Simplification of Complex Process Models by Abstracting Infrequent Behaviour

David Chapela-Campa, Manuel Mucientes, and Manuel Lama

Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS)
Universidade de Santiago de Compostela. Santiago de Compostela, Spain
{david.chapela, manuel.mucientes, manuel.lama}@usc.es

Abstract. Several simplification techniques have been proposed in process mining to improve the interpretability of complex processes, such as the structural simplification of the model or the simplification of the log. However, obtaining a comprehensible model explaining the behaviour of unstructured large processes is still an open challenge. In this paper, we present WoSimp, a novel algorithm to simplify processes by abstracting the infrequent behaviour from the logs, allowing to discover a simpler process model. This algorithm has been validated with more than 10 complex real processes, most of them from Business Process Management Challenges. Experiments show that WoSimp simplifies the process log and allows to discover a better process model than the state of the art techniques.

Keywords: event abstraction, model simplification, log simplification, process mining

1 Introduction

During the past years process mining has emerged as a discipline focusing on techniques to discover, monitor and enhance real processes [1]. One of the key areas of process mining is process discovery, whose objective is to generate a process model describing the behaviour of the event log of a process. Once a model is discovered, the analysis and enhancement of the process can be performed to detect possible improvements. However, in scenarios where the quality of the discovered process model is far too low —e.g. spaghetti models—, this analysis and enhancement become more difficult.

With the entrance of process mining in the *Big Data* era, these complex and incomprehensible processes have become more and more common. Different simplification techniques have been developed with the objective of obtaining an understandable process model, in order to be able to analyze and enhance the real process behind it. The first proposals focused on a structural simplification of the process model using only the information of the model itself [2]. But they quickly evolved to simplify the process model using also the information from the event log [3, 4]. The drawback of these structural simplification techniques

is that, to maintain the fitness, they may produce unstructured models that deteriorate the understandability of the process.

Other approaches first simplify the log, and then discover an understandable process model. Some of these techniques search for outliers in the log traces, removing them with the aim of retaining the frequent behaviour of the process. In [5] this outlier identification is performed by using the probability of occurrence of each event conditioned by both its k predecessors and its k successors —e.g. how probable is that a follows, or is followed by, the sequence $\langle b, c \rangle$. One drawback of this technique is that, due to the use of sequential conditional probability, parallel relations are not considered. There are also approaches, like [6], that entirely remove activites from the log based on their contribution to the chaotic structure of the process. One drawback of this technique is that the decision of removing an activity depends on its relations with all the other activities, making the approach unscalable when the number of activities grows. Furthermore, the removal of activities from the log can produce the loss of important information if the activity is chaotic in some scenarios, but not in others.

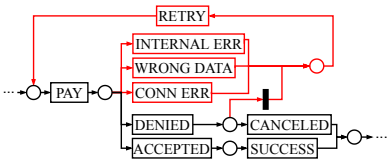
An approach overcoming some of the previous drawbacks is the abstraction of subparts —subprocesses— of the process. This procedure replaces subprocesses with new activities, either in the log or structurally in the model. In [7] the authors propose a supervised method to abstract in the log behavioural activity patterns that capture domain knowledge. Given a set of activity patterns, they compose an abstraction model and align the behaviour of this abstraction model with the original log, creating an abstracted event log. The need of expert domain knowledge is solved in [8], where an unsupervised version of this method is proposed. This technique uses frequent local process models [9] as the activity patterns to abstract. The drawback of this technique is the significant penalization in its quality due to the abstraction of frequent subprocesses —the removal of frequently executed behaviour penalizes the fitness, and the addition of activities not recorded in the log the precision. This abstraction does not help to discover a significantly better process model in terms of F-score, not even undoing the parts of the abstraction after the discovery, as shown in their experimentation.

Fig. 1 shows a motivational example, where an ideal abstraction of the infrequent behaviour is performed allowing to focus in the frequent one. In this case, the frequent behaviour is related to the paths through `DENIED-CANCELED` and through `ACCEPTED-SUCCESS`. The removal of the other —infrequent— traces would cause a lost of all the behaviour in each trace, not only in the infrequent one. For instance, the behaviour previous to `PAY` in these infrequent traces might be important in an analysis phase. Table 1c and Fig. 1d show an abstraction where the infrequent behaviour of the paths going through the loop is encapsulated in one activity, `ERROR AND RETRY`, letting the rest untouched.

