured by construction. On the contrary, HM$_6$ produces models that are mostly unstructured, with an average structuredness of 0.43 (0.40 for the unsound models). However, these models are then maximally structured by S-HM$_6$, with an average structuredness of 0.94 (0.97 for the unsound ones).

The improvement of our method on top of HM$_6$ is substantial when the latter discovers unsound models. In this case, S-HM$_6$ does not only notably increase structuredness, but it also repairs unsoundness, allowing us to measure accuracy and generalization. More importantly, our method significantly outperforms IM in terms of precision, and ETM in terms of both fitness and precision, leading in both cases to the highest F-score across all artificial logs in our dataset. ETM strikes a better trade-off between accuracy and complexity compared to IM, but at the price of significantly longer execution times, due to the high complexity of this method.

As an illustration, Fig. 11 shows the BPMN model generated by IM, HM$_6$ and

Discovery	**Complexity**			**Exec.**
Method	**Size**	**CFC**	**Struct.**	**Time (sec)**
IM	**24 ± 8**	11 ± 5	**1.00 ± 0.00**	0.8 ± 0.9
ETM	25 ± 8	**8 ± 4**	**1.00 ± 0.00**	1,800 ± 0.0
HM$_6$	25 ± 8	11 ± 8	0.43 ± 0.18	**0.5 ± 0.4**
S-HM$_6$	29 ± 14	10 ± 6	0.94 ± 0.17	13 ± 45
IM	**21 ± 8**	10 ± 7	**1.00 ± 0.00**	0.4 ± 0.5
ETM	23 ± 8	**8 ± 5**	**1.00 ± 0.00**	1,800 ± 0.0
HM$_6$	26 ± 12	14 ± 9	0.40 ± 0.21	**0.3 ± 0.3**
S-HM$_6$	35 ± 19	14 ± 8	0.97 ± 0.10	19 ± 47

Table 3: Complexity and execution time results on synthetic logs.

S-HM$_6$ from one of the SAP R/3 logs and the corresponding quality measures.[11] 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, while the output of HM$_6$ is unsound. By structuring and fixing the soundness of this latter model, S-HM$_6$ scores a perfect 1 for both F-score and generalization.

Table 4 shows the results of the measurements on the models discovered from the real-life logs. As expected, all models discovered by IM and ETM are sound, given that they are structured by construction. HM$_6$ only discovered two sound models out of twelve, though S-HM$_6$ managed to repair seven out of the ten unsound models returned by HM$_6$. In line with the results on the synthetic logs, IM produced highly fitting models, scoring the best result in fitness on eight models out of twelve. Nonetheless, these models generally have low precision, ranging from 0.70 to as low as 0.18, with an outlier score of 1.00 in the BPIC13$_{cp}$ log. On the other hand, ETM outperformed all other methods in ten logs in terms of precision, ranging from a minimum of 0.76 to a maximum of 1.00. However, this is achieved at the expenses of fitness, which is sensibly lower than that obtained by the other methods, except in the BPIC13$_{cp}$ log where ETM scores the highest fitness.

This stark difference between fitness and precision for both IM and ETM (see e.g. the BPIC13$_{cp}$ and RTFMP logs), is counteracted by a better balance between the two measures, achieved by S-HM$_6$. In fact, our method scores the best F-score seven times out of twelve, against four times for ETM, and twice for IM and for HM. In particular, our method obtains the highest fitness in four cases, and the second highest fitness in all other cases for which such measure could be computed, with similar results obtained for generalization.

