Stage-based Discovery of Business Process Models from Event Logs

Hoang Nguyen[a,b], Marlon Dumas[c], Arthur H.M. ter Hofstede[b], Marcello La Rosa[a], Fabrizio Maria Maggi[c]

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

Abstract

An automated process discovery technique generates a process model from an event log recording the execution of a business process. For it to be useful, the generated process model should be as simple as possible, while accurately capturing the behavior recorded in, and implied by, the event log. Most existing automated process discovery techniques generate flat process models. When confronted to large event logs, these approaches lead to overly complex or inaccurate process models. An alternative is to apply a divide-and-conquer approach by decomposing the process into stages and discovering one model per stage. It turns out, however, that existing divide-and-conquer process discovery approaches often produce less accurate models than flat discovery techniques, when applied to real-life event logs. This article proposes an automated method to identify business process stages from an event log and an automated technique to discover process models based on a given stage-based process decomposition. An experimental evaluation shows that: (i) relative to existing automated process decomposition methods in the field of process mining, the proposed method leads to stage-based decompositions that are closer to decompositions derived by human experts; and (ii) the proposed stage-based process discovery technique outperforms existing flat and divide-and-conquer discovery techniques with respect to well-accepted measures of accuracy and achieves comparable results in terms of model

Email addresses: `hoang.nguyen1@unimelb.edu.au` (Hoang Nguyen), `marlon.dumas@ut.ee` (Marlon Dumas), `a.terhofstede@qut.edu.au` (Arthur H.M. ter Hofstede), `marcello.larosa@unimelb.edu.au` (Marcello La Rosa), `f.m.maggi@ut.ee` (Fabrizio Maria Maggi)

complexity.

Keywords: Process mining, automated process discovery, modularity.

1. Introduction

Modern organizations generally execute their business processes on top of process-aware information systems, such as Enterprise Resource Planning (ERP) systems, Customer Relationship Management (CRM) systems, and Business Process Management Systems (BPMS), among others [1]. These systems record a wealth of events that occur during the execution of the processes they support, including events signaling the creation and completion of business process instances (herein called *cases*) and the start and completion of activities within each case. These event records can be extracted and pre-processed to produce business process event logs [2]. A business process event log consists of a set of traces, each trace consisting of the sequence of event records produced by the execution of one case. Each event record captures relevant data about the execution of a given activity in the process.

Process mining is a family of techniques to extract insights and knowledge from business process event logs [2]. Among other things, process mining techniques allow us to automatically discover a business process model from an event log – an operation known as *automated process discovery*. For it to be useful, an automatically discovered process model must accurately reflect the behavior recorded in or implied by the log. Specifically, the process model should recognize (i.e. parse) as much as possible the behavior in the event log, while not allowing too much behavior that has not been observed in the log. The former property is called *fitness* while the latter is called *precision*. At the same time, the model should be as simple as possible, a property usually quantified via *complexity* measures.

Traditional automated process discovery techniques are designed to produce a single "flat" process model from an event log. Examples of such techniques include the Heuristics Miner [3], the Inductive Miner [4] and Fodina (FO) [5]. When applied to real-life logs, these techniques produce overly large and complex process models, oftentimes with either low fitness or low precision [6]. For example, when applied to the event log of the 2012 Business Process Intelligence Challenge (BPI12) [7] – an event log of a loan origination process at a bank – the Inductive Miner produces a process model with a precision below 0.1. After applying a filtering step to simplify the event log and hyper-parameter optimization to find the best settings of the Inductive

Miner on this log, the precision raises to 0.77, which is relatively low for a simplified event log of a well-scoped and well-defined process.

A natural approach to tackle these weaknesses is to apply a divide-and-conquer approach. A divide-and-conquer automated discovery technique starts by decomposing a log into a set of sublogs, then discovers one process model per sublog, and finally merges the resulting process models into a single one [8, 9, 10]. But while existing divide-and-conquer approaches speed-up the execution time of automated process discovery compared to flat discovery techniques, it turns out that they often have a negative impact on the quality of the resulting model, both in terms of accuracy and simplicity. For example, if we apply the Decomposed Process Miner [10] to the aforementioned BPI12 log, the precision is less than 0.1 on the unfiltered event log, and it only raises to 0.35 on the filtered one. A similar observation can be made about the divide-and-conquer approach proposed in [8].

Inspired by previous studies on the benefits of modularity in process modeling [11], this article argues that in order to enhance the quality of process models produced by a divide-and-conquer approach, we need to decompose the process in a way that maximizes modularity. To develop this argument, the article proposes: (i) an automated method to identify stages from an event log based on a modularity measure; and (ii) an automated technique to discover a process model from an event log based on a given decomposition of the activities in the event log into stages. The article then reports on an experimental evaluation designed to test two hypotheses: (i) relative to existing automated process decomposition methods in the field of process mining, the proposed stage-based decomposition method identifies stages that are closer to decompositions derived by human experts; and (ii) the proposed stage-based process discovery technique (namely the *Staged Process Miner*) outperforms existing flat and divide-and-conquer discovery techniques with respect to well-accepted measures of accuracy (fitness and precision) while achieving comparable results in terms of model complexity.

This article is an extension of a previous conference paper [12]. The conference paper introduced the method for identifying stages from an event log and evaluated the first hypothesis postulated above, namely that automated decompositions that maximize modularity are closer to human expert decompositions compared to automated decompositions that do not take into account modularity. This article extends this previous result by presenting an automated process discovery technique based on a given stage decomposition and validating the second of the above hypotheses.

The rest of the paper is organized as follows. Section 2 introduces existing automated process discovery techniques. Section 3 outlines the stage decomposition method while Section 4 presents the stage-based process discovery technique. Next, Section 5 discusses an empirical evaluation of the entire approach. Finally, Section 6 concludes the paper and spells out directions for future work.

2. Related Work

Process discovery techniques available in the literature can be classified into techniques that produce flat process models and techniques that generate decomposed models. The Alpha Miner [13] is one of the earliest techniques falling in the first group. The Alpha Miner has several limitations, e.g. it cannot deal with short loops, non-local relations and noise. To overcome these limitations, the Heuristics Miner was proposed in [3]. The main idea of the Heuristics Miner is to use a set of heuristics to deal with noisy data. Recent improvements of the Heuristics Miner include Fodina [5] and the Structured Miner [14]. Fodina enhances the Heuristics Miner with improved causal reasoning and more robust capability to deal with noise and duplicate activities. The Structured Miner is an extension of the Heuristics Miner, which tries to fix unsound constructions in the output of the Heuristics Miner and to convert the resulting process model into a block-structured process model.[1] However, Structured Miner uses a best-effort approach and it sometimes produces non-block-structured and unsound models (as Fodina does). Inductive Miner [4] is an automated process discovery technique that produces exclusively block-structured process models and hence guarantees soundness, sometimes at the expense of precision as discussed above. Other techniques such as the Genetic Miner [15], the Evolutionary Tree Miner [16], and the ILP Miner [17] also produce flat models. However, they suffer from long execution times when applied to large and complex event logs.

One of the earliest approaches that produces decomposed models (also known as divide-and-conquer approaches) is the region-based process discovery technique [8]. This technique applies graph cuts to a transition system built from the event log, and then recursively discovers a Petri net for each subgraph. Other techniques that produce decomposed models are presented

[1] A block-structured process model is a process model such that every split gateway has a unique corresponding join gateway and vice-versa. A block-structured process models can be represented as a tree where the internal nodes are operators such as Sequence, XOR, AND, and Loop and the leaves correspond to activities of the process.

in [18, 19]. The technique proposed in [18] is based on passages that can be used to localize and decompose the discovery problem into smaller problems. In [19], the authors propose an approach that employs approximate functional and inclusion dependency discovery techniques in order to elicit a process-subprocess hierarchy. Recently, the problem of divide-and-conquer process discovery has been formalized [9] and validated with a specific implementation in the Decomposed Miner [10]. As shown in the experimental evaluation reported later in this article, the Decomposed Miner (DM) produces relatively imprecise process models (i.e. process models with high levels of additional behavior) and unsound models, when applied to real-life logs.

In addition to the aforementioned decomposed process discovery approaches, there are other automated process discovery techniques that use trace clustering to discover a collection of process models from an event log, such that each discovered model describes a subset of the traces in the log [20, 21, 22]. Also, approaches exist in the literature that tackle the opposite problem of discovering one single process model from collections of logs. These approaches can be classified into two main categories: the ones that merge the logs before mining (e.g. [23, 24]) and the ones that merge the process models obtained by mining the logs separately (e.g. [25, 26]). In this article, we address the problem of discovering one single process model that describes the behavior of one single event log, as opposed to discovering multiple models that describe the behavior of different subsets of the log or discovering one single model from a collection of logs.

3. Stage Decomposition Method

The proposed method for extracting stages from an event log proceeds in two steps. In the first step, we construct a weighted graph from the event log capturing the directly-follows relation between activities in the process. In the second step, we split the nodes in the graph (i.e. the activities) into stages with the aim of maximizing a modularity measure. Below we introduce each of these two steps in detail.

3.1. From Event Log to Directly-Follows Graph

Our approach takes as input an event log defined as follows.

Definition 1 (Event Logs). *Given a set of events E, an event log EL is a multiset of cases C, where a case $c \in C$ is a sequence of events, i.e. $c = \langle e_1, e_2, \ldots, e_n \rangle$, with $e_i \in E, 1 \leq i \leq n$. Each event is associated with an* *activity label $a \in A$, which refers to an activity performed within a process. We retrieve the activity label and the case of an event with functions $act : E \rightarrow A$ and $case : E \rightarrow C$.*

Given a set of activity labels $A = \{A, B, C, D, E, F, G, H, U\}$, a possible log EL is given in Table 1. This log contains 9 distinct traces, where each distinct trace is called a trace variant. Each variant has a number of occurrences in the log.

Variant	**Trace**	**Occurrences**
1	ABCUEF	10
2	ADUG	10
3	ABEFADUG	4
4	ACDUG	1
5	ABUG	1
6	ABDUG	1
7	ADUH	2
8	ADUHE	1
9	ADUGF	1
10	ADUK	1

Table 1: Example event log

A *process graph* is a directed graph in which *nodes* represent activities and *edges* represent directly-follows relations between activities. For example, if activity b occurs after activity a in a case, the graph contains a node a, a node b and a directed edge from a to b. In addition, edges carry *weights* representing the frequency of the directly-follows relation between two related activities in the log.

Definition 2 (Process Graph). *A process graph of an event log EL=(E, ET, A, C, time, act, type, case) is a graph $G_{EL} = (V_{EL}, F_{EL}, W_{EL})$, where:*

- *V_{EL} is a set of nodes, each representing an activity, i.e. $V_{EL} = A$.*
- *F_{EL} is a set of directed edges, each representing the directly-follows relation between two activities based on events. Activity a_2 directly follows activity a_1 if there is a case in which the event e_2 of a_2 follows the event e_1 of a_1 without any other events in-between, i.e. e_1 is in a*

direct sequence with e_2. *Event* e_1 *is in a direct sequence with* e_2, *denoted* $e_1 \longrightarrow e_2$, *iff* $e_1 \in E \wedge e_2 \in E \wedge e_1 \neq e_2 \wedge case(e_1) = case(e_2) \wedge e_1 \lesssim_E e_2 \wedge \nexists e_3 \in E[e_3 \neq e_1 \wedge e_3 \neq e_2 \wedge case(e_3) = case(e_1) \wedge e_1 \lesssim_E e_3 \wedge e_3 \lesssim_E e_2]$. *Thus,* $F_{EL} = \{(a_1, a_2) \in V_{EL} \times V_{EL} | \exists e_1, e_2 \in E[act(e_1) = a_1 \wedge act(e_2) = a_2 \wedge e_1 \longrightarrow e_2]\}$.

- W_{EL} *is a function that assigns a weight to an edge,* W_{EL}: $F_{EL} \rightarrow \mathbb{N}_0^+$. *The weight of an edge connecting node* a_1 *to node* a_2, *denoted* $W_{EL}(a_1, a_2)$, *is the frequency of the directly-follows relation between* a_1 *and* a_2 *in the log, i.e.* $W_{EL}(a_1, a_2) = |\{(e_1, e_2) \in E \times E | act(e_1) = a_1 \wedge act(e_2) = a_2 \wedge e_1 \longrightarrow e_2\}|$.

The process graph constructed above has a set of start nodes called *firstacts* containing the first activities of all cases, and a set of end nodes called *lastacts* containing the last activities of all cases, i.e. $firstacts(V_{EL}) = \{a \in V_{EL} | \exists e \in E\colon [act(e) = a \wedge \nexists e' \in E | e' \longrightarrow e]\}$, and $lastacts(V_{EL}) = \{a \in V_{EL} | \exists e \in E\colon [act(e) = a \wedge \nexists e' \in E | e \longrightarrow e']\}$.

From a process graph, we can derive a corresponding *directly-follows graph*, which has only one source node i and one sink node o.

Definition 3 (Directly-Follows Graph - DFG). *The DFG of a process graph* $G_{EL} = (V_{EL}, F_{EL}, W_{EL})$ *is a graph* $FL(G_{EL}) = (V_{EL}^{FL_G}, F_{EL}^{FL_G}, W_{EL}^{FL_G})$, *where:*

- $V_{EL}^{FL_G} = V_{EL} \cup \{i, o\}, \{i, o\} \cap V_{EL} = \varnothing$.
- $F_{EL}^{FL_G} = F_{EL} \cup \{(i, x) | x \in firstacts(V_{EL})\} \cup \{(x, o) | x \in lastacts(V_{EL})\}$
- $W_{EL}^{FL_G}(a_1, a_2) = \begin{cases} W_{EL}(a_1, a_2) & \text{if } a_1 \neq i \wedge a_2 \neq o \\ |\{e \in E | act(e) = a_2 \wedge [\nexists e' \in E | e' \longrightarrow e]\}| & \text{if } a_1 = i \\ |\{e \in E | act(e) = a_1 \wedge [\nexists e' \in E | e \longrightarrow e']\}| & \text{if } a_2 = o \end{cases}$

Figure 1 illustrates a *DFG* constructed from the example log in Table 1.

3.2. Stage Decomposition and Quality Measure

Given a *DFG*, we seek to partition it into a sequence of fragments (which we will call stages), each consisting of a subset of the nodes in the *DFG*, such that the stages correspond to *quasi-SESE* (single entry single exit) fragments of the *DFG*. Here, a quasi-SESE fragment is one that has an identifiable

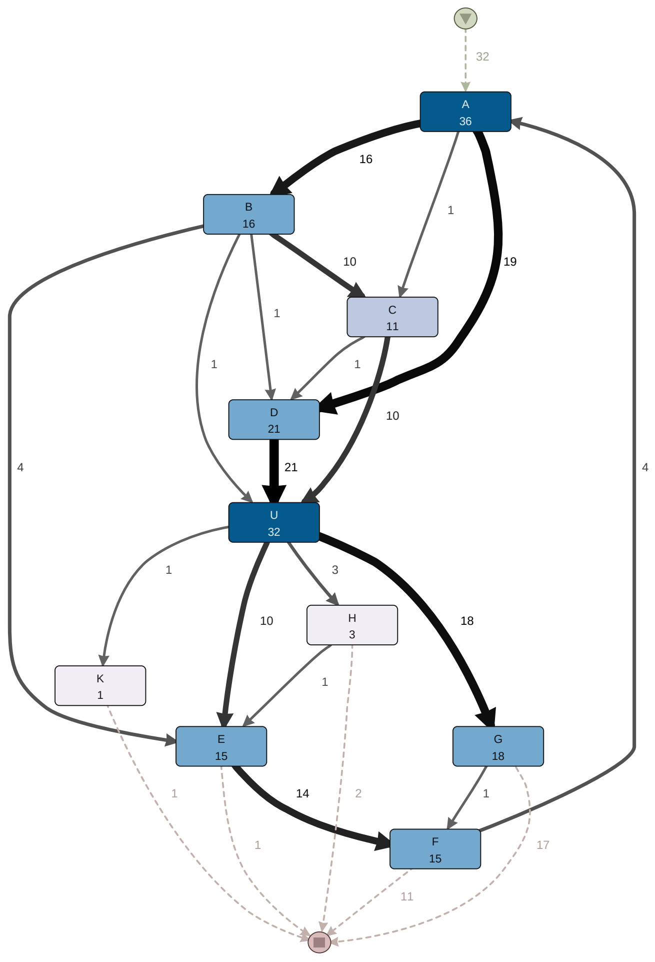


Figure 1: *DFG* created from the event log in Table 1 using the Disco tool.

entry node and an identifiable exit node, such that the entry node has a high inflow and the exit node has a high outflow, where the inflow (outflow) of a node is defined as the total weight of its incoming (outgoing) arcs. The entry and the exit nodes, which connect consecutive stages, are called *milestones*. By *identifiable*, we mean that the entry and the exit nodes are the main connection points between a stage and its preceding and/or succeeding stages. There may be arcs going from a node in one stage directly to a node in another stage, but these arcs should have a low weight relative to the milestones.