In this paper, we present WoSimp, a novel algorithm to simplify processes by abstracting the infrequent behaviour from the log and maintaining the more frequent one, allowing to discover a simpler process model. The main novelty of our approach is that it detects the frequent behaviour of the process in a first

Trace	
... PAY - DENIED - CANCELED ...	(×27)
... PAY - ACCEPTED - SUCCESS ...	(×43)
... PAY - INTERNAL ERR - RETRY - PAY ...	(×6)
... PAY - WRONG DATA - RETRY - PAY ...	(×7)
... PAY - CONN ERR - RETRY - PAY ...	(×9)
... PAY - DENIED - RETRY - PAY ...	(×8)

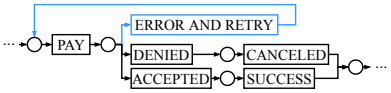
(a) Partial event log.



(b) Model for log in Table 1a.

Trace	
... PAY - DENIED - CANCELED ...	(×27)
... PAY - ACCEPTED - SUCCESS ...	(×43)
... PAY - ERROR AND RETRY - PAY ...	(×30)

(c) Abstracted event log.



(d) Model for log in Table 1c.

Fig. 1. Motivational example for the algorithm presented in this paper.

phase —using the frequent patterns extracted by WoMine [10]— and abstracts the infrequent behaviour in a second phase. The use of WoMine to detect frequent behaviour allows our technique to retain not only frequent activities, but frequent subprocesses, abstracting the infrequent behaviour which obfuscates the understanding of the overall process. The algorithm has been validated with a set of 11 complex real process logs, 10 of them from Business Process Management Challenges, and one from the health domain. Experiments show that WoSimp simplifies the process log allowing to discover better process models than the state of the art techniques.

2 Preliminaries

In this paper, we represent process models with place/transition Petri nets [11] (P/T Petri net) due to its higher comprehensibility, and the easiness to explain the executed behaviour. A P/T Petri net (Definition 1) is a directed bipartite graph composed by two kinds of nodes: places and transitions —circles and boxes, respectively—, and where arcs connect two nodes of different type, as can be seen in Fig. 2a. Being A the set of activities of a process, each transition in a Petri net modeling its behaviour is identified by a label corresponding to the activity it represents. We assume that the transition labels are unique, i.e. there are no repeated activities in the net. An exception is made for silent transitions, which are unlabeled. Silent transitions are only executed for routing purposes and do not correspond to any activity of the process.

Definition 1 (Petri net). A Petri net is a tuple $M = (P, T, F)$, where

- P is a finite set of places;
- T is a finite set of transitions;
- $P \cap T = \emptyset$; and
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs.

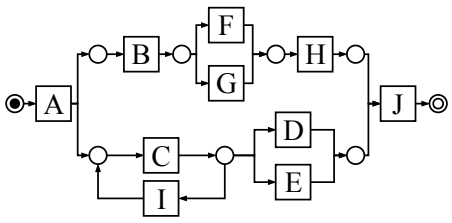
We denote as $\bullet t$ and $t\bullet$ the input and output places of $t \in T$ (according to F). The state of a Petri net is defined by the marking function $m : P \nrightarrow A$. m is a partial function returning, for each place $p \in P$, the label of a transition —representing a token— or $\emptyset$ if there are no tokens in that place. The label of a token corresponds with the transition which has produced it. Therefore, a transition t is said to be *enabled* if $\forall p \in \bullet t$, $m(p) \neq \emptyset$. The execution of an enabled transition t consumes a token in each $p \in \bullet t$, and produces another token with its label in each $p \in t\bullet$. Silent transitions maintain the label of the consumed tokens in those it produces. The difference with a usual marking is that the tokens carry the label of their producing transition. This allows to know, when a transition is executed, which visible transitions have produced the tokens it consumed.

Definition 2 (Event, Trace and Event Log). An event ε corresponds to the execution of the activity $\alpha \in A$ in a particular instant. A trace is a list (sequence) $\tau = \langle \varepsilon_1, ..., \varepsilon_n \rangle$ of events ε_i occurring at a time index i relative to the other events in τ. Each trace corresponds to an execution of the process, i.e., a process instance. An event log $L = [\tau_1, ..., \tau_m]$ is a multiset of traces τ_i.

In this paper, to ease the comprehension, an event is represented only with the label of the executed activity, but usually events store more information as timestamps, resources, etc. Nevertheless, it is important to distinguish between an activity —an action from a process that can be modeled with a single transition in the Petri net— and an event —a single execution of an activity. The replacement of an activity implies the replacement of all its events and the transition in the Petri net, but the replacement of an event only implies the replacement of that single execution.

Definition 3 (Behavioural Event). A behavioural event is a tuple $\beta = (\mathcal{B}^\beta, \alpha)$ where:

- $\alpha \in A$ is the activity which execution is recorded in the behavioural event; and
- $\mathcal{B}^\beta$ is the set of behavioural events which have produced the tokens consumed by the execution of α.