The complexity of the models obtained by our method is generally higher than that of the other methods (up to six times higher in the BPIC14$_f$ and SEPSIS logs), due to the structuring phase, which introduces more gateways and duplicates fragments. However, even if S-HM$_6$ only manages to fully structure four models out of twelve, in the remaining cases it increases, often substantially, the degree of structuredness w.r.t. HM$_6$ (see e.g. the BPIC12 log, where structuredness goes from 0.05 to 0.40). In two

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

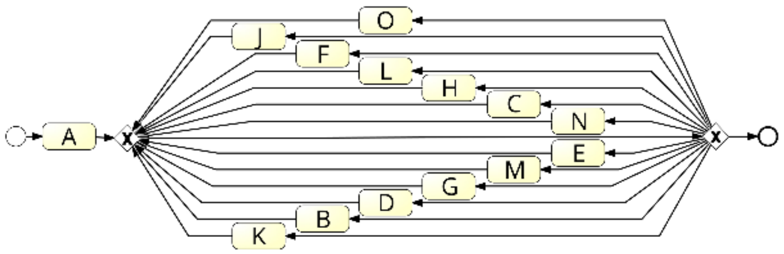
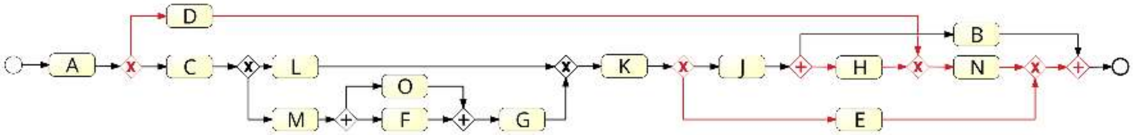
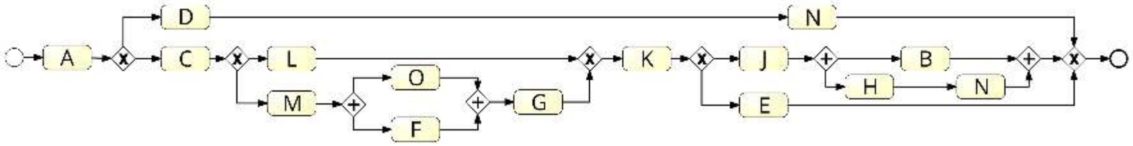
IM	Fitness: 1.00 Precision: 0.20 F-score: 0.33 Generaliz.: 1.00	 A O J F L H C N E M G D B K
HM_6	Fitness: - Precision: - F-score: - Generaliz.: -	 A D C L M O F G K J H B N E
S-HM_6	Fitness: 1.00 Precision: 1.00 F-score: 1.00 Generaliz.: 1.00	 A D C L M O F G K J B H N N E

Figure 11: Model discovered using IM, HM_6 and S-HM_6 (injections and gateways causing unsoundness in the HM model are highlighted) from one of the SAP R/3 logs.

logs (BPIC14$_\mathrm{f}$ and BPIC15$_\mathrm{1f}$), we were unable to measure the structuredness of the model produced by HM$_6$ because this was disconnected.

Figure 12 reports the model discovered by IM, HM$_6$ and S-HM$_6$ for the BPIC13$_\mathrm{cp}$ log. This is one of the cases where HM$_6$ produces an unsound model which is then fixed by the structuring phase of our method.

In the artificial logs, clone detection led on average to the removal of two cloned activities per model, with a maximum of fifteen activities in the best case. In the real-life logs, clone detection did not add any extra value.

In the above experiments we disabled the pull-up operator to ensure weak-bisimulation equivalence between the model discovered by S-HM$_6$ and its originating model obtained by HM$_6$. As a result, we could not fully structure 18 models in the synthetic dataset and 8 in the real-life dataset, which explains values of structuredness less than one for S-HM$_6$ in Tables 3 and 4. When we enabled the pull-up operator, all the discovered models from the synthetic dataset were indeed fully structured, at the price of losing weak bisimilarity, but this was not the case for the real-life logs, where the results did not sensibly improve.

Time performance. Despite having exponential complexity in the worst case scenario, the time our method took to structure the models used in this evaluation was within acceptable bounds, ranging from up to one minute in the synthetic dataset, to 4.5 minutes in the real-life dataset. In comparison, IM and HM$_6$ are much faster (taking less than one second per log in the synthetic dataset and less than 15 seconds per log in the real-life dataset), while ETM takes significantly longer (this method always timed out to 30 minutes in the synthetic dataset and to one hour in the real-life dataset).