Figure 1 shows an example *DFG* created from the log in Table 1 with

U as the milestone node. We aim at developing a method to extract a list of stages from a *DFG* called a *stage decomposition* as shown in Figure 2, where stages are *non-overlapping* sets of nodes with one node playing as the milestone. The main flow of the process goes from the source node (i) through stages and milestone nodes to the sink node (o). There are also minor flows connecting stages and bypassing the milestone nodes.

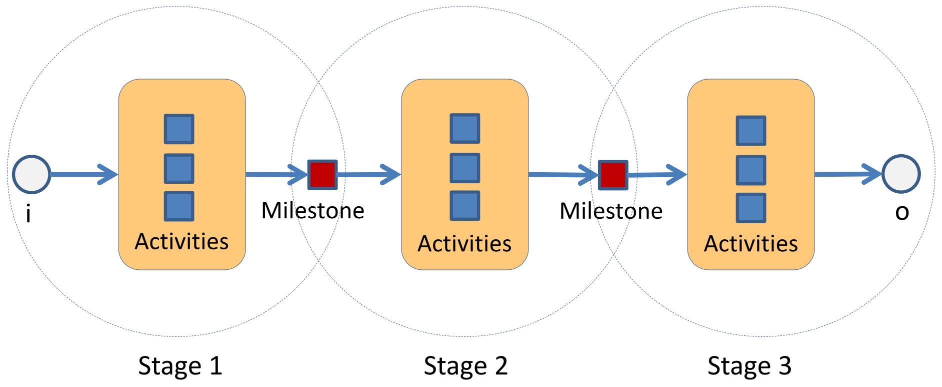


Figure 2: Stage decomposition (arrow lines represent the main flow)

In order to measure the quality of a stage decomposition, we reuse a measure of *modularity* [27] previously proposed for detecting community structures in social networks. A community structure is characterized by a high density of edges within a community and a low number of edges connecting different communities. The higher the modularity is, the more a network exhibits a community structure. Intuitively, this is a key property we seek in a stage-based decomposition of a process. A stage-based decomposition should be such that the density of connections between nodes inside a stage is high relative to the density of connections between nodes across stages. Note that despite having been developed in the field of social networks, the measure of modularity in [27] does not make any assumption on the semantics of the arcs in the graph. In particular, it does not assume that the edges in the graph correspond to social relations.

We slightly adapt the modularity measure in [27] to make it applicable to weighted and directed graphs, which are characteristics of *DFG*s. Concretely, let S be a stage decomposition extracted from a *DFG* based on an event log EL, and $S_i \in S$, where $i = 1 \ldots |S|$, be a stage. Let $W_{EL}^{FL_G}(S_i, S_j)$ be the total weight of edges connecting S_i to S_j (excluding self-loops), $W_{EL}^{FL_G}(S_i, S_j) = \sum_{a_1 \in S_i, a_2 \in S_j, a_1 \neq a_2} W_{EL}^{FL_G}(a_1, a_2)$. Let W^T be the total weight of all edges in the graph excluding self-loops, $W^T = \sum_{a_1,a_2 \in V_{EL}^{FL_G}, a_1 \neq a_2} W_{EL}^{FL_G}(a_1, a_2)$. Self-loops are excluded in this definition because they should have a neutral effect on modularity. Indeed, the chosen notion of modularity captures the proportion of connections between two nodes inside a module (i.e. stage) relative to the connections between two nodes located in different modules. A self-loop does not involve two nodes and hence should not affect the modularity.

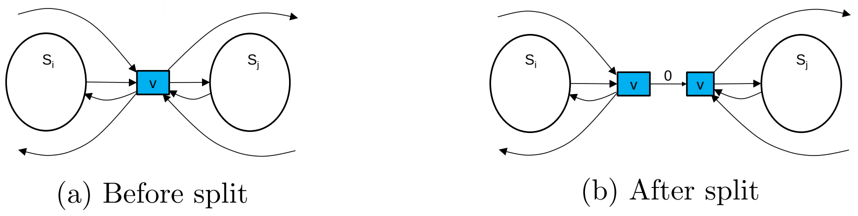


Figure 3: Division of the milestone node for computing modularity

Let $W_{EL}^{FL'_G}(S_i, S_j)$ be the total weight of edges connecting S_i to S_j in the modular graph. The modularity of a stage decomposition S is computed as follows.

$$Q = \sum_{i=1}^{|S|} (E_i - A_i^2), \tag{1}$$

where $E_i = \frac{W_{EL}^{FL'_G}(S_i,S_i)}{W^T}$ is the fraction of edges that connect nodes within stage S_i and $A_i = \frac{\sum_{j=1}^{|S|} W_{EL}^{FL'_G}(S_j,S_i)}{W^T}$ is the fraction of edges that connect to stage S_i, including those within stage S_i and those from other stages.

A minor technical issue arises from the fact that a milestone node effectively belong to two stages, since milestone nodes are the boundary between successive stages. Hence, before computing the modularity of a stage decomposition, we duplicate each milestone node into two consecutive nodes. The arc connecting these two nodes is given a weight of zero, in order not to affect the modularity. The modularity measure is then computed on the *adjusted DFG* obtained after duplicating the milestones in the stage decomposition.

A stage decomposition may end up being too fragmented. In the worst-case, each fragment could consist of only two connected activities. To avoid over-fragmentation, we introduce a parameter, namely the*minimum stage* *size*, corresponding to the smallest number of activities in any given stage. In this way, the user can tune the granularity of the stages.

3.3. Stage Decomposition Algorithm

Given an event log, we seek to find a stage decomposition with high modularity. To this end, we propose a method that starts from the *DFG* constructed from the log, and recursively decomposes it into sets of nodes using the notion of min-cut as calculated by the Ford-Fulkerson's algorithm. Note that the min-cut here is the one found in the graph after a node has been removed. The set of edges in that min-cut is called a *cut-set* associated with the removed node, and the total weight of edges in the cut-set is called *cut-value*. Together, a node and its cut-set form a boundary between two graph fragments. The lower the cut-value is, the more the related fragments will resemble quasi-SESE fragments. Therefore, if we find a set of nodes with low cut-values, we can take multiple graph cuts on those nodes and their cut-sets to obtain a stage decomposition that can approximate the maximum modularity.

Milestone nodes intuitively have lower cut-values than the min-cut found by the Ford-Fulkerson's algorithm in the original *DFG* (called *source-min-cut* and equal to the number of traces in the log). Thus, we can use the source-min-cut as a threshold when selecting a candidate list of cut-points, i.e. nodes with cut-values smaller than that of the source-min-cut will be selected. Further, in a *DFG*, the source-min-cut can be computed in constant time as it is equal to the set of outgoing edges of the source node of the graph or the set of incoming edges of the sink node.

Once we have a candidate list, the key question is how to find a subset of nodes to form a stage decomposition that can maximize modularity. One way is to generate all possible subsets from the list, create stage decompositions based on all subsets, and select the one that has the highest modularity. However, this approach may suffer from combinatorial problems if the number of candidate nodes is large. For example, if we assume that the *DFG* has 60 nodes and the candidate list has 30 nodes, the total number of subsets would be $\binom{30}{1} + \binom{30}{2} + ... + \binom{30}{30} = 1{,}050{,}777{,}736$. We thus propose two algorithms (Algorithm 1 & 2) to find a stage decomposition that can approximate the maximum modularity. The inputs to the algorithms are an event log and a minimum stage size.

Algorithm 1 is a greedy algorithm. The main idea (Lines 9-22) is to search in the candidate list for a cut-point that can result in a stage decomposition with two stages and of highest modularity. The *find_stage* function searches

Algorithm 1: Highest Modularity Stage Decomposition

```
Input: EL: an event log
       minStateSize: minimum number of activities in a stage
Output: A list of stages, each is a set of activities
1  $G = create\_flow\_graph(EL)$
2  $CandidateNodes := \{\}$
3  forall $v$ in $V_{EL}^{FL_G} \setminus \{i, o\}$ do
4      $<v.mincut, v.cutset>:= node\_min\_cut(G, v)$
5      if $v.mincut < source\_min\_cut(G)$ then
6          $CandidateNodes := CandidateNodes \cup \{v\}$
7  $CurrentBestSD := [V_{EL}^{FL_G} \setminus \{i, o\}]$
8  $NewBestSD := CurrentBestSD$
9  while $CandidateNodes \neq \{\}$ do
10     forall $v$ in $CandidateNodes$ do
11         $CutStage := find\_stage(CurrentBestSD, v)$  // search for the stage containing $v$
12         $<PreStage, SucStage>:= cut\_graph(G, v, CutStage)$  // refer to Algorithm 3
13         if $|PreStage| \geq minStateSize$ and $|SucStage| \geq minStateSize$ then
14             $NewSD := copy\_sd(CurrentBestSD, CutStage, PreStage, SucStage)$
15             if $modularity(NewSD, G) > modularity(NewBestSD, G)$ then
16                 $NewBestSD := NewSD$
17                 $BestCutPoint := v$
18     if $NewBestSD \neq CurrentBestSD$ then
19         $CurrentBestSD := NewBestSD$
20         $CandidateNodes := CandidateNodes \setminus \{BestCutPoint\}$
21     else
22         break // stop when modularity is not increased
23 return $CurrentBestSD$
```

in the current decomposition for a stage that contains the candidate cut-point, then the *cut_graph* function (Algorithm 3) cuts the found stage into two substages, and then the *copy_sd* function creates a new decomposition from the current one by replacing the found stage with its two substages. Then it removes the node from the candidate list (Line 20) and searches in the list again for another cut-point that can create a new decomposition with three stages and of highest modularity, i.e. higher than the former decomposition and the highest among all decompositions with three stages, and so on until it cannot either find a stage decomposition of higher modularity or all new decompositions have a stage of smaller size than the minimum stage size. Note that stage decomposition is recursive meaning a stage in the current decomposition will be decomposed into two substages based on a selected cut-point (Line 14). Modularity is computed according to Equation 1 based on the modular graph as described above (Line 15).

Algorithm 3 also shows that each stage is marked with starting and ending

Algorithm 2: Lowest Cut-value Stage Decomposition

```
Input: $EL$: an event log
       $minStageSize$: minimum number of activities in a stage
Output: A list of stages, each is a set of activities
// Line 1-7 is the same as Algorithm 1
8  $Candidates\_sorted := sort(CandidateNodes, min\_cut, asc)$
9  while $Candidates\_sorted \neq []$ do
10     $v := head(Candidates\_sorted)$
11     $CutStage := find\_stage(CurrentBestSD, v)$ // search for the stage containing $v$
12     $<PreStage, SucStage> := cut\_graph(G, v, CutStage)$ // refer to Algorithm 3
13     if $|PreStage| \geq minStateSize$ and $|SucStage| \geq minStateSize$ then
14         $NewSD := copy\_sd(CurrentBestSD, CutStage, PreStage, SucStage)$
15         if $modularity(NewSD, G) > modularity(CurrentBestSD, G)$ then
16             $CurrentBestSD := NewSD$
17         else
18             break // stop when modularity is not increased
19     $Candidates\_sorted := tail(Candidates\_sorted)$
20 return $CurrentBestSD$
```

Algorithm 3: *cut_graph*

```
Input: $G$: a $DFG$
       $v$: a node
       $CutStage$: a node set containing $v$
Output: a pair of subsets of $CutStage$
1  $G\_aftercut := remove\_edges(remove\_node(G, v), v.cutset)$          // Graph cut
2  $G\_source := source\_graph(G\_aftercut)$          // The subgraph containing the source
3  $PreStage := (CutStage \cap V_{G\_source}) \cup \{v\}$
4  $SucStage := CutStage \setminus PreStage$
5  $PreStage.end := v$
6  $SucStage.start := v$
7  if $CutStage = V_G$ then
8      $PreStage.start := G.source$
9      $SucStage.end := G.sink$
10 else
11     $PreStage.start := CutStage.start$
12     $SucStage.end := CutStage.end$
13 return $<PreStage, SucStage>$
```

milestone nodes based on the selected cut-points. The first stage would take the source node of the *DFG* as its (artificial) starting milestone, and the last stage would take the sink node of the *DFG* as its (artificial) ending milestone. Two consecutive stages would share the cut-point, i.e. the ending milestone of the preceding stage and the starting milestone of the succeeding stage.

Algorithm 2 has the same structure as Algorithm 1, but uses the lowest cut-value as a heuristic. Firstly, it sorts the candidate list in ascending order of cut-values, then it sequentially picks every node from the list to create recursive stage decompositions until the modularity is not increased or all new decompositions have a stage of smaller size than the minimum threshold.

The worst-case time complexity of functions used in the algorithms can be computed as follows. The *create_flow_graph* function is $O(V+F)$, where $V = V_{EL}^{FL_G}$ and $F = F_{EL}^{FL_G}$. The *node_min_cut* function removes a node from the graph and uses the Ford-Fulkerson's algorithm to find a min-cut; it is $O(Fw)$, where w is the maximum weight of edges in the *DFG* [28]. The *source_min_cut* function is $O(1)$ since it only computes the total weight of edges originating from the source node. The *find_stage* function searches in the current decomposition for a stage that contains a node; it is $O(V)$. The *cut_graph* function (Algorithm 3) is $O(V+F)$, which performs a depth-first search to find disconnected components in the graph [28]. The *copy_sd* function is $O(V)$ (replace a stage with two substages). The *modularity* function is $O(V+F)$, which involves copying the original graph to a new one with a special treatment for cut-points ($O(V+F)$) and computing the modularity based on Equation 1 ($O(F)$). The *get_activity_labels* function is $O(V)$ (extract activity labels from nodes). The *sort* function (Algorithm 2) is $O(V \log V)$. Based on these observations, the complexity of Algorithm 1 is $O(V^2(V+F))$, and Algorithm 2 is $O(V(V+F))$.

Given the example log shown in Table 1, Algorithm 1 produces the stage decomposition visualized in Figure 4. It has two stages: $\{i, A, B, C, D, U\}$ and $\{U, E, F, G, H, K, o\}$. The milestone node (or cut-point) U is visualized as a node between two boxes. An edge on this visualization is aggregated from the edges on the *DFG* and the edge weight is the total of the aggregated edge weights on the *DFG*s.

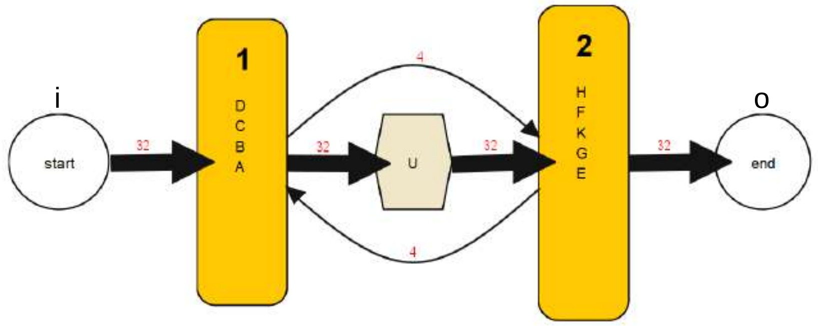


Figure 4: Example stage decomposition

4. Stage-based Process Discovery

This section presents the proposed divide-and-conquer technique for automated discovery of process models from event logs. The section starts by presenting the quality criteria that we adopt to evaluate a process model discovered from an event log. Next, we present the overall process discovery technique, followed by each of its main steps.

4.1. Quality of automatically discovered process models

To evaluate the quality of the resulting process models, we use three measures of accuracy, namely fitness, precision, and F-score, as defined in previous research in the field of automated process discovery. *Fitness* is the ability of a model to reproduce the behavior contained in the log. A fitness of 1 means that the model can reproduce every trace in the log. In this paper, we use the fitness measure proposed in [29], which measures the degree to which every trace in the log can be aligned with a trace produced by the model. *Precision* is the ability of a model to only generate the behavior found in the log. A score of 1 indicates that any trace produced by the model is contained in the log. We use the alignment-based ETC precision measure defined in [30], which is based on similar principles as the above fitness measure. We note that this measure of precision is not universally accepted. A recent study has found that neither this nor other existing measures of precision satisfy a set of basic axioms that a precision measure should intuitively fulfill [31]. In particular, ETC precision is non-deterministic. While new precision measures have been proposed recently [32] fulfilling these axioms, there is currently no consensus yet as to which measure of precision is the most suitable. In the absence of consensus, we adopt the alignment-based ETC precision measure since it is widely used for comparing automated process discovery techniques [33].