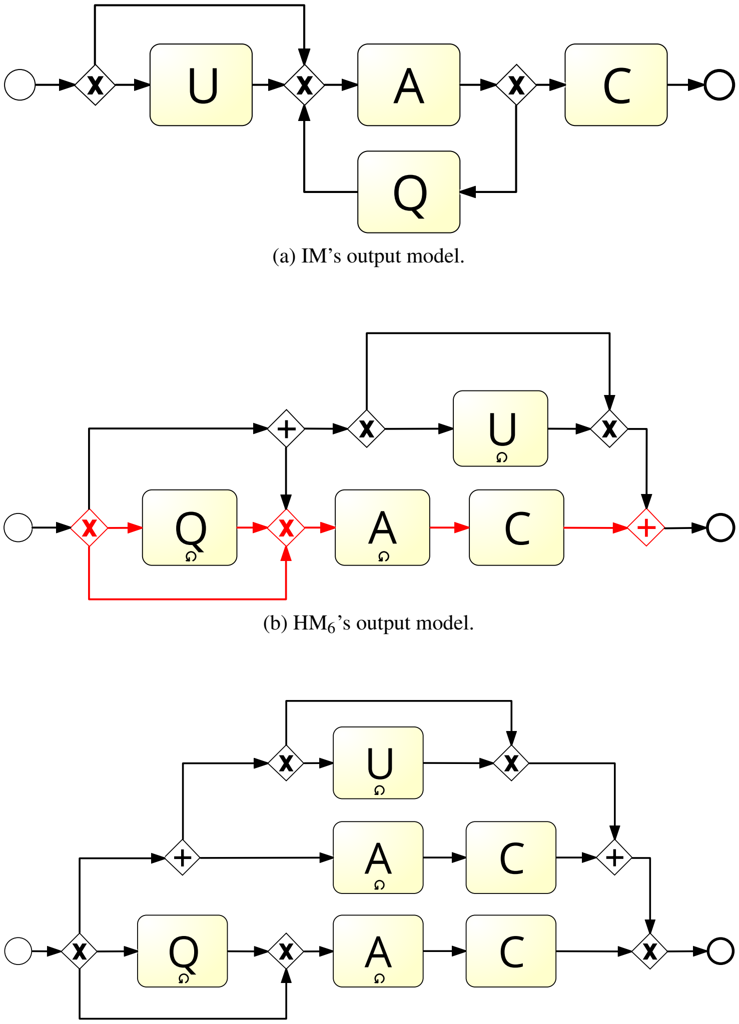
4.3. Threats to Validity

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 judgment. However, the three chosen complexity metrics (size, CFC and structuredness) have been empirically shown to be highly correlated with perceived understandability and error-proneness [12, 20, 21]. Further, while the process models obtained with our method are affected by the individual accuracy (fitness and precision) and generalization of the base discovery algorithm used, Structured Miner is independent of these algorithms, and our experiments show that the method always improves on structuredness while keeping at least the same level of accuracy and generalization. In addition, the method frequently fixes issues related to soundness.

The choice of a large range of varied real-life logs, originating from different domains, contributes to the external validity of the results. These logs are publicly available, as well as the artificial logs that we generated, so the experiments are fully reproducible. In addition, the great majority of the artificial logs used in the first experiment, originate from two real-life process model collections. Finally, the use of a synthetic dataset allowed us to evaluate our method against a large variety of unstructured model topologies, including some complex ones not observed in the real-life dataset.