Fitness and precision can be combined into a single measure of accuracy, known as *F-score*, which is the harmonic mean of the two measurements, as shown below.

$$\text{F-score} = 2 \cdot \frac{\mathit{Fitness} \cdot \mathit{Precision}}{\mathit{Fitness} + \mathit{Precision}} \tag{2}$$

In addition to having a high F-score, it is natural to expect that a process model discovered from an event log is *syntactically and semantically correct.* A basic syntactic correctness criterion for process models is that all the nodes are on a path from a start node to an end node. In other words, there are no disconnected nodes or dangling arcs. A well-accepted semantic correctness notion is *soundness* [34], which has been defined in the context of Workflow nets[2] and is also applicable to BPMN process models. Specifically, a BPMN process model with one start and one end event is sound iff: (i) given any sequence of task firings starting from the state where there is a token in the start event, the resulting state is either the final state (i.e. the state where there is a token in the end event) or it is possible to reach the final state from that state; (ii) when a token reaches the end event, no other token remains elsewhere; and (iii) it is not possible to reach a state where there are two tokens on the same sequence flow. The first property is called *option to complete*, the second *proper completion*, and the third *safeness*.[3]

4.2. Overview of the Stage-Based Process Discovery Technique

The input of the proposed technique is an event log and a decomposition of the activities in the process into sequential stages as described above. The output is a (BPMN) process model. The technique is composed of four main steps: *Mine, Chain, Adjust*, and *Stitch*, or *MCAS* for short. The *Mine* step decomposes the original event log into one sublog per stage and then mines a submodel for each stage using a base miner (called a *subminer*). The *Chain* step connects the resulting submodels together by merging the milestone nodes to form a *chained model.* Since the subminer is applied on each sublog separately, it cannot exploit information about the overall behavior observed in the full event log, and hence the chained model may have low accuracy. To improve accuracy, the *Adjust* step analyzes the chained model to detect sources of additional behavior, then filters out these sources of additional behavior from the sublogs, and re-applies the Mine and Chain steps again to produce a chained model with higher overall accuracy. This step is orthogonal to the stage-based decomposition approach and can be treated as optional, but it is nonetheless useful if we seek to maximize the accuracy of the process model. Finally, the *Stitch* step adds inter-stage edges to the chained model to complete the overall process model.

[2]A Workflow net is a Petri net with a single source (start) place, a single sink (end) place, and such that every transition is on a path from the start place to the end place.

[3]BPMN allows process models to have multiple start and multiple end events, but such process models can be re-written as process models with a single start and a single end event, hence we can restrict ourselves to process models with a single start and a single end event without loss of generality. Also, the models produced by the automated process discovery techniques used in this article have a single start and a single end event.

The approach is described in detail in Algorithm 4. Lines 1-8 correspond to Mine, Chain, and Adjust steps. The *DiagnoseImprecision* function identifies a set of imprecise behaviors on the chained model as part of the Adjust step. The adjustment can be made optional by replacing Line 5 with *imprecision* $:= \emptyset$. Lines 9-18 correspond to the Stitch step, which takes an exhaustive approach to add a collection of inter-stage edges to the chained model such that the new model can maximize the F-score. It uses a frequency threshold to select all inter-stage edges whose frequency exceeding the threshold. The selection and stitch is repeated for each frequency threshold ranging from 0.0 to 1.0 with incremental step of 0.05.

Algorithm 4: Stage-Based Process Discovery

```
Input: $EL$: An Event Log
       $SD$: A Stage Decomposition
Output: A Process Model
1  $subLogs := ExtractSubLogs(EL, SD)$
2  do
3      $subModels := MineSubModels(subLogs)$
4      $chainModel := Chain(subModels)$
5      $imprecision := DiagnoseImprecision(chainModel, EL)$
6      if $imprecision \neq \emptyset$ then
7          $subLogs := Filter(subLogs, imprecision)$
8  while $imprecision \neq \emptyset$
9  $bestModel := chainModel$
10 $bestFscore := ComputeFscore(bestModel, EL)$
11 $\mathcal{E} := GetInterStageEdges(SD, EL)$
12 for $freqThres \leftarrow 0.0$ to $1.0$ by $0.05$ do
13     $\mathcal{E} := GetInterStageEdges(SD, EL, freqThres)$
14     $stitchModel := Stitch(\mathcal{E}, bestModel)$
15     $newFscore := ComputeFscore(stitchModel, EL)$
16     if $newFscore > bestFscore$ then
17         $bestModel := stitchModel$
18         $bestFscore := newFscore$
19 return $bestModel$
```

The next subsections describe the details of the Mine, Chain, Adjust and Stitch steps.

4.3. Mine

Recall that Algorithms 1 and 2 return a stage decomposition as shown in Figure 2. For example, the stage decomposition discovered from the event log shown in Table 1 is shown in Figure 4 with two stages $\{i, A, B, C, D, U\}$ and $\{U, E, F, G, H, K, o\}$, where U is the milestone node.

In order to create sublogs, we decompose each trace of the log into subtraces – multiple subtraces corresponding to a stage can be extracted from each trace. A subtrace corresponding to a given stage contains all directly-following events in the original trace that belong to that stage without being interrupted by any events of other stages. The subtrace corresponding to a stage is called a *stage-trace* and the sublog containing all stage-traces of a given stage is called as a *stage-log*.

Definition 4 (Stage-trace). *A stage-trace ST^c_S extracted from case $c \in C$ for stage S is a maximal sequence of events in c such that all activities in that sequence belong to S. Formally, ST^c_S is a triple (c, i, j), where $1 \leq i \leq j \leq |c|$ such that $\forall i \leq v \leq j \; [act(c_v) \in S] \wedge (i = 1 \vee act(c_{i-1}) \notin S) \wedge (j = |c| \vee act(c_{j+1}) \notin S)$.*

Definition 5 (Stage-log). *A stage-log SL^{EL}_S extracted from event log EL for stage S is the set of all stage-traces of S extracted from all cases in EL for stage S, i.e. $SL^{EL}_S = \{ST^c_S \mid c \in C\}$.*

A small technical issue arises from the fact that a milestone node between two stages belongs to both stages. However, in the resulting process model, we only need to capture this node once. To resolve this technical issue, we delete each milestone node from the second of the stages in which it appears. In other words, we only retain the copy of a milestone node in the first of the two stages in which it appears.

Consider for example the event log shown in Table 1 and the stage decomposition shown in Figure 4 with two stages $\{i, A, B, C, D, U\}$ and $\{U, E, F, G, H, K, o\}$. We first delete the milestone node U from the second stage, hence the stages become $\{i, A, B, C, D, U\}$ and $\{E, F, G, H, K, o\}$. We then apply the above definitions to decompose the log into stage-logs. Tables 2 and 3 are the two stage-logs. Note that the start event (i) is placed in the subtrace of the first stage-trace of each trace, the end event (o) is placed in the last stage-trace of each trace, and the two stages are non-overlapping.

A subminer is called for the sublogs 1 and 2, producing the BPMN models shown in Figures 5 and 6.

Variant	Trace	Occurrences
1	iABCU	10
2	iADU	18
3	iABU	5
4	iACDU	1
5	iABDU	1

Table 2: Stage-log 1.

Variant	Trace	Occurrences
1	EFo	14
2	Go	17
3	Ho	2
4	HEo	1
5	GFo	1
6	Ko	1

Table 3: Stage-log 2.

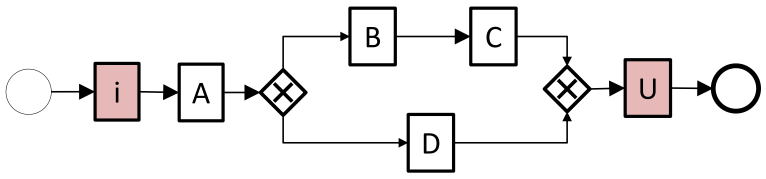


Figure 5: Submodel 1 mined from the sublog 1.

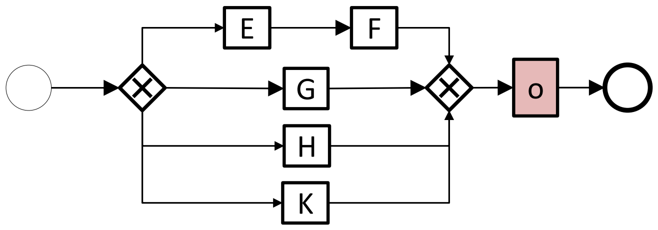


Figure 6: Submodel 2 mined from the sublog 2.

4.4. Chain

The goal of the chain step is to sequentially combine the submodels discovered for each stage into a single model. Without loss of generality, we assume that the submodels discovered by a subminer have a start event with a single outgoing sequence flow (called *starting sequence flow*) and an event with a single incoming sequence flow (called *ending sequence flow*). If this property does not hold (e.g. the start event has multiple outgoing flows), it is always possible to re-factor the model by adding a gateway at the start in order to achieve this property. With this property in place, the chaining step consists in concatenating every two adjacent submodels by merging the ending sequence flow of one submodel with the starting sequence flow of the successive submodels and removing the end event and start event of the two corresponding submodels, as illustrated in Figure 7. In other words, the end sequence flow of the first stage is re-directed so that it points to the target of the starting sequence flow in the second stage.

Coming back to the running example, Figure 8 illustrates the model obtained by chaining the submodel 1 in Figure 5 and the submodel 2 in Figure 6.

We note that the chaining step preserves soundness. In other words, if two models are sound, the resulting chained model will be sound too. Indeed, if the first model is sound, it means that the milestone task can always be reached (and no token is left elsewhere in the first model). And since the second model is sound as well, then starting from a token in the flow coming out from the milestone task, it is possible to reach the end event of the second model, which is the end event of the chained model, without any token being left behind.

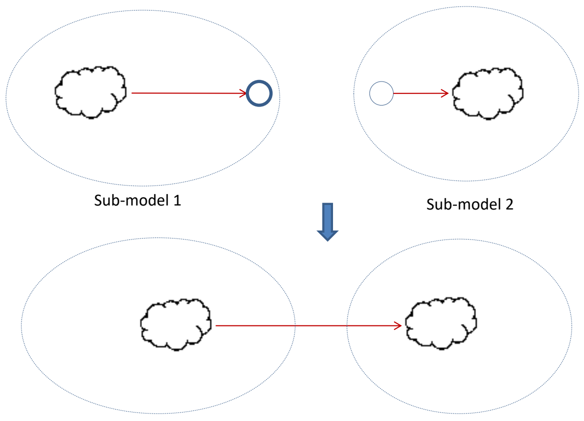


Figure 7: Chaining submodels.

4.5. Adjust

Algorithm 5 implements the diagnosis of imprecise relations in the *adjust* step. Given that a process model is a collection of directed relations (or edges) between activity nodes, the quality of the model depends on how precise such relations are compared with those existing in the logs. Hence, in order to improve the model, the purpose of the algorithm is to find imprecise relations, i.e. those that appear infrequently in the log but are strongly

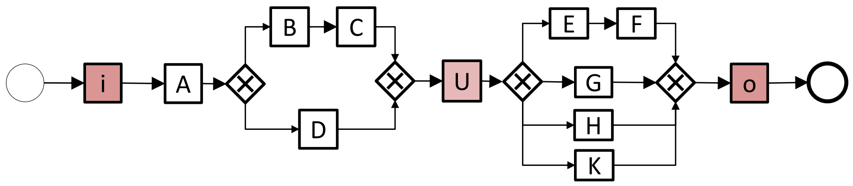


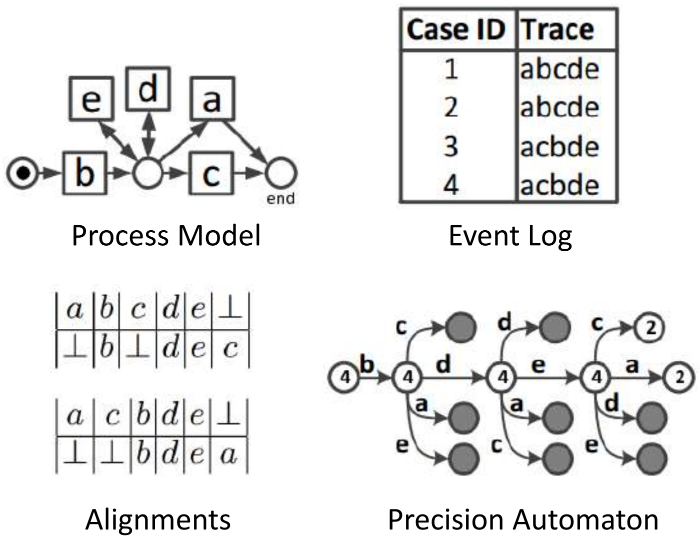
Figure 8: Chained model built from submodel 1 and 2.

represented by the model. It does this by checking details of the precision computation for the model. From [35], this computation is based on an automaton constructed from the alignment between the log and the model. As the alignment and precision are computed on Petri nets, all BPMN models are first converted to Petri nets as an input to this algorithm. If the generated BPMN models only contain AND and XOR gateways, the converted Petri nets are ensured to be equivalent to the BPMN models [36]. We note that we do not support generated models containing OR-join gateways.

Figure 9 shows an example of a precision automaton. The automaton is created from the alignments between the model and the log (the log has two trace variants). "White" nodes represent prefixes in the model-move part (the lower part) of the alignments while a no-move denoted as $\perp$ is ignored. For example, the third "white" node in the figure is associated with prefix *bd* starting from the source node. Edges represent transitions on the model. "Gray" nodes represent model moves not aligned with any traces. Thus, gray nodes are also called *available transitions* of the connected "white" node. Node numbers indicate the total frequency of the corresponding prefixes from the alignments.

From the automaton, we call an *available relation* a pair of transitions where the first transition is the incoming edge of a "white" node and the second transition is any outgoing edge of the same node. They represent available relations allowed by the model. We also call an *actual relation* a pair of transitions where they are incoming and outgoing edges of a "white" node that connects to another "white" node. For example, *ba*,*bc*,*bd*, and *be* are available relations but only *bd* is an actual relation in Figure 9. In addition, we are only interested in *visible relations*, i.e. relations between visible transitions. Thus, we need a way to search for visible relations from a given available or actual relation.

The purpose of Algorithm 5 is to check the automaton for visible relations with significantly higher frequency as available relations than as actual relations. Those are called *imprecise relations*, which need to be removed



Case ID	Trace
1	abcde
2	abcde
3	acbde
4	acbde

$$\begin{array}{|c|c|c|c|c|c|} a & b & c & d & e & \perp \\ \hline \perp & b & \perp & d & e & c \end{array}$$

$$\begin{array}{|c|c|c|c|c|c|} a & c & b & d & e & \perp \\ \hline \perp & \perp & b & d & e & a \end{array}$$

Figure 9: Example automaton (extracted from [35]).

from the model to improve its quality. They exist because of imperfection in the filtering and mining techniques of base miners.

First, the algorithm visits all "white" nodes on the automaton to search for *visible available relations*. If the first transition in the relation is invisible, the algorithm will backtrack from the visited node along the incoming edge until it reaches the first edge associated with a visible transition ($getVisibleEdge$). If the second transition in the relation is invisible, the algorithm will traverse on the model to search for all visible successor transitions ($getVisibleSuccessorTransitions$). As a result, $R_{available}$ is a set of all visible and available relations. Meanwhile, the algorithm also searches for *real actual relations* by traversing forward from the visited node on the automaton until it reaches an edge associated with a visible transition. R_{actual} is a set of all visible and actual relations.

A relation is considered imprecise if it has high frequency (i.e. weight) in $R_{available}$ but low frequency in R_{actual}. To determine the amount of imprecise relations, an imprecision threshold is used, which is the ratio between the frequency of the relation in R_{actual} and its frequency in $R_{available}$. Another type of imprecise relation is one that is allowed by the model (found in $R_{available}$) but that is never found in R_{actual} (Lines 21-23). If so and if they are highly available, i.e. their presence exceeds a threshold, e.g. 80% of the maximum weight in $R_{available}$, they are also considered imprecise relations.

This threshold is called *Minimum Availability*, which is the ratio between the frequency of the relation and the maximum frequency in $R_{available}$. $R_{imprecise}$ is the set of all imprecise relations.

Once all imprecise relations have been collected, entire traces that contain any imprecise relation will be removed instead of the relations (i.e. pair of transitions). This is because removing a pair of transitions from a trace will make new relations appear, thus change log behavior. This makes new models deviate from the original log, which causes lower fitness and precision. As the frequency of imprecise relations is usually low, the expected effect is that the new model is primarily similar to the previous one but with less imprecise relations.

The diagnosis and removal of imprecise relations can be repeated multiple times until no more imprecise relations can be found.

From the chained model shown in Figure 8, Figure 10 shows the alignment of the chained model and the original log in Table 1, and Figure 11 shows the precision automaton of the chained model with three escaping edges highlighted. From the alignment and the precision automaton, the fitness and precision of the current chained model are calculated as *0.92* and *0.83*, respectively (F-score = 0.87).

Based on the precision automaton of the chained model, the Algorithm 5 determines that *UH* and *UK* are two imprecise relations on the model. They are removed from the sublog 2 resulting in the adjusted sublog 2 shown in Table 4. Then, the mining and chaining steps are executed again due to changes in sublogs. Figures 12 and 13 show the adjusted submodel 2 and the chained model.

Figures 14 and 15 show the alignment of the adjusted chained model with the original log and the new precision automaton. The fitness and precision of the adjusted chained model are *0.9* and *1.0*, respectively. It can be seen that the fitness is negligibly reduced because of the removed imprecise relations; however, the precision is significantly improved leading to the improvement of F-score (F-score = 0.95).

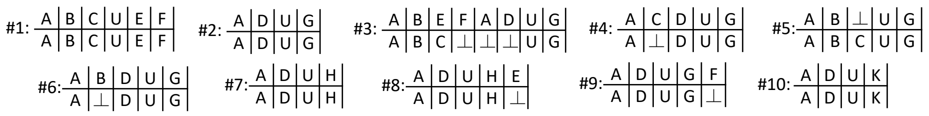

#1: A B C U E F / A B C U E F
#2: A D U G / A D U G
#3: A B E F A D U G / A B C ⊥ ⊥ ⊥ U G
#4: A C D U G / A ⊥ D U G
#5: A B ⊥ U G / A B C U G
#6: A B D U G / A ⊥ D U G
#7: A D U H / A D U H
#8: A D U H E / A D U H ⊥
#9: A D U G F / A D U G ⊥
#10: A D U K / A D U K

Figure 10: Alignment of the chained model in Figure 8 and the log in Table 1.

Algorithm 5: Diagnose Imprecise Relations

```
Input: $M$: Petri net converted from BPMN Model
       $L$: Event Log
       $ImpThres$: Imprecision Threshold
       $MinAvail$: Availability Threshold
Output: A set of imprecise relations
1  $Automaton := GetPrecisionAutomaton(M, L)$
2  $R_{available} := \{\}$
3  $R_{actual} := \{\}$
4  foreach “White” node $a$ in $Automaton$ do
5      if $a$ is not root node then
6          $s_a := a.incomingEdge$
7          if $s_a$ is invisible then
8              $s_a := getVisibleEdge(a)$
9          if $s_a \neq null$ then
10             foreach Transition $o_a \in a.outgoingEdges$ do
11                 $T_{target} = \{\}$
12                 if $o_a$ is visible then
13                     $T_{target} := T_{target} \cup \{o_a\}$
14                 else
15                     $T_{target} := T_{target} \cup getVisibleSuccessorTransitions(o_a)$
16             foreach Transition $t_a \in T_{target}$ do
17                 $r := (s_a, t_a)$
18                 if $r \notin R_{available}$ then
19                     $r.AvailableWeight := a.Weight$
20                     $R_{available} := R_{available} \cup \{r\}$
21                 else
22                     $r.AvailableWeight := r.AvailableWeight + a.Weight$
23             foreach Transition $o_a \in a.outgoingEdges$ do
24                 if $o_a$ is visible then
25                     $r := (s_a, o_a)$
26                     if $r \notin R_{actual}$ then
27                         $r.ActualWeight := o_a.Target.Weight$
28                         $R_{actual} := R_{actual} \cup \{r\}$
29                     else
30                         $r.ActualWeight := r.ActualWeight + e.Target.Weight$
31 $R_{imprecise} := \{\}$
32 foreach Relation $r \in R_{actual}$ do
33     if $r.ActualWeight/r.AvailableWeight \leq ImpThres$ then
34         $R_{imprecise} := R_{imprecise} \cup \{r\}$
35 foreach Relation $r \in R_{available} \setminus R_{actual}$ do
36     if $r.AvailableWeight >= MinAvail * MaxWeight(R_{available})$ then
37         $R_{imprecise} := R_{imprecise} \cup \{r\}$
38 return $R_{imprecise}$
```

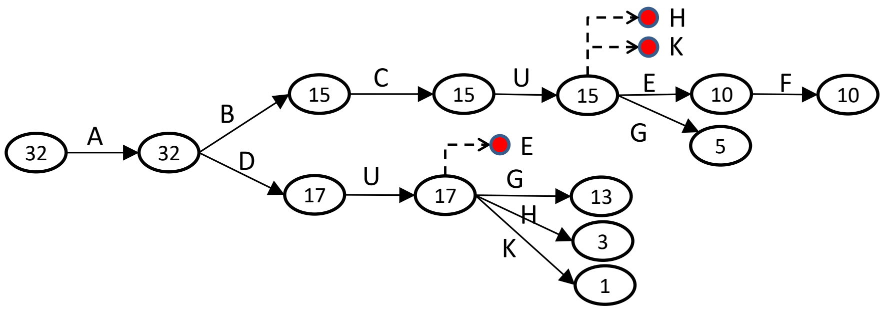


Figure 11: Precision automaton of the chained model in Figure 8.

Variant	Trace	Occurrences
1	EFo	14
2	Go	17
4	GFo	1

Table 4: Adjusted sublog 2.

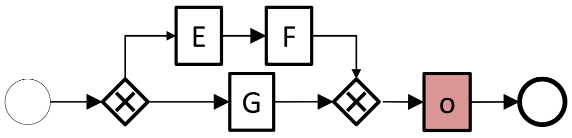


Figure 12: Adjusted submodel 2.

We note that the Adjust step leads to a sound model, assuming that the base miner produces a sound model. Indeed, the miner filters the sublog for each stage, applies the base miner to obtain one submodel per stage, and then it chains the resulting submodels. And as discussed above, a chained model is sound if the submodels are sound.

4.6. Stitch

During the mining step, any edges (i.e. directly-follows dependencies) in the *DFG* that go from the middle of one stage to the middle of another stage are left aside. Instead, one isolated model is discovered for each stage. The chaining step reconnects the stages via edges from the end of one stage to the start of the next stage, but it does not reinstate the other edges lost during the mining step. The stitching step aims to add many of these inter-stage edges into the chained model.

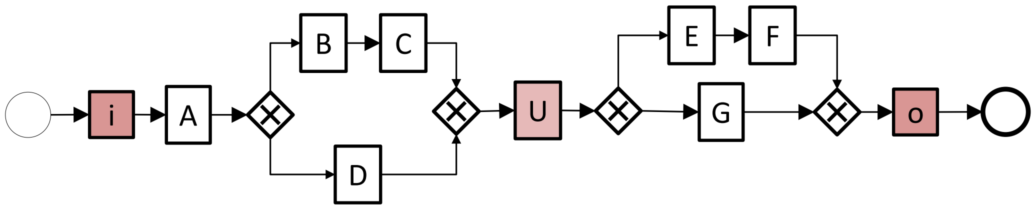
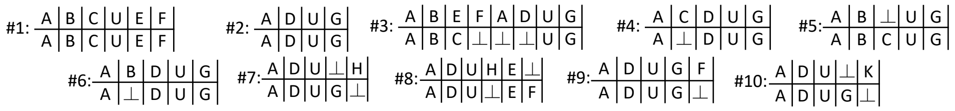


Figure 13: Adjusted chained model.



#1:

A	B	C	U	E	F
A	B	C	U	E	F

#2:

A	D	U	G
A	D	U	G

#3:

A	B	E	F	A	D	U	G
A	B	C	⊥	⊥	⊥	U	G

#4:

A	C	D	U	G
A	⊥	D	U	G

#5:

A	B	⊥	U	G
A	B	C	U	G

#6:

A	B	D	U	G
A	⊥	D	U	G

#7:

A	D	U	⊥	H
A	D	U	G	⊥

#8:

A	D	U	H	E	⊥
A	D	U	⊥	E	F

#9:

A	D	U	G	F
A	D	U	G	⊥

#10:

A	D	U	⊥	K
A	D	U	G	⊥

Figure 14: Alignment of the adjusted chained model and the original log (Table 1).

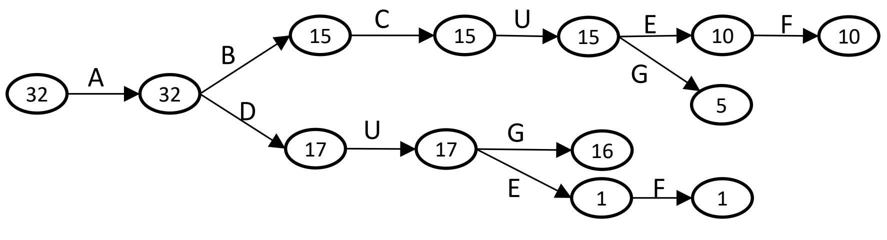


Figure 15: Precision automaton of the adjusted chained model.

We recall that the stage decomposition leads to a linearly ordered set of stages. Let SD be a stage decomposition and S_1, S_2 be two stages in SD, i.e. $S_1, S_2 \in SD$, we denote $S_1 \ll S_2$ *iif* S_1 is followed by S_2 in SD, and $S_1 < S_2$ *iif* S_1 is directly followed by S_2 in SD. An inter-stage edge is classified as a *forward edge* when it goes from one stage to a later stage and it is classified as a *backward edge* otherwise, as illustrated in Figure 16. These two types of edges are formalized as follows.

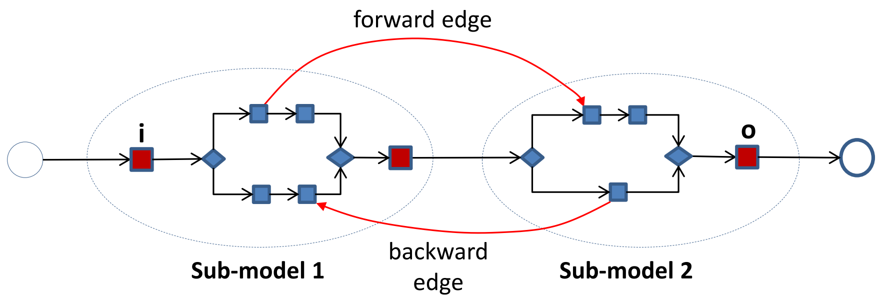


Figure 16: Overview of Stitching.

Definition 6 (Forward Edges). *The set of forward edges on DFG with respect to SD, denoted as* $\mathcal{E}_{SD}^{JF}$*, are directed edges whose source is a node of a stage in SD (other than the milestone node) and whose target is a node of the following stages, i.e.* $\mathcal{E}_{SD}^{JF} = \{(a_1, a_2) \in DFG \mid \exists S_1, S_2 \in SD\ [(S_1 < S_2 \wedge a_1 \in S_1 \wedge a_1 \neq \mathcal{M}_{S_1} \wedge a_2 \in S_2) \vee (S_1 \ll S_2 \wedge S_1 \not< S_2 \wedge a_1 \in S_1 \wedge a_2 \in S_2)]\}$*, where* $\mathcal{M}_{S_1}$ *is the milestone node of* S_1*.*

Definition 7 (Backward Edges). *The set of backward edges on DFG with respect to SD, denoted as* $\mathcal{E}_{SD}^{JB}$*, are directed edges whose source is a node of a stage in SD and target is a node of the preceding stages, i.e.* $\mathcal{E}_{SD}^{JB} = \{(a_1, a_2) \in DFG \mid \exists S_1, S_2 \in SD\ [S_2 \ll S_1 \wedge a_1 \in S_1 \wedge a_2 \in S_2]\}$*.*

When stitching two task nodes belonging to different stages in the chained model, the connection between them is always done via XOR gateways as shown in Figure 17. In other words, when drawing a sequence flow from a task A in one stage to a task B in another stage, we insert an XOR-split gateway right after task A, an XOR-join right before task B, and we draw a jumping edge from the XOR-split to the XOR-join gateway. The addition of jumping edges can make the resulting model unsound if the target of a jumping edge is located in a parallel branch, i.e. a branch that starts from an AND-split gateway. The reason is that when we inject a token in one of multiple parallel branches and not in the other parallel branches, the AND-join gateway(s) where these parallel branches converge will receive one token from one of its incoming flows, but not from the others. Hence, there will be a deadlock in the downstream AND-join gateways, which affects the *option to complete* property discussed in Section 4.1. To avoid such deadlocks, we replace every AND-gateways of the target stage, which are reachable from the point where a jumping edge ends, with inclusive OR-join gateways. An inclusive OR-join gateway will fire if it receives a token in only one of its incoming branches and no token can arrive from the other branches.

While the use of OR-join gateways allows us to guarantee the option-to-complete property, it does not allow us to ensure proper completion, i.e. it does not ensure that when a token reaches the end event, no token is left that can reach the end event later. The stitching procedure is such that improper completion may arise when an edge is added from a parallel branch of a stage to a different stage. These edges create a situation where tokens are present in two different stages simultaneously. These tokens might not be synchronized (e.g. when the edge is a backward edge), leading to improper completion. This same issue arises in the Split Miner [37] – an automated process discovery technique that relies on the use of OR-joins to ensure option-to-complete but does not ensure proper completion. Experimental evaluations conducted in the context of the Split Miner have shown that this problem does not arise when discovering process models from directly-follows graphs extracted from real-life event logs. This observation was confirmed in our experiments, where all models generated using our technique exhibited proper completion and hence soundness.

We also note that existing techniques for measuring the fitness and precision of automatically discovered process models are not able to handle BPMN process models with OR-joins. The reason is that they require a BPMN process model that can be readily converted into a Petri net and OR-join gateways do not have a direct mapping to Petri nets. Hence, to calculate fitness and precision, we need to convert a stitched process model with OR-joins into a model without OR-joins. To replace the introduces OR-joins with XOR and AND-join gateways, we use the technique to remove OR-join gateways proposed in [37], which itself is based on the technique in [38] (the latter being restricted to acyclic process models). The technique in [37] is not able to remove OR-join gateways in all cases, but it does so in practical cases. In the experimental evaluation reported later in this paper, we did not encounter a situation where the OR-join could not be removed by this technique and hence this was not an impediment for measuring fitness and precision.

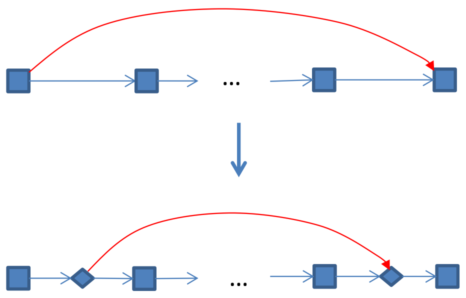


Figure 17: Stitching step.

Coming back to the running example, we note that the *DFG* in Figure 1 has two inter-stage edges: $B \rightarrow E$ and $F \rightarrow A$. Figure 18 shows the stitched model created from the chained model in Figure 8 and the *DFG* in Figure 1.

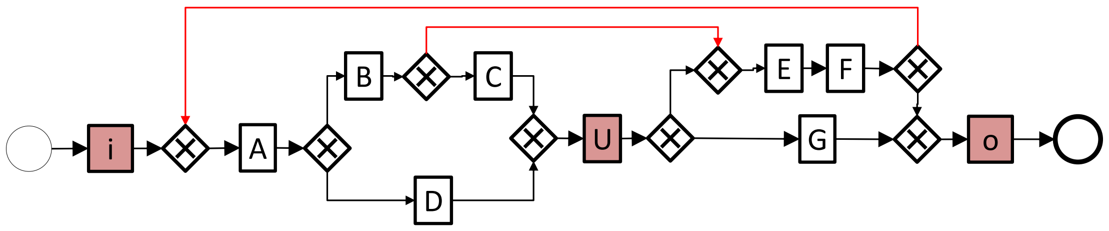


Figure 18: Stitched model created from the chained model in Figure 13 and the *DFG* in Figure 1.

5. Evaluation

We implemented the Staged Process Miner as a set of open-source plugins for the Apromore[4] and ProM[5] The input to the technique is an event log and the output is a process model. A stage decomposition is produced as an intermediate artifact.

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

[5]`http://apromore.org/platform/tools` platforms for process mining. The source code is released in GitHub[6] as well as on the ProM source repository.[7]

Using this implementation, we carried out a two-pronged evaluation on a range of real-life logs and against various log decomposition and automated discovery baselines. Our first evaluation aimed at determining if it is possible to algorithmically produce stage decompositions of event logs that mimic decompositions produced manually by domain experts (our ground truth). This aim is summarized by the following two questions:

Q1. How does the quality of the stage decomposition produced by our proposed method compare to that of existing event log decomposition baselines?

Q2. How does the quality of the decomposition produced by our proposed method vary depending on the minimum stage size?

Our second evaluation aimed at determining the quality of the final process model produced by our technique, starting from a given stage decomposition. This aim is summarized by the following question:

Q3. How does the quality of process models produced by our technique compare to that of automated discovery baselines?