Log	**Discovery**	**Accuracy**			**Gen.**	**Complexity**				**Exec.**
Name	**Method**	**Fit.**	**Prec.**	**F-score**	**(3-Fold)**	**Size**	**CFC**	**Struct.**	**Sound?**	**Time(sec)**
BPIC12	IM	**0.98**	0.50	0.66	**0.98**	**59**	37	**1.00**	yes	6.6
	ETM	0.33	**0.98**	0.49	0.38	69	**10**	**1.00**	yes	3,600
	HM$_6$	-	-	-	-	85	99	0.05	no	**2.5**
	S-HM$_6$	-	-	-	-	88	46	0.40	no	227.8
BPIC13$_{cp}$	IM	0.82	**1.00**	0.90	0.82	**9**	**4**	**1.00**	yes	**0.1**
	ETM	**0.99**	0.76	0.86	**0.99**	11	17	**1.00**	yes	3,600
	HM$_6$	-	-	-	-	12	6	0.67	no	**0.1**
	S-HM$_6$	0.94	0.99	**0.97**	0.94	15	6	**1.00**	yes	130.0
BPIC13$_{inc}$	IM	**0.92**	0.54	0.68	**0.92**	13	7	**1.00**	yes	1.0
	ETM	0.84	0.80	0.82	0.88	28	24	**1.00**	yes	3,600
	HM$_6$	0.91	**0.96**	**0.93**	0.91	**9**	**4**	**1.00**	yes	**0.8**
	S-HM$_6$	0.91	**0.96**	**0.93**	0.91	**9**	**4**	**1.00**	yes	**0.8**
BPIC14$_f$	IM	**0.89**	0.64	0.74	**0.89**	31	18	**1.00**	yes	3.4
	ETM	0.68	**0.94**	**0.79**	0.57	**22**	**15**	**1.00**	yes	3,600
	HM$_6$	-	-	-	-	43	51	-	no	**3.3**
	S-HM$_6$	-	-	-	-	202	132	0.73	no	147.4
BPIC15$_{1f}$	IM	**0.97**	0.57	**0.71**	**0.96**	164	108	**1.00**	yes	0.6
	ETM	0.57	**0.89**	0.69	0.56	**73**	**21**	**1.00**	yes	3,600
	HM$_6$	-	-	-	-	150	98	-	no	**0.5**
	S-HM$_6$	-	-	-	-	204	116	0.56	no	128.1
BPIC15$_{2f}$	IM	0.93	0.56	0.70	0.94	193	123	**1.00**	yes	**0.7**
	ETM	0.62	**0.90**	0.73	0.57	**78**	**19**	**1.00**	yes	3,600
	HM$_6$	-	-	-	-	194	158	0.11	no	**0.7**
	S-HM$_6$	**0.98**	0.59	**0.74**	**0.97**	259	150	0.29	yes	163.2
BPIC15$_{3f}$	IM	**0.95**	0.55	0.70	**0.95**	159	108	**1.00**	yes	1.3
	ETM	0.66	**0.88**	0.75	0.64	**78**	**26**	**1.00**	yes	3,600
	HM$_6$	**0.95**	0.67	**0.79**	**0.95**	157	151	0.07	yes	**0.8**
	S-HM$_6$	**0.95**	0.67	**0.79**	**0.95**	159	151	0.13	yes	139.9
BPIC15$_{4f}$	IM	0.96	0.58	0.73	0.96	162	111	**1.00**	yes	0.7
	ETM	0.66	**0.95**	**0.78**	0.63	**74**	**17**	**1.00**	yes	3,600
	HM$_6$	-	-	-	-	156	127	0.13	no	**0.5**
	S-HM$_6$	**0.99**	0.64	**0.78**	**0.99**	209	137	0.37	yes	136.9
BPIC15$_{5f}$	IM	0.94	0.18	0.30	0.94	134	95	**1.00**	yes	1.5
	ETM	0.58	**0.89**	0.70	0.56	**82**	**26**	**1.00**	yes	3,600
	HM$_6$	-	-	-	-	166	124	0.15	no	**1.2**
	S-HM$_6$	**1.00**	0.70	**0.82**	**1.00**	211	135	0.35	yes	141.9
BPIC17$_f$	IM	**0.98**	0.70	0.82	**0.98**	35	20	**1.00**	yes	13.3
	ETM	0.72	**1.00**	**0.84**	0.82	31	**5**	**1.00**	yes	3,600
	HM$_6$	-	-	-	-	**29**	10	0.45	no	**6.5**
	S-HM$_6$	0.95	0.62	0.75	0.94	42	13	0.97	yes	143.2
RTFMP	IM	**0.99**	0.70	0.82	**0.99**	**34**	**20**	**1.00**	yes	10.9
	ETM	0.79	**0.98**	0.87	0.81	46	33	**1.00**	yes	3,600
	HM$_6$	-	-	-	-	47	50	0.06	no	**7.8**
	S-HM$_6$	0.98	0.95	**0.96**	0.98	163	97	**1.00**	yes	262.7
SEPSIS	IM	**0.99**	0.45	0.62	**0.96**	50	32	**1.00**	yes	0.4
	ETM	0.71	**0.84**	**0.77**	0.70	**30**	**15**	**1.00**	yes	3,600
	HM$_6$	-	-	-	-	81	132	0.17	no	**0.03**
	S-HM$_6$	0.92	0.42	0.58	0.92	279	198	**1.00**	yes	242.7