5.1. Datasets

We used eight real-life event logs taken from the 4TU Centre for Research Data.[8] Specifically, we focused on the Business Process Intelligence (BPI) Challenge 2012, 2013, 2015 and 2017 logs. These logs record executions of business processes in various domains such as finance, IT service management and government. We derived the ground truth for the stages in two ways. For the BPI12 and BPI15 logs, the log itself comes with event attributes indicating the stage the event belongs to. For the BPI13 and BPI17 logs, we derived information about the stages from the supporting documentation that accompanies the logs.

BPI12 [7] is a loan origination process in a Dutch bank. Its stages are: i) pre-assess application completeness, ii) assess eligibility, iii) offer & negotiate loan packages with customers, and iv) validate & approve. These stages are marked in the original log by milestone events occurring at the end of each stage, such as A_PREACCEPTED (stage i) and A_ACCEPTED (stage ii),

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

where “A” stands for Application. In this log, we observed that some milestone events within the same case have the same timestamp. This is possibly a logging error. Accordingly, we preprocessed this log by replacing any group of milestone events occurring simultaneously within a case, with a single representative milestone event. This pre-processing step was also required since our technique supports a single milestone per stage only. We used the same pre-processed log to evaluate the baseline techniques considered in this paper.

BPI13 [39] is an IT incident handling process at Volvo Belgium. A stage in this process reflects the IT helpdesk level (team) where an IT incident ticket is processed. The IT department has three levels from 1 to 3. The ground truth of stages is the helpdesk level of the resource who initiates an event, as per the accompanying documentation. This log is preprocessed by removing incomplete cases, which are incident tickets that have not been closed and resolution activities are taking place.

BPI15 [40] is a set of five logs from five Dutch municipalities relating to a building permit application process. This process consists of a main process (HOOFD in the action_code field) and multiple subprocesses. The main process has a number of stages, such as: i) application receipt, ii) completeness check of the application, iii) investigation leading to a resolution (e.g. accept, reject, ask for more info), iv) communication of the resolution, v) public review, vi) decision finalization, and vii) objection and complaint filing. The ground truth of stages in this process is encoded in the action_code field, which has a generic format 01_HOOFD_xyy, where x indicates the stage number ranging from 0 to 8 and yy indicates the activity code within the stage. However, in each of these logs, there is only one case with two activities with Stage number 6, and there is only one activity with Stage number 7, which has similar meaning to an activity in Stage 8 for closing the case. We considered these exceptions as noise and preprocessed each BPI15 log by removing the only case containing Stage 6 and changing the stage number in the events of Stage 7 from 7 to 8. This led to seven stages in total, with the stage number taking values from 0 to 5 and 8. Another issue of these logs is that many events in a case have the same timestamp. We cleaned these logs by sorting the events with the same timestamp within a case using their stage number.

BPI17 [41] is from the same loan origination process and organization as the BPI12 log with a number of changes. This process has three stages to process a loan application: i) pre-assess, ii) assess, and iii) validation. We identified this information about the stages from the supporting documentation of this log.

Table 5 reports descriptive statistics on the size of these datasets. It can be seen that the BPI12 and BPI17 logs have the most number of cases and events. However, the BPI15 logs have the most number of distinct activities and also the largest mean number of distinct activities per case.

Dataset	**Business process**	**Cases**	**Events**	**Activities**
BPI12	Loan Origination	13,087	127,290	19
BPI13	IT Incident Handling	7,456	64,975	35
BPI15-1	Building Permit Application	1,198	39,695	143
BPI15-2		830	31,972	132
BPI15-3		1,402	45,258	132
BPI15-4		1,052	34,523	118
BPI15-5		1,155	43,019	128
BPI17	Loan Origination	31,509	475,306	24

Table 5: Statistics on the datasets used in the evaluation

5.2. Baselines

The goal of the experimental evaluation is to assess the quality of the decompositions produced by the proposed stage decomposition method, and the quality of the process models discovered by the proposed Staged Process Miner (SPM).

To assess the stage decomposition method, we compared it against two state-of-the-art event log decomposition methods, namely *Divide and Conquer* (DC) and *Performance Analysis with Simple Precedence Diagram* (SPD), both presented in Section 2 and available as ProM plugins. Similar to our method, both DC and SPD start from the directly-follows graph constructed from an event log. DC forms a cluster starting from heavy edges (edges with high weights) and grows the cluster to adjacent edges whose weight exceeds a threshold, until the number of nodes in the cluster exceeds a maximum size threshold. DC can produce overlapping clusters, i.e. clusters with some common nodes, because the next found cluster always include both nodes of unvisited edges while some of these nodes belonging to visited edges have been included in the clusters found earlier. SPD searches for clusters based on medoids, i.e. a central node in a directly-follows graph that is close to all other nodes in the cluster, where closeness is measured by the frequency of the directly-follows relation between nodes. SPD can also produce overlapping clusters because the membership function used to assess whether a node belongs to a cluster is a fuzzy function based on a proximity measure of the node to the medoid of the cluster.

Since ground-truth stages are non-overlapping clusters, we adjusted the baseline methods to produce non-overlapping clusters in order to compare their result with the ground truth. For DC, we adjusted the technique so that a node is not added to a cluster if it already belongs to one of the clusters built earlier. This makes the assignment of nodes to clusters arbitrary (it depends on the order in which the nodes are given in the input), but deterministic. For SPD, we adjusted the method to use a crisp membership function that assigns a node to a cluster if the closeness between the node and the corresponding medoid is the highest.

To assess the quality of SPM, we compared this technique with both flat and divide-and-conquer techniques as outlined in Figure 19. As a representative of flat discovery techniques, we selected *Fodina* (FO) which is an improved variant of the Heuristics Miner. This technique does not guarantee that the discovered process model is sound, but in practice, it generally produces sound models. Next, we selected Inductive Miner (IM) and Region-Based Miner (RM) as two decomposed approaches with built-in activity clustering and model discovery. IM discovers block-structured process models from event logs. It has the advantage that it guarantees that the discovered model is sound. RM decomposes an event log into so-called language-based regions, then uses a technique to convert transition systems into Petri nets in order to discover a process model for each region. Finally, we selected *Decomposed Miner* (DM) which is a meta technique in the sense that it cannot discover a process model by itself. Instead, it decomposes the event log into sub-logs and relies on another miner (a base miner) to discover a process model for each fragment in the decomposition. The latter is a characteristic shared with SPM, which is also a meta technique. FO, IM and DM are available in ProM, while RM is implemented in the genet tool. We excluded other decomposition methods that rely on different decomposition dimensions than ours such as BPMN Miner [19]. The latter decomposes a BPMN model into sub-processes based on data objects.

We note that DM sometimes produces Petri nets that are not Workflow nets because they have more than one start or end place. It is straightforward to turn such Petri nets into Workflow nets by adding a transition from a single start place to the initial marking (where each start place has a token) and a transition from every final marking to a single end place. We can then check soundness and compute accuracy on the resulting Workflow net.

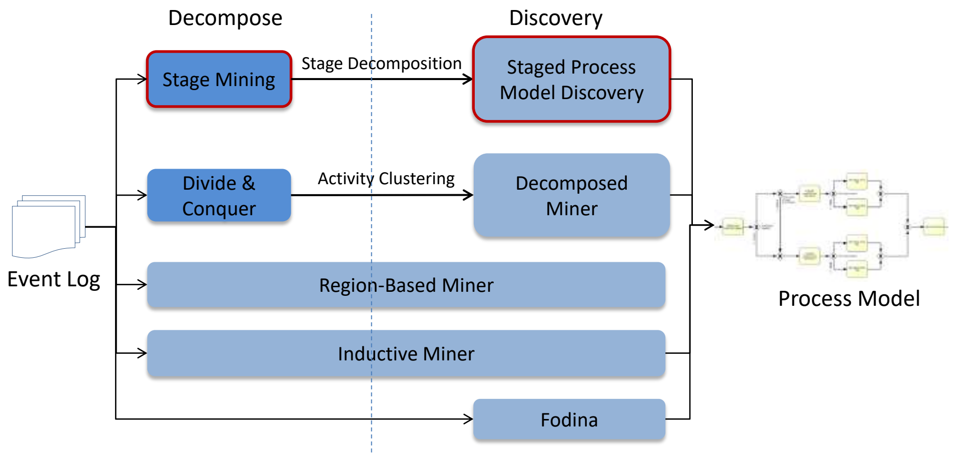


Figure 19: Overview of the techniques used in the automated process discovery evaluation.

5.3. Quality Measures

To evaluate the accuracy of a stage decomposition against the ground truth, we experimented with three well-known external indexes of clustering quality: *Rand*, *Fowlkes–Mallows* and *Jaccard* [42]. These indexes are used to evaluate the similarity of two clusterings. The higher the index is, the more similar the two clusterings are. In our tests, the Rand Index was very high even for less similar clusterings while Jaccard was often low even for very similar clusterings. Fowlkes–Mallows provided more reasonable results between those returned by the other two indexes. Thus, we decided to report the results using the Fowlkes–Mallows index only, given that Rand and Jaccard also showed consistent results across all datasets and techniques. The formula for Fowlkes–Mallows is provided below, where n_{11} is the number of activities that are in the same stage in both decompositions, and $n_{10}(n_{01})$ is the number of activities that are in the same stage in the first (second) decomposition but in different stages in the second (first) decomposition.

$$\text{Fowlkes–Mallows} = \frac{n_{11}}{\sqrt{(n_{11}+n_{10})(n_{11}+n_{01})}} \tag{3}$$

To evaluate the accuracy of the discovered models, we report on fitness, precision, and F-score as defined in Section 4.1. As noted in Section 4.1, ETC-precision is non-deterministic [31]. To mitigate the threat to validity that this non-determinism entails, we computed the ETC precision twice for each log. We found that the observed ETC precision value was the same results in both runs for all logs and miners, except for the BPI12 log when using inductive miner with noise thresholds of 0.3 and below. Accordingly, for this latter log, we computed the ETC precision five times on the BPI12 log and reported the average and the confidence interval.

We hyper-parameter optimized all techniques in order to find the best F-score for each of them. For SPM and SPD, we experimented with the following parameters: minimum stage size and number of clusters, respectively. For DC, we ran the technique with the *Decompose* configuration used in the evaluation of the original paper [10]. The results reported are based on the best parameters configuration for each technique (see Table 6).

Besides fitness, precision and their F-score, we report on model complexity to avoid the bias of the so-called "flower" model (i.e. a model where all tasks can be performed in any order) or the "enumeration" model (a model that enumerates all distinct traces of the log through an XOR-split gateway) [43], which can easily achieve perfect fitness and F-score, respectively. *Complexity* quantifies how difficult it is to understand a model. Several complexity metrics have been shown to be (inversely) related to understandability [44], including *Size* (number of nodes), *Control-Flow Complexity (CFC)* (the amount of branching caused by split gateways in the model), and *Structuredness* (the percentage of nodes located directly inside a well-structured single-entry single-exit fragment).

Finally, we complement the above quality measures with run-time performance. For decomposition, we measure the time from reading a log to the generation of a given decomposition. For automated process discovery, we measure the time from reading a log to the generation of the final process model, including the time for the evaluation of fitness and precision to select models with the best F-score. The latter is needed to compare the various discovery techniques using their best possible model obtained via hyper-parameter optimization. As some generated process models are overly complex and time for model-log alignment can be extremely long, we set a timeout of 200 seconds. All the above measures are summarized in Table 7.

All experiments were conducted on a laptop with Intel i7 2.1GHz 64-bit CPU, 16GB RAM, 8GB RAM for Java 8 Virtual Machine, and Microsoft Windows 10.

5.4. Stage Decomposition Results

We present the evaluation results in light of the first two questions identified above.

Technique	**Description**	**Parameter**	**Min. value**	**Max. value**	**Step**
FO	Fodina	Dependency threshold	0.2	0.8	0.2
		Pattern threshold	0.2	0.8	0.2
IM	Inductive Miner	Noise Threshold	0.1	0.7	0.1
RM	Region-based Miner	-	-	-	-
DM_{im}	Decomposed Miner on top of Inductive Miner	Noise Threshold	0.0	0.7	0.1
DM_{fo}	Decomposed Miner on top of Fodina	Dependency threshold	0.2	0.8	0.2
		Pattern threshold	0.2	0.8	0.2
SPM_{im}	SPM on top of Inductive Miner	Noise threshold	0.0	0.7	0.1
		Interstage threshold	0.0	0.8	0.1
		Imprecision threshold	0.06	0.06	-
SPM_{fo}	SPM on top of Fodina	Dependency threshold	0.2	0.8	0.2
		Pattern threshold	0.2	0.8	0.2
		Interstage threshold	0.0	0.8	0.1
		Imprecision threshold	0.06	0.06	-

Table 6: Parameters, value ranges and step increments used to evaluate each automated process discovery technique

Measure	**Type**	**Description**
Fowlkes–Mallows	Decomposition accuracy	Similarity between a stage decomposition and its ground truth
Fitness	Model accuracy	Ability to reproduce the behavior in the log
Precision	Model accuracy	Ability to only generate the behavior in the log
F-score	Model accuracy	Harmonic mean of fitness and precision
Size	Model complexity	Number of nodes in the model
CFC	Model complexity	Amount of branching caused by split gateways in the model
Structuredness	Model complexity	Percentage of nodes in a well-structured single-entry single-exit fragment
Soundness	Model correctness	Behavioral correctness
Discovery time	Time performance	Time to discover a process model with the highest F-score

Table 7: Quality and time performance measures used to compare different techniques for stage decomposition and automated process discovery

Q1. How does the quality of the stage decomposition produced by our technique compare to that of existing baselines?

Table 8 shows the Fowlkes–Mallows index for the three techniques, for each log. SPM, in either of its two variants (highest modularity and lowest cut-value) consistently outperformed the two baseline techniques across all datasets except SPD for the BPI13 log. These results attest the appropriateness of the modularity measure for stage decomposition, with lowest cut-value being a good approximation of the ground truth. In addition, our heuristics-based techniques with highest modularity and lowest cut-value can approximate the optimal selection of cut-points when comparing with the exhaustive technique for the BPI12 and BPI13 logs. For the BPI15-x logs, the exhaustive technique does not finish after running for several hours due to the large number of combinations of cut-points.

Figures 20, 21 and 22 visualize the three decompositions by SPM, DC and SPD, respectively. Activities highlighted in red are those in wrong clusters. SPM identifies correct clusters for all activities, while DC has many wrongly placed activities and SPD has some wrongly placed activities.

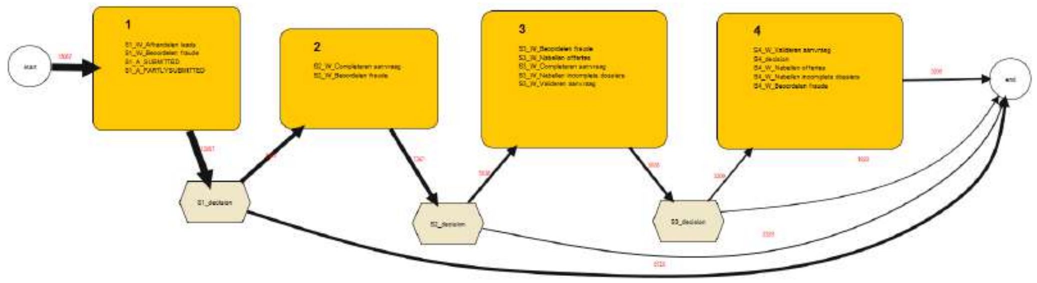


Figure 20: Stage decomposition produced by SPM for the BPI12 log.

For the BPI13 log, SPD achieves the best index among the techniques with one wrongly placed activity, SPM comes the second with some wrongly placed activities and DC has many wrongly placed activities (see Figures 23, 24 and 25). This process has a unique feature: each stage has a key activity that often receives IT tickets from the other stages and is strongly connected with all other activities in a stage (see Figure 28). Based on this feature, SPD can detect these activities as medoids and correctly identifies clusters based on the closeness between the medoids and other nodes. In contrast to the BPI12 log without this feature, SPD fails to identify correct clusters. In case of the BPI13 log, SPM detects these activities as cut-points and produces a

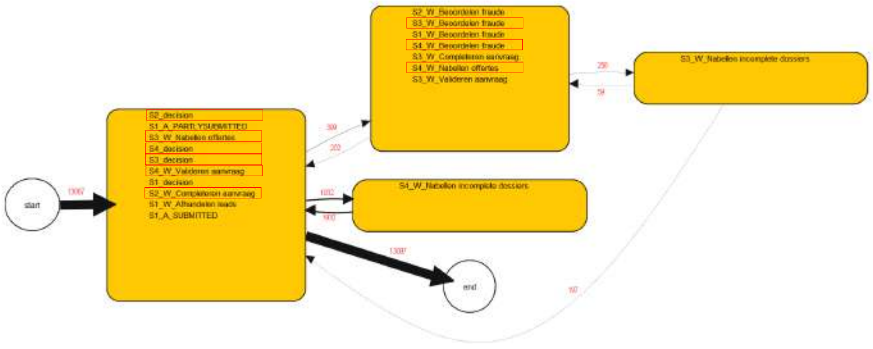


Figure 21: Stage decomposition produced by DC for the BPI12 log.

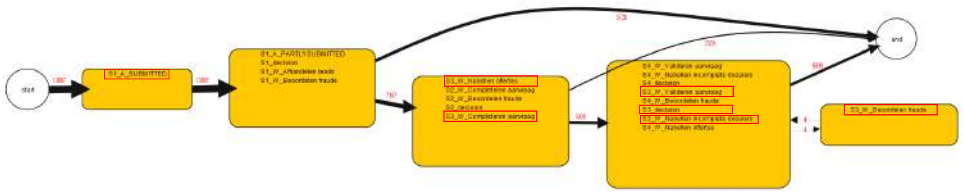


Figure 22: Stage decomposition produced by SPD for the BPI12 log.

close decomposition to the ground truth with a few wrongly placed activities.

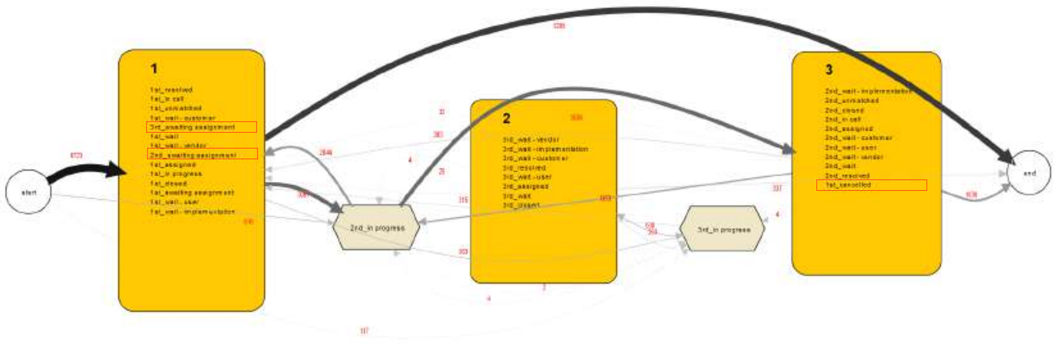


Figure 23: Stage decomposition produced by SPM for the BPI13 log.

For other logs, SPM consistently achieves higher accuracy index than SPD and DC. Figures 26 and 27 visualize the stage decomposition of SM for the BPI15-1 and BPI17 logs. For the BPI15-1 log, although SPM identifies the correct order of stages and closely the number of stages as the ground truth (8 vs. 7), there are a number of wrongly placed activities that affect its accuracy index. As each log in the set of BPI15 logs has a large number

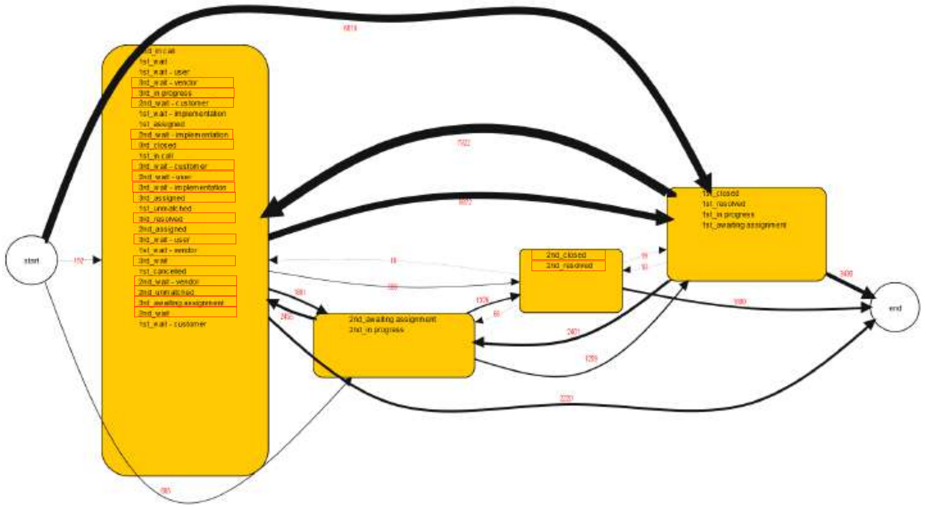


Figure 24: Stage decomposition produced by DC for the BPI13 log.

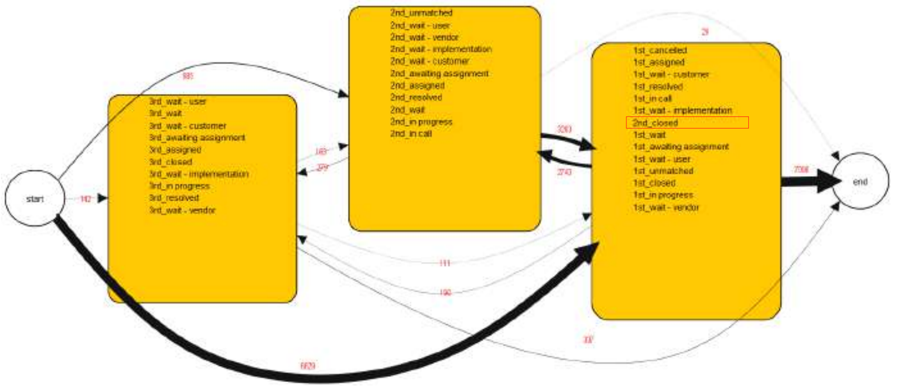


Figure 25: Stage decomposition produced by SPD for the BPI13 log.

of activities, relatively large number of stages and variations of activities in a stage, SPM is the best effort among these techniques that can produce the closest decomposition to the ground truth. For the BPI17 log, SPM identifies the correct order and number of stages with three wrongly placed activities.

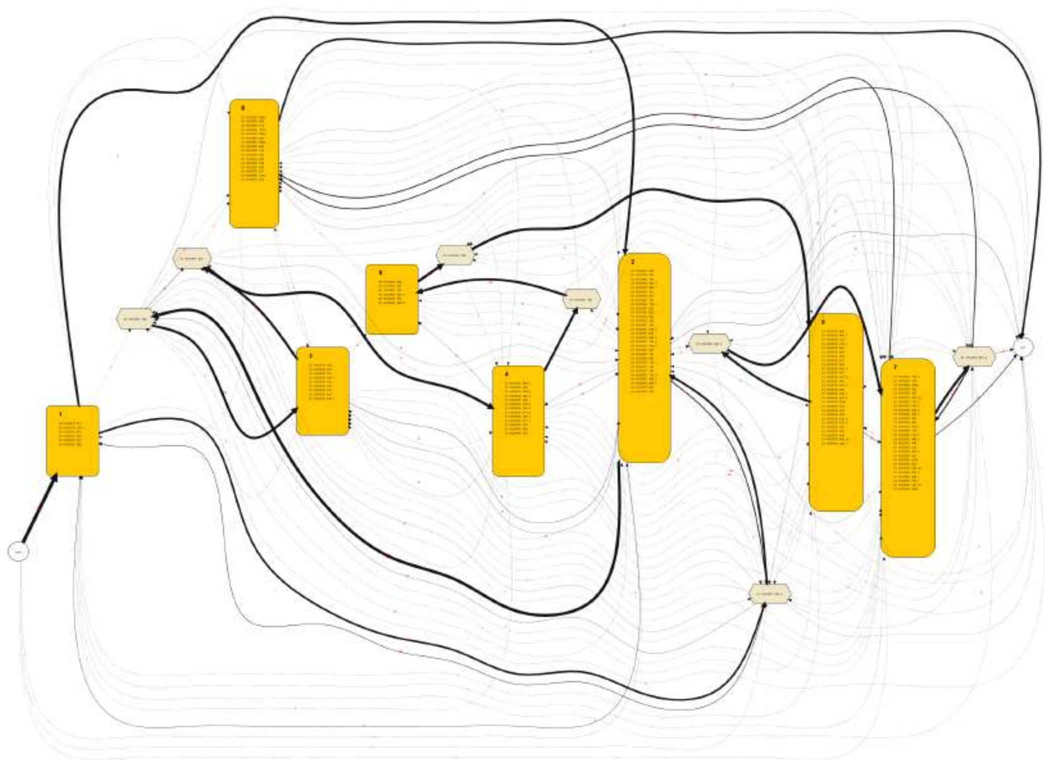


Figure 26: Stage decomposition produced by SM for the BPI15-1 log.

Among these techniques, only SPM can retrieve the order of stages while ordering is not part of the results provided by the two baseline techniques. For these logs, SPM can correctly identify the order of stages from BPI12, BPI15 and BPI17 logs. For the BPI13 log, as shown in Figure 23, Stage 3 is placed between Stage 1 and Stage 2. This is because this process has a strong flow between Stage 1 and Stage 3 without going through Stage 2 (IT incident tickets are sent directly from the Helpdesk line 1 to line 3), making the stage decomposition algorithm wrongly take Stage 3 as the second stage.

In terms of runtime performance, both our technique and the two baselines perform within reasonable bounds, in the order of seconds. However, as mentioned before, the exhaustive variant of our technique could not finish

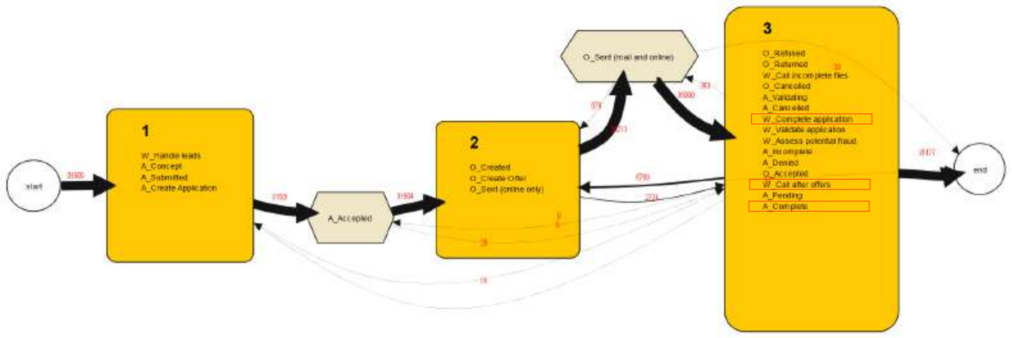


Figure 27: Stage decomposition produced by SM for the BPI17 log.

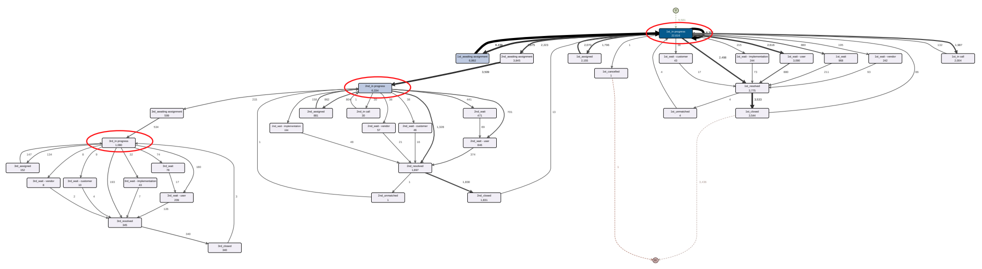


Figure 28: Simplified *DFG* created from the BPI13 log using the Disco tool.

Dataset	DC	SPD	SPM		
			Highest Modularity	**Lowest Cut-value**	**Exhaustive**
BPI12	0.26	0.56 (Clusters=5)	**1.0** (MinSS=3)	**1.0**	**1.0**
BPI13	0.45	**0.94** (Clusters=3)	0.73 (MinSS=4)	0.73	0.73
BPI15-1	0.39	0.42 (Clusters=4)	**0.55** (MinSS=6)	**0.55**	Timed-out
BPI15-2	0.31	0.37 (Clusters=2)	**0.55** (MinSS=7)	**0.55**	Timed-out
BPI15-3	0.36	0.37 (Clusters=9)	**0.54** (MinSS=6)	**0.54**	Timed-out
BPI15-4	0.32	0.37 (Clusters=6)	**0.61** (MinSS=7)	**0.61**	Timed-out
BPI15-5	0.30	0.36 (Clusters=9)	**0.54** (MinSS=7)	**0.54**	Timed-out
BPI17	0.44	0.44 (Clusters=2)	**0.75** (MinSS=4)	**0.75**	**0.75**

Table 8: Fowlkes–Mallows index for the evaluated techniques (MinSS: Min. stage size).

for the BPI15-x logs after running for several hours.

Q2. How does the quality of the decomposition produced by our technique vary depending on the minimum stage size?

To answer this question, we run our technique with the highest modularity algorithm using different values of minimum stage size (minSS), from 2 to half of the total number of activities in an event log. Table 9 provides the characteristics of different stage decompositions, each for a minSS value. It shows that the modularity is usually higher when minSS is small (minSS is 2 or 3). This is because there are often strong edges between 2 or 3 nodes on DFGs representing highly common activity sequences. As minSS gets smaller, the technique will decompose the graph into small stages as long as the modularity keeps increasing.

MinSS	BPI12			BPI13			BPI15-1			BPI17		
	Stages	Mod	FM	Stages	Mod	FM	Stages	Mod	FM	Stages	Mod	FM
2	6	0.53	0.79	4	**0.35**	0.57	11	0.75	0.49	7	0.67	0.56
3	4	**0.59**	**1.00**	3	0.34	0.59	9	**0.76**	0.51	5	**0.68**	0.59
4	4	0.52	0.81	3	0.31	**0.73**	9	0.75	0.49	3	0.56	**0.75**
5	3	0.52	0.82	3	0.31	0.73	9	0.75	0.49	2	0.47	0.70
6	3	0.44	0.70	3	0.31	0.73	8	0.75	**0.55**	2	0.47	0.70
7	2	0.42	0.68	3	0.31	0.73	8	0.75	0.55	2	0.47	0.70
8	2	0.42	0.67	3	0.31	0.73	6	0.69	0.53	2	0.47	0.70
9				2	0.31	0.63	6	0.69	0.53	2	0.47	0.70
10				2	0.31	0.63	6	0.69	0.53	2	0.47	0.73
11				2	0.31	0.63	5	0.69	0.52	2	0.47	0.73
12				1	0.10	0.56	5	0.69	0.52			
13				1	0.10	0.56	5	0.69	0.52			
14				1	0.10	0.56	5	0.69	0.52			
15				1	0.10	0.56	5	0.69	0.52			
16				1	0.10	0.56	5	0.69	0.50			
17							4	0.67	0.50			
18							4	0.67	0.50			
19							4	0.67	0.50			
20							4	0.67	0.50			
21							4	0.67	0.50			
22							4	0.67	0.50			
23							4	0.67	0.50			
24							4	0.67	0.50			
25							3	0.58	0.49			
26												
27												
28												
29												

Table 9: Highest-modularity SPM with different minimum stage sizes (MinSS=Minimum Stage Size, Mod=Modularity, FM=Fowlkes–Mallows).

5.5. Automated Process Discovery Results

In the last part of our evaluation, guided by our third question shown below, we focus on the quality of the models automatically discovered by our technique, and compare the results with various baselines.

Q3. How does the quality of process models produced by our technique compare to that of the baselines?

To answer this question, we compared Staged Process Miner on top of Inductive Miner (called $\mathrm{SPM}_{\mathrm{im}}$) and on top of Fodina ($\mathrm{SPM}_{\mathrm{fo}}$), against various baseline techniques for automated process discovery: two base miners, Inductive Miner (IM), Fodina (FO), and two decomposition miners, Decomposed Miner on top of IM and FO (called $\mathrm{DM}_{\mathrm{im}}$and $\mathrm{DM}_{\mathrm{fo}}$), and Region-based Miner (RM). As discussed, we used the following quality measures: fitness, precision and their F-score as proxies for model accuracy, size, CFC and structuredness (as proxies for model complexity), and soundness. We complemented these quality measures with runtime performance, where we set a time out of five minutes for each technique, which includes the time to compute the F-score.

Table 10 reports the evaluation results. We can observe that SPM always achieves the best F-score across all logs. This occurs when SPM is used on top of IM, except for the BPI17 log, where the best score is achieved with SPM on top of Fodina, though $\mathrm{SPM}_{\mathrm{im}}$ is close off. The $\mathrm{SPM}_{\mathrm{fo}}$ variant also performs quite well in terms of accuracy, often reaching the second-highest F-score. This confirms that our technique indeed enhances the accuracy of the base miner it is used atop. Notably, SPM can achieve better F-score than all other baselines because precision significantly improves while fitness is only marginally affected negatively.