Table 4: Accuracy and complexity results on real-life logs.



(a) IM's output model.

(b) HM_6's output model.

(c) S-HM_6's output model.

Figure 12: Models discovered from log $\mathrm{BPIC13}_{\mathrm{cp}}$.

5. Conclusion

This article presented a discover-and-structure method to generate a structured process model from an event log. The method builds upon the hypothesis that, instead of attempting to discover a block-structured process model directly, higher-quality process models can be obtained by first discovering an initial, potentially unstructured process model, and then transforming it into a structured one in a best-effort manner.

The experimental results support to a large extent this hypothesis. The experiments show that the discover-and-structure method generally leads to higher F-score relative to two existing methods that discover a structured process model by construction. In addition, the discover-and-structure method is more modular, insofar as different discovery and structuring methods can be plugged into it. The experiments also reveal that the structuring phase of the proposed method significantly improves the F-score relative to the models produced by the base discovery algorithm employed in the discovery phase. As a by-product, the structuring phase turns most of the unsound models produced by the first phase into sound ones, although soundness is not always achieved.

On the other hand, the proposed method partially inherits from the limitations of the base algorithm employed in the discovery phase. The experiments shows that in those cases where the first phase led to an imprecise model or a spaghetti-like model, the structuring phase could not fully structure the model nor repair its unsoundness. Another weakness exposed by the experiments is that when structuring an unstructured process model, the size of the model increases, often substantially due to the addition of new gateways and the duplication of model fragments. This is an inherent limitation that is only slightly mitigated by the removal of exact clones. Another weakness is time performance, which, while manageable, is in the order of minutes for the real-life logs included in the experiments.

A natural avenue for future work is to address these weaknesses. Given that there is a tradeoff between duplication and structuredness [12], it may be possible to improve the proposed method by adaptively stopping the structuring procedure when it is found that the benefits of continuing (higher structuredness) are not offset by the cost (larger size). Optimization heuristics could be applied to manage this tradeoff. Another direction for future work is to incorporate a more robust method for eliminating behavioral errors in the process models produced in the discovery stage. A number of methods for repairing unsound process models have been proposed [10, 14], which if suitably adapted, may be able to eliminate some of the behavioral errors that cannot be eliminated by the proposed method.

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*, Proc. of bpm workshops, 2012, pp. 137–149.

[2] A. Adriansyah, B.F. van Dongen, and W.M.P. van der Aalst, *Conformance checking using cost-based fitness analysis*, Proc. of edoc, 2011.

[3] A Augusto, R. Conforti, M. Dumas, M. La Rosa, and G. Bruno, *Automated discovery of structured process models: Discover structured vs. discover and structure*, Proc. of er, 2016, pp. 313–329.

[4] Yngvi Björnsson, Vadim Bulitko, and Nathan R Sturtevant, *TBA*: Time-bounded A**, Proc. of IJCAI, 2009, pp. 431–436.

[5] 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*, Proc. of coopis, 2012, pp. 305–322.

[6] R. Conforti, M. Dumas, L. García-Bañuelos, and M. La Rosa, *BPMN Miner: Automated discovery of BPMN process models with hierarchical structure*, Information Systems **56** (2016), 284–303.

[7] R. Conforti, M. Dumas, L. García-Bañuelos, and M. La Rosa, *Beyond tasks and gateways: Discovering BPMN models with subprocesses, boundary events and activity markers*, Proc. of bpm, 2014.