Among the decomposition methods, SPM is the only one that returns sound models. In contrast, DM and RM produce unsound models because of their stitching technique. Although Table 10 shows fitness and precision measurements for the unsound models produced by DM and RM, these measurements are not reliable. In the case of an unsound model, the alignment fitness measure might not reflect the alignment penalty between each trace in the log and the closest trace that the model can generate, but rather the penalty of an alignment between each trace of the log and a trace of the model with the least number of skip transitions.

The complexity of the models obtained by SPM is generally low compared to the models obtained by the two base miners IM and FO, generally higher,

Log name	Discovery technique	Accuracy			Complexity			Sound?	Runtime (seconds)
		Fitness	Precision	F-score	Size	CFC	Struct.		
BPI12	IM	0.98	0.82±0.068	0.93	54	36	0.94	**yes**	**78**
	FO	1.0	0.85	0.92	45	32	0.67	**yes**	594
	SPM$_{\text{im}}$	0.97	0.95	**0.96**	31	20	**1.0**	**yes**	107
	SPM$_{\text{fo}}$	0.77	1.0	0.87	**24**	**10**	0.75	**yes**	99
	DM$_{\text{im}}$	-	-	-	86	61	0.34	-	timed out
	DM$_{\text{fo}}$	-	-	-	89	75	0.42	-	timed out
	RM	1.0*	0.20*	0.33*	22	17	-	no	15
BPI13	IM	1.0	0.58	0.68	63	37	**0.98**	**yes**	831
	FO	1.0	0.61	0.76	85	145	0.15	**yes**	1,105
	SPM$_{\text{im}}$	0.78	0.76	0.77	97	67	0.83	**yes**	**543**
	SPM$_{\text{fo}}$	0.77	0.84	**0.80**	**37**	**35**	0.3	**yes**	460
	DM$_{\text{im}}$	-	-	-	91	55	0.78	-	timed out
	DM$_{\text{fo}}$	-	-	-	111	148	0.74	-	timed out
	RM	-	-	-	-	-	-	-	timed out
BPI15-1	IM	-	-	-	-	-	-	-	timed out
	FO	-	-	-	-	-	-	-	timed out
	SPM$_{\text{im}}$	0.72	0.76	**0.74**	**180**	**103**	**0.78**	**yes**	**818**
	SPM$_{\text{fo}}$	0.81	0.65	0.72	213	161	0.46	**yes**	1057
	DM$_{\text{im}}$	-	-	-	-	-	-	-	timed out
	DM$_{\text{fo}}$	-	-	-	496	540	0.15	-	timed out
	RM	-	-	-	-	-	-	-	timed out
BPI15-2	IM	-	-	-	-	-	-	-	timed out
	FO	-	-	-	-	-	-	-	timed out
	SPM$_{\text{im}}$	0.73	0.64	**0.68**	**214**	**105**	**1.0**	**yes**	**1,715**
	SPM$_{\text{fo}}$	-	-	-	321	525	0.18	**yes**	timed out
	DM$_{\text{im}}$	-	-	-	441	301	0.39	no	timed out
	DM$_{\text{fo}}$	-	-	-	567	712	0.11	-	timed out
	RM	-	-	-	-	-	-	-	timed out
BPI15-3	IM	-	-	-	-	-	-	-	timed out
	FO	-	-	-	-	-	-	-	timed out
	SPM$_{\text{im}}$	0.74	0.78	0.76	**140**	**66**	**1.0**	**yes**	**571**
	SPM$_{\text{fo}}$	0.82	0.65	**0.72**	182	146	0.36	**yes**	960
	DM$_{\text{im}}$	-	-	-	169	120	0.31	no	timed out
	DM$_{\text{fo}}$	-	-	-	460	451	0.18	no	timed out
	RM	-	-	-	-	-	-	-	timed out
BPI15-4	IM	-	-	-	-	-	-	-	timed out
	FO	-	-	-	-	-	-	-	timed out
	SPM$_{\text{im}}$	0.79	0.76	**0.78**	**187**	**105**	**0.68**	no	**417**
	SPM$_{\text{fo}}$	0.85	0.61	0.71	195	163	0.3	**yes**	531
	DM$_{\text{im}}$	-	-	-	-	-	-	-	timed out
	DM$_{\text{fo}}$	-	-	-	445	486	0.14	-	timed out
	RM	-	-	-	-	-	-	-	timed out
BPI15-5	IM	-	-	-	-	-	-	-	timed out
	FO	-	-	-	-	-	-	-	timed out
	SPM$_{\text{im}}$	0.70	0.77	**0.73**	**149**	**76**	**0.86**	**yes**	**955**
	SPM$_{\text{fo}}$	-	-	-	312	396	-	no	timed out
	DM$_{\text{im}}$	-	-	-	-	-	-	-	timed out
	DM$_{\text{fo}}$	-	-	-	483	479	0.12	-	timed out
	RM	-	-	-	-	-	-	-	timed out
BPI17	IM	0.59	0.96	0.73	**21**	**5**	**1.0**	**yes**	610
	FO	1.0	0.57	0.73	64	119	0.11	**yes**	1,513
	SPM$_{\text{im}}$	0.84	0.83	0.84	42	18	0.76	**yes**	718
	SPM$_{\text{fo}}$	0.95	0.81	**0.87**	50	43	0.36	**yes**	789
	DM$_{\text{im}}$	-	-	-	83	54	-	no	timed out
	DM$_{\text{fo}}$	-	-	-	110	116	0.24	-	timed out
	RM	1.0*	0.21*	0.34*	29	21	-	no	**52**

Table 10: Comparison of techniques for automated process discovery (* indicates unreliable value due to the model being unsound).

and by DM_fo, which is always the highest.

Another finding is that SPM achieves equal or better runtime performance than the base miners. This is thanks to the decomposition approach where sub-models are mined and assessed separately, from smaller sub-logs. At the same time, SPM uses a relatively fast decomposition approach. In contrast, RM suffers from scalability issues due to its decomposition approach, to the extent that it times-out in several cases.

To illustrate the differences between the evaluated approaches, let us consider the models obtained by the various techniques assessed above, starting from the BPI12 log. Figures 29 and 30 show the models produced by IM and FO for this log, while Figure 31 shows the model obtained by SPM on top of IM, and Figures 32 and 33 show the models obtained by DM and RM. As this log is highly structured, the models obtained by IM and FO have high F-score but at the same time, high model complexity. However, the model obtained by SPM on top of IM has the highest F-score and is the simplest in terms of Size and CFC measures. Compared to IM, SPM has simplified the model by removing a number of edges, yet the model is able to keep a higher F-score. In contrast, the models obtained by other decomposed techniques are overly complex with low precision and unsound. DM suffers from the way it stitches submodels by using many AND gateways, while RM suffers from its decomposition approach, which often results in a low-precision model.

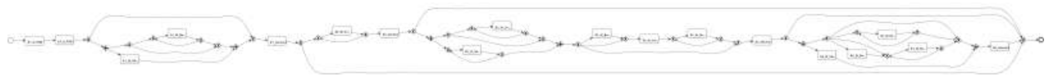


Figure 29: Process model discovered from the BPI12 log by Inductive Miner.

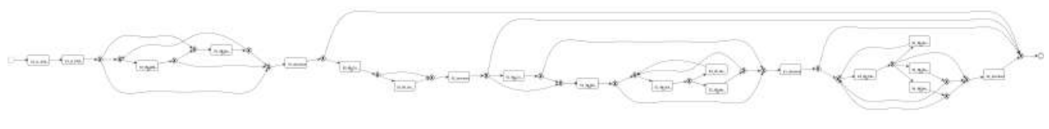


Figure 30: Process model discovered from the BPI12 log by Fodina.

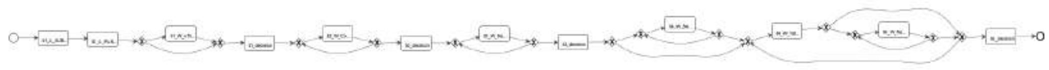


Figure 31: Process model discovered from the BPI12 log by SPM on top of Inductive Miner.

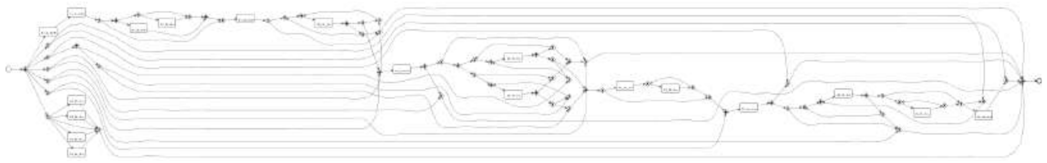


Figure 32: Process model discovered from the BPI12 log by Decomposed Miner on top of Inductive Miner.

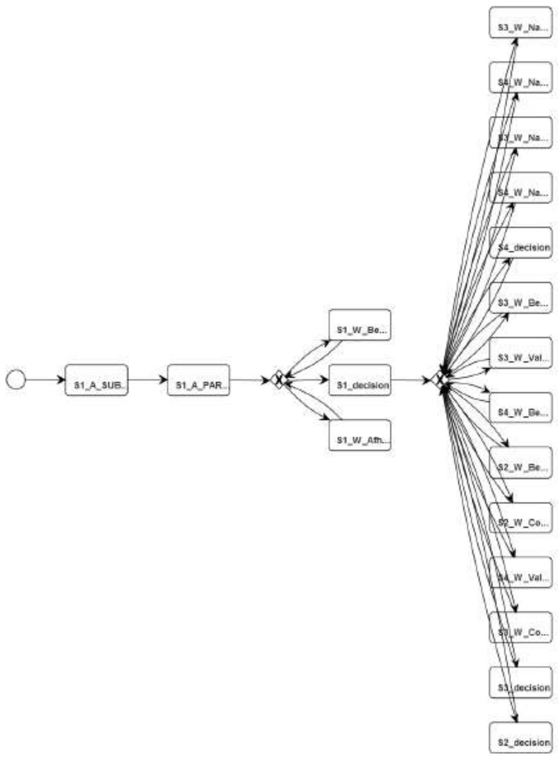


Figure 33: Process model discovered from the BPI12 log by Region-based Miner. Note the absence of an end event.

For the BPI13 log, the log is less structured than the BPI12 log and reveals more differences between the techniques. Figures 34 and 35 show the models discovered by IM and FO, while Figures 36 and 37 show the model obtained by SPM on top of IM and FO, respectively. The model obtained by IM is highly structured (Structuredness = 1.0 by design, however, 0.98 is returned based on the latest Inductive Miner implementation). On the contrary, the model obtained by Fodina is highly unstructured (Structuredness = 0.15) and also highly complex. This is because Fodina keeps all activities in order to achieve high fitness while IM always maintains a structured model while also filtering out infrequent activities to improve the model precision. In contrast, the model obtained by SPM on top of IM is slightly less structured than the one by IM because of the stitching of inter-stage edges. Thus, it improves over IM in terms of precision but results in lower fitness. This is because when SPM uses IM to mine submodels, IM has filtered out significant behaviors in sublogs to achieve the best F-score. When submodels are chained, the fitness of the chaining model is rather low (approximately 0.5). Subsequently, when the submodels are stitched with inter-stage edges, it can only improve the precision and fitness to a certain level given that fitness has been lost in the submodels. On the other hand, the model obtained by SPM on top of Fodina improves over SPM on top of IM. This is because Fodina tends to keep as many activities and edges as possible, which can then be improved by SPM. The model obtained by SPM on top of Fodina is also much simpler than the one by Fodina itself. Compared to IM, it also has much lower Size and CFC values. In a nutshell, SPM depends on the subminer used. It can produce significantly simpler models than the base techniques (e.g. compared with Fodina). It can also achieve higher F-score than the base techniques (e.g. SPM on top of Fodina). In terms of structuredness, despite the low structuredness of the process model from this log, SPM has an advantage that it is always aware of the overall structure of the produced model because it is based on a stage decomposition. For other decomposed techniques, the model obtained by Decomposed Miner is overly complex with many AND gateways as shown in Figure 38. Both the log-model alignment and soundness check timed out. Similarly, the Region-Based Miner could not complete the discovery of the model from this log within the allotted time.

For the BPIC15 logs, the main challenge is due to their complexity: these logs have a large number of activities, long traces and extreme variance between traces. For these logs, both Inductive Miner and Fodina produce highly complex models, which often make the computation of model quality/complexity and soundness verification time out. Figures 39 and 40 illus-

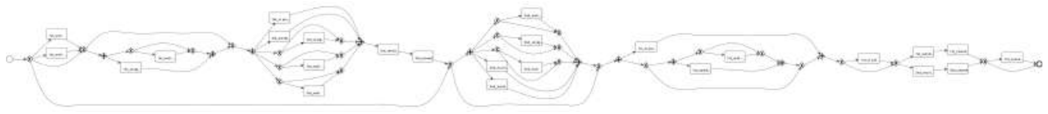


Figure 34: Process model discovered from the BPI13 log by Inductive Miner.

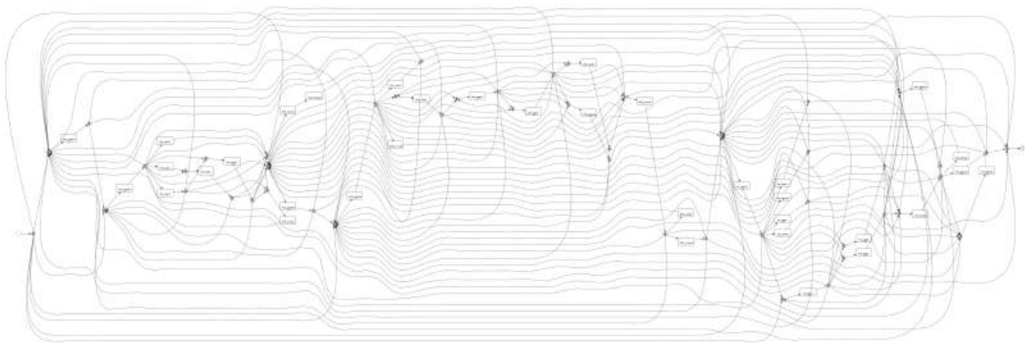


Figure 35: Process model discovered from the BPI13 log by Fodina.

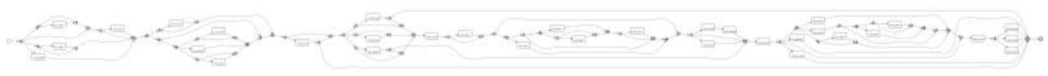


Figure 36: Process model discovered from the BPI13 log by SPM on top of Inductive Miner.

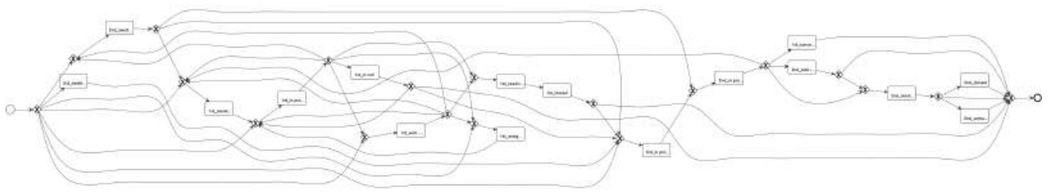


Figure 37: Process model discovered from the BPI13 log by SPM on top of Fodina.

trate the models obtained by Inductive Miner and Fodina from the BPI15-1 log. The former exhibits a form of flower model with edges connecting from an XOR join gateway back to an XOR split gateway. The latter is a typical example of a spaghetti model. In contrast, Figures 41 and 42 show much simpler models discovered by SPM on top of Inductive Miner and Fodina. These results highlight the advantage of SPM over the base techniques in dealing with complex logs. We also observe that for the BPI15-5 log, as Fodina produces an unsound submodel, the chained model produced by SPM by chaining submodels is also unsound. For other decomposed techniques, Decomposed Miner often produces overly complex models, which makes it unable to check the model quality and soundness within an acceptable timeframe, while the Region-Based Miner often fails to produce a model within the same timeframe.

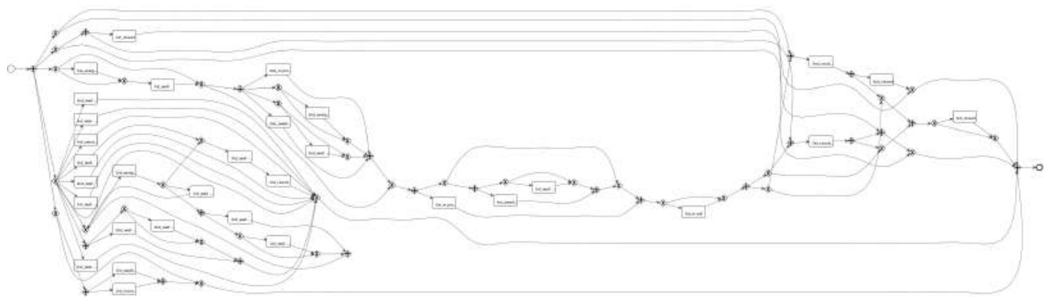


Figure 38: Process model discovered from the BPI13 log by Decomposed Miner on top of Inductive Miner.

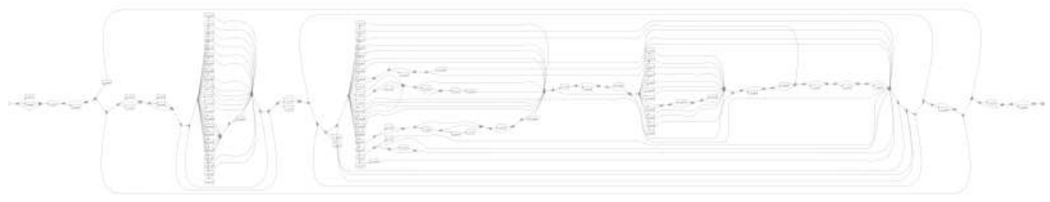


Figure 39: Process model discovered from the BPI15-1 log by Inductive Miner.

For the BPI17 log, the results are similar to those of the BPI12 log. SPM can produce simpler models with higher F-score than those discovered by Inductive Miner and Fodina, while Decomposed Miner has a highly complex model and Region-Based Miner suffers from a low-precision model.

Finally, Table 11 illustrates the results of SPM without the Adjusting step. In comparison with the results shown in Table 10, we can see that

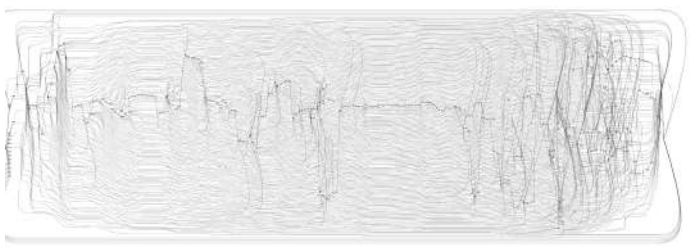


Figure 40: Process model discovered from the BPI15-1 log by Fodina.

SPM with the Adjusting step always improves the quality of the process model (both in terms of precision and complexity), particularly in the case of the BPI15 logs. As an example, Figure 43 shows the process model discovered from the BPI15-1 log without the Adjusting step. This is visibly more complex than the same extract of model, using the Adjusting step, as shown in Figure 42.

Log name	**Discovery technique**	**Accuracy**			**Complexity**			**Sound?**	**Runtime (seconds)**
		Fitness	**Precision**	**F-score**	**Size**	**CFC**	**Struct.**		
BPI12	SPM_{im}	0.89	0.9	0.9	48	28	1.0	yes	68
	SPM_{fo}	0.81	0.98	0.88	46	32	0.54	yes	122
BPI13	SPM_{im}	0.78	0.76	0.77	97	67	0.83	yes	494
	SPM_{fo}	0.8	0.67	0.73	83	120	0.23	yes	858
BPI15-1	SPM_{im}	0.75	0.4	0.52	238	144	0.92	yes	680
	SPM_{fo}	0.85	0.4	0.54	335	423	0.21	yes	1,216
BPI15-2	SPM_{im}	0.73	0.65	0.69	214	105	1.0	yes	1,444
	SPM_{fo}	-	-	-	321	525	0.18	yes	timed out
BPI15-3	SPM_{im}	0.77	0.63	0.69	260	144	1.0	yes	397
	SPM_{fo}	0.86	0.41	0.56	293	328	0.27	yes	540
BPI15-4	SPM_{im}	0.8	0.49	0.61	232	136	1.0	yes	405
	SPM_{fo}	0.87	0.43	0.57	274	322	0.19	yes	557
BPI15-5	SPM_{im}	0.81	0.52	0.63	272	155	0.82	yes	1333
	SPM_{fo}	-	-	-	312	396	-	no	timed out
BPI17	SPM_{im}	0.92	0.74	0.82	53	28	1.0	yes	425
	SPM_{fo}	0.97	0.64	0.77	68	105	0.13	yes	760

Table 11: SPM without the Adjusting step.

6. Conclusion

This article put forward a technique to identify stages from an event log in a way that seeks to maximize modularity. The experimental evaluation showed that this modularity-driven method leads to stage-based process decompositions that are closer to human expert decompositions, compared to

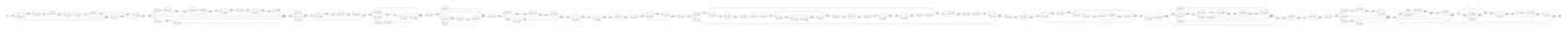


Figure 41: Process model discovered from the BPI15-1 log by SPM on top of Inductive Miner.

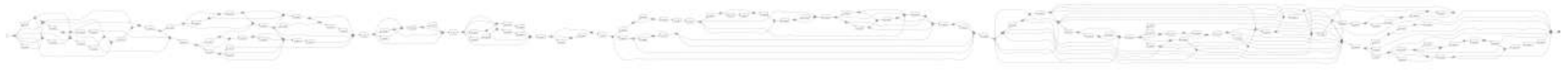


Figure 42: Process model discovered from the BPI15-1 log by SPM on top of Fodina.

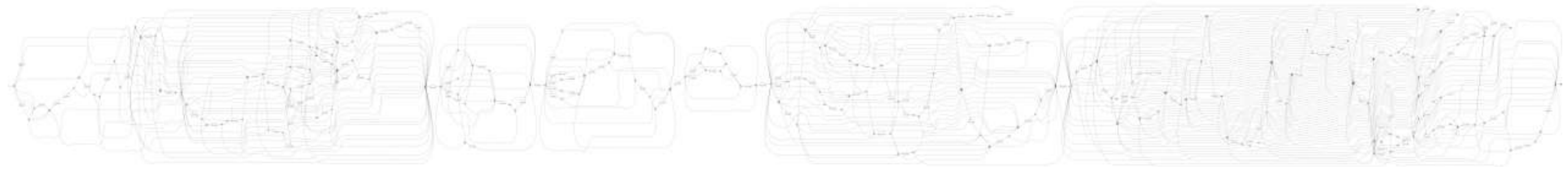


Figure 43: Process model discovered from the BPI15-1 log by SPM on top of Fodina without the Adjusting step.

non-modular decomposition approaches previously proposed in the literature. This result confirms previous findings that human experts intuitively produce decompositions of business processes with high modularity [11].

Based on this modularity-driven stage decomposition method, the article also presented a new divide-and-conquer tecnique for automated process discovery (i.e. discovering process models from event logs). The experimental evaluation showed that the proposed stage-based automated process discovery technique discovers relatively simple models for each stage, and merges them together into a final process model with higher F-score than existing flat and divide-and-conquer automated process discovery techniques.

Besides automated process discovery, the proposed stage-based decomposition method has other potential applications in the field of process mining. For example, in [45] we have shown that stage-based decompositions can be used to produce effective visualizations for process performance analysis. An avenue for future work is to develop a comprehensive process performance analysis framework based on automatically extracted stage-based process decompositions. Another potential application is in the field of conformance checking, where divide-and-conquer approaches have been proposed to scale-up existing conformance checking techniques [46].

Beyond the field of process mining, the proposed stage-based decomposition method could find applications in customer journey analysis, by allowing analysts to identify stages from customer session logs. It may be possible to extend the proposed stage identification technique to compute abstracted views of large event sequences for interactive visual data mining.

The proposed automated process decomposition method is inherently limited in scope to identifying linearly ordered stages. Such linear structures can be found for example in application-to-approval and order-to-cash processes. However, they cannot always be found in highly variable business processes such as those found in the healthcare domain (e.g. patient flows). In these processes, there may be parallel and alternative stages as well as frequent jumps between stages. A possible direction for future work is to extend the proposed automated process decomposition method in order to identify stages that are not linearly ordered and stages that do not have clear boundaries (e.g. with frequent back-and-forth transitions between stages). A possible direction for tackling this challenge is to combine existing automated process decomposition approaches (based on clustering) with the modularity-driven approach proposed in this article.

Acknowledgments. This research is funded by the Australian Research

Council (grant DP150103356) and the Estonian Research Council (grant IUT20-55).

References

[1] M. Dumas, M. La Rosa, J. Mendling, H. Reijers, Fundamentals of Business Process Management, Springer 2nd Edition, 2018.

[2] W. van der Aalst, Process Mining: Data Science in Action, Springer, 2016.

[3] A. Weijters, W. M. van Der Aalst, A. A. De Medeiros, Process mining with the heuristics miner-algorithm, Technische Universiteit Eindhoven, Tech. Rep. WP 166 (2006) 1–34.

[4] S. J. Leemans, D. Fahland, W. M. van der Aalst, Discovering block-structured process models from event logs containing infrequent behaviour, in: International Conference on Business Process Management, Springer, 2013, pp. 66–78.

[5] S. vanden Broucke, J. De Weerdt, J. Vanthienen, B. Baesens, Fodina: a robust and flexible heuristic process discovery technique, Decision Support Systems.

[6] A. Augusto, R. Conforti, M. Dumas, M. L. Rosa, F. M. Maggi, A. Marrella, M. Mecella, A. Soo, Automated discovery of process models from event logs: Review and benchmark, CoRR abs/1705.02288.

[7] 4TU Data Center, BPI Challenge 2012 Event Log (2012). doi:10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f.

[8] J. Carmona, J. Cortadella, M. Kishinevsky, Divide-and-conquer strategies for process mining, in: Proc. of BPM, Springer, 2009, pp. 327–343.

[9] W. M. Van der Aalst, Decomposing Petri nets for process mining: A generic approach, Distributed and Parallel Databases 31 (4) (2013) 471–507.

[10] H. Verbeek, W. van der Aalst, J. Munoz-Gama, Divide and conquer: A tool framework for supporting decomposed discovery in process mining, The Computer Journal (2017) 1–26.

[11] H. A. Reijers, J. Mendling, R. M. Dijkman, Human and automatic modularizations of process models to enhance their comprehension, Inf. Syst. 36 (5) (2011) 881–897.

[12] H. Nguyen, M. Dumas, A. H. ter Hofstede, M. La Rosa, F. M. Maggi, Mining business process stages from event logs, in: International Conference on Advanced Information Systems Engineering, Springer, 2017, pp. 167–185.

[13] W. M. P. van der Aalst, T. Weijters, L. Maruster, Workflow mining: discovering process models from event logs, IEEE TKDE 16 (9) (2004) 1128–1142.

[14] A. Augusto, R. Conforti, M. Dumas, M. La Rosa, G. Bruno, Automated discovery of structured process models: Discover structured vs. discover and structure, in: Conceptual Modeling: 35th International Conference, ER 2016, Gifu, Japan, November 14-17, 2016, Proceedings 35, Springer, 2016, pp. 313–329.

[15] A. K. de Medeiros, A. J. Weijters, W. M. van der Aalst, Genetic process mining: an experimental evaluation, Data Mining and Knowledge Discovery 14 (2) (2007) 245–304.

[16] J. C. Buijs, B. F. Van Dongen, W. M. van Der Aalst, On the role of fitness, precision, generalization and simplicity in process discovery, in: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems", Springer, 2012, pp. 305–322.

[17] J. M. E. Van der Werf, B. F. van Dongen, C. A. Hurkens, A. Serebrenik, Process discovery using integer linear programming, in: International Conference on Applications and Theory of Petri nets, Springer, 2008, pp. 368–387.

[18] W. M. Van Der Aalst, Decomposing process mining problems using passages, in: International Conference on Application and Theory of Petri nets and Concurrency, Springer, 2012, pp. 72–91.

[19] R. Conforti, M. Dumas, L. García-Bañuelos, M. La Rosa, BPMN miner: Automated discovery of BPMN process models with hierarchical structure, Inf. Syst. 56 (2016) 284–303.

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

[21] C. C. Ekanayake, M. Dumas, L. García-Bañuelos, M. La Rosa, Slice, mine and dice: Complexity-aware automated discovery of business process models, in: International Conference on Business Process Management (BPM), Springer, 2013, pp. 49–64.

[22] J. De Weerdt, J. Vanthienen, B. Baesens, Active trace clustering for improved process discovery, Knowledge and Data Engineering, IEEE Transactions on 25 (12) (2013) 2708–2720.

[23] L. Raichelson, P. Soffer, E. Verbeek, Merging event logs: Combining granularity levels for process flow analysis, Information Systems 71 (2017) 211 – 227. doi:https://doi.org/10.1016/j.is.2017.08.010.

[24] J. Claes, G. Poels, Merging event logs for process mining: A rule based merging method and rule suggestion algorithm, Expert Systems with Applications 41 (16) (2014) 7291 – 7306. doi:https://doi.org/10.1016/j.eswa.2014.06.012.

[25] J. C. A. M. Buijs, B. F. van Dongen, W. M. P. van der Aalst, Mining configurable process models from collections of event logs, in: Business Process Management - 11th International Conference, BPM 2013, Beijing, China, August 26-30, 2013. Proceedings, 2013, pp. 33–48. doi:10.1007/978-3-642-40176-3_5.

[26] L. García-Bañuelos, M. Dumas, M. L. Rosa, J. D. Weerdt, C. C. Ekanayake, Controlled automated discovery of collections of business process models, Information Systems 46 (2014) 85–101. doi:10.1016/j.is.2014.04.006.

[27] M. E. Newman, M. Girvan, Finding and evaluating community structure in networks, Physical review E 69 (2) (2004) 026113.

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, The MIT Press, 2009.

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

[30] A. Adriansyah, J. Muñoz-Gama, J. Carmona, B. van Dongen, W. van der Aalst, Measuring precision of modeled behavior, ISeB 13 (1) (2015) 37–67.

[31] N. Tax, X. Lu, N. Sidorova, D. Fahland, W. M. P. van der Aalst, The imprecisions of precision measures in process mining, Inf. Process. Lett. 135 (2018) 1–8.

[32] A. Augusto, A. Armas-Cervantes, R. Conforti, M. Dumas, M. L. Rosa, D. Reißner, Abstract-and-compare: A family of scalable precision measures for automated process discovery, in: Proceedings of the 16th International Conference on Business Process Management (BPM), Springer, 2018, pp. 158–175.

[33] A. Augusto, R. Conforti, M. Dumas, M. La Rosa, F. Maggi, A. Marrella, M. Mecella, A. Soo, Automated discovery of process models from event logs: Review and benchmark, IEEE Transactions on Knowledge and Data EngineeringTo appear.

[34] W. van der Aalst, K. van Hee, A. ter Hofstede, N. Sidorova, H. Verbeek, M. Voorhoeve, M. Wynn, Soundness of workflow nets: classification, decidability, and analysis, Formal Asp. Comput. 23 (3).

[35] A. Adriansyah, J. Munoz-Gama, J. Carmona, B. F. van Dongen, W. M. van der Aalst, Alignment based precision checking, in: Proceedings of BPM, Springer, 2012, pp. 137–149.

[36] C. Favre, D. Fahland, H. Völzer, The relationship between workflow graphs and free-choice workflow nets, Information Systems 47 (2015) 197–219.

[37] A. Augusto, R. Conforti, M. Dumas, M. La Rosa, A. Polyvyanyy, Split Miner: automated discovery of accurate and simple business process models from event logs, Knowledge and Information Systems 59 (2) (2019) 251–284.

[38] C. Favre, H. Völzer, The difficulty of replacing an inclusive or-join, in: International Conference on Business Process Management, Springer, 2012, pp. 156–171.

[39] 4TU Data Center, BPI Challenge 2013 Event Log (2013). doi:doi:10.4121/uuid:500573e6-accc-4b0c-9576-aa5468b10cee.

[40] 4TU Data Center, BPI Challenge 2015 Event Log (2015). doi:doi:10.4121/uuid:31a308ef-c844-48da-948c-305d167a0ec1.

[41] 4TU Data Center, BPI Challenge 2017 Event Log (2017). doi:doi:10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b.

[42] M. Halkidi, Y. Batistakis, M. Vazirgiannis, On clustering validation techniques, J. Intell. Inf. Syst. 17 (2-3) (2001) 107–145.

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

[44] J. Mendling, Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness, Springer, 2008.

[45] H. Nguyen, M. Dumas, A. H. ter Hofstede, M. La Rosa, F. M. Maggi, Business process performance mining with staged process flows, in: Proc. of CAiSE, Springer, 2016, pp. 167–185.

[46] S. K. L. M. vanden Broucke, J. Munoz-Gama, J. Carmona, B. Baesens, J. Vanthienen, Event-based real-time decomposed conformance analysis, in: Proceedings of the OTM 2014 Confederated International Conferences, Springer, 2014, pp. 345–363.