[8] R. Conforti, M. La Rosa, and A.H.M. ter Hofstede, *Filtering out infrequent behavior from business process event logs*, IEEE Trans. Knowl. Data Eng. **29** (2017), no. 2.

[9] T. Curran and G. Keller, *Sap r/3 business blueprint: Understanding the business process reference model*, Upper Saddle River, 1997.

[10] J. Dehnert and A. Zimmermann, *Making workflow models sound using petri net controller synthesis*, Proc. of OTM conferences, 2004, pp. 139–154.

[11] M. Dumas, L. García-Bañuelos, M. La Rosa, and Reina Uba, *Fast detection of exact clones in business process model repositories*, Inf. Syst. **38** (2013), no. 4, 619–633.

[12] 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*, Proc. of caise, 2012, pp. 31–46.

[13] D. Fahland, C. Favre, J. Koehler, N. Lohmann, H. Völzer, and K. Wolf, *Analysis on demand: Instantaneous soundness checking of industrial business process models*, Data Knowl. Eng. **70** (2011), no. 5, 448–466.

[14] M. Gambini, M. La Rosa, S. Migliorini, and A. H. M. ter Hofstede, *Automated error correction of business process models*, Proc. of bpm, 2011, pp. 148–165.

[15] 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** (1968), no. 2, 100–107.

[16] R. Kohavi, *A study of cross-validation and bootstrap for accuracy estimation and model selection*, Proc. of ijcai, 1995, pp. 1137–1145.

[17] M. La Rosa, H.A. Reijers, W.M.P. van der Aalst, R.M. Dijkman, J. Mendling, M. Dumas, and L. García-Bañuelos, *APROMORE: An Advanced Process Model Repository*, Expert Syst. Appl. **38** (2011), no. 6.

[18] S.J.J. Leemans, D. Fahland, and W.M.P. van der Aalst, *Discovering block-structured process models from event logs - a constructive approach*, Proc. of petri nets, 2013.

[19] J. Li, R.P.J.C. Bose, and W.M.P. van der Aalst, *Mining context-dependent and interactive business process maps using execution patterns*, Proc. of bpm workshops, 2011.

[20] J. Mendling, *Metrics for process models: Empirical foundations of verification, error prediction, and guidelines for correctness*, Springer, 2008.

[21] Jan Mendling, Hajo A Reijers, and Wil MP van der Aalst, *Seven process modeling guidelines (7pmg)*, Information and Software Technology **52** (2010), no. 2, 127–136.

[22] T. Molka, D. Redlich, W. Gilani, X.-J. Zeng, and M. Drobek, *Evolutionary computation based discovery of hierarchical business process models*, Proc. of bis, 2015, pp. 191–204.

[23] G. Oulsnam, *Unravelling unstructured programs*, Comput. J. **25** (1982), no. 3, 379–387.

[24] ———, *The algorithmic transformation of schemas to structured form*, Comput. J. **30** (1987), no. 1, 43–51.

[25] A. Polyvyanyy, L. García-Bañuelos, and M. Dumas, *Structuring acyclic process models*, Inf. Syst. **37** (2012), no. 6, 518–538.

[26] A. Polyvyanyy, L. García-Bañuelos, D. Fahland, and M. Weske, *Maximal structuring of acyclic process models*, Comput. J. **57** (2014), no. 1, 12–35.

[27] A. Polyvyanyy, J. Vanhatalo, and H. Völzer, *Simplified computation and generalization of the refined process structure tree*, Proc. of WS-FM, 2010, pp. 25–41.

[28] Stuart J Russell, *Efficient memory-bounded search methods.*, Ecai, 1992, pp. 1–5.

[29] R. Uba, M. Dumas, L. García-Bañuelos, and M. La Rosa, *Clone detection in repositories of business process models*, Business process management, 2011, pp. 248–264.

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

[31] 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** (2004), no. 9.

[32] 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** (2012), no. 7.

[33] A.J.M.M. Weijters and J.T.S. Ribeiro, *Flexible Heuristics Miner (FHM)*, Proc. of cidm, 2011.