(a) Petri net example.

$\langle A, C, B, I, C, F, H, E, J \rangle$

(b) Trace example.

$\langle (\emptyset, A), (\{A\}, C), (\{A\}, B),$
$(\{C\}, I), (\{I\}, C), (\{B\}, F),$
$(\{F\}, H), (\{C\}, E), (\{H, E\}, J) \rangle$

(c) Behavioural trace example.

Fig. 2. Example of a Petri net, a trace, and the corresponding behavioural trace obtained by replaying the trace in the Petri net. For the sake of simplicity, in each behavioural event, $\mathcal{B}^\beta$ is represented as a set of activities instead of behavioural events.

Similar to an event, a behavioural event can store more information like timestamps, resources, etc. Moreover, a behavioural event also stores its causal inputs, i.e., the previous behavioural events which produced the tokens it consumed. An example can be seen in Fig. 2c, where 9 behavioural events are shown. For instance, the last behavioural event, $(\{H, E\}, J)$, records the execution of J, consuming the tokens generated by the executions of H and E.

Definition 4 (Behavioural Trace). Let M be the Petri net of a process, and $\tau = \langle \varepsilon_1, ..., \varepsilon_n \rangle$ a trace of the same process. The corresponding behavioural trace of τ w.r.t. M is the sequence $\pi = \langle \beta_1, ..., \beta_n \rangle$ of behavioural events. π is the result of a replay of all $\varepsilon_i \in \tau$ in M, extending each ε_i by adding the behavioural events corresponding to the execution of each $\alpha' \in m_i(p)$ for all $p \in {}^\bullet\alpha$, being α the activity executed in ε_i —i.e., the behavioural events producing the tokens consumed by ε_i.

Fig. 2c shows the behavioural trace obtained by replaying the trace in Fig. 2b in the Petri net of Fig. 2a.

Definition 5 (Behavioural Log). We define a behavioural event log, or behavioural log, as a multiset $L^\pi = [\pi_1, ..., \pi_m]$ of behavioural traces π_i.

Definition 6 (Abstraction). Given a behavioural trace π, and being A_π the set of activities executed in π. We define an abstraction in π as $\lambda = (\varepsilon^{abs}, \mathcal{B}, A_I, A_O)$ where:

- ε^{abs} is an event representing the execution of an abstracted activity;
- $\mathcal{B}$ is a set of behavioural events from π to be replaced with ε^{abs};
- $A_I \subset A_\pi$ is a set of activities of the events causing the execution of any event in $\mathcal{B}$; and
- $A_O \subset A_\pi$ is a set of activities of the events in π whose execution is caused by events in $\mathcal{B}$,

such that:

- $\mathcal{B}_I = \{\beta' \mid \beta' \in \mathcal{B}^\beta \wedge \beta \in \mathcal{B}\}$;
- $A_I = \{\beta.activity \mid \beta \in \mathcal{B}_I \setminus \mathcal{B}\}$;
- $\mathcal{B}_O = \{\beta' \mid \beta' \in \pi \wedge \beta \in \mathcal{B}^{\beta'} \wedge \beta \in \mathcal{B}\}$; and
- $A_O = \{\beta.activity \mid \beta \in \mathcal{B}_O \setminus \mathcal{B}\}$.

For instance, being π the trace depicted in Fig. 2c, an abstraction could be formed by a new activity *Abs* as ε^{abs}; $(\{A\}, B)$, $(\{B\}, F)$ and $(\{F\}, H)$ as $\mathcal{B}$; being A the only activity in A_O; and J the only activity in A_I. After the abstraction process, ε^{abs} would replace the behavioural events of $\mathcal{B}$. Related to Definition 6, we use the term *empty abstraction*, represented by $\lambda^\emptyset = (\mathcal{B}, A_I, A_O)$, to define an abstraction without assigned event, and the term *anti-abstraction*, represented by $\bar{\lambda}$, to define the set of behavioural events of a behavioural trace to keep in the abstracted log, i.e., those events not to be abstracted.

3 WoSimp Algorithm

In this section we present WoSimp (Alg. 1), an algorithm to abstract the infrequent behaviour of a log. The execution of a discovery algorithm over that abstracted log allows to obtain a more precise and simpler process model, keeping a good fitness. Our proposal takes as input an event log, a process model and a frequency threshold, and returns the event log with the abstraction of the infrequent behaviour. The first step of WoSimp is to identify the frequent behaviour to be kept in the log. For this purpose WoMine [10] is used (Alg. 1: 2), extracting from the process model a set of behavioural patterns executed in a percentage of the traces of the log frequent w.r.t. the defined threshold. Later, the behavioural log with the causal relations of each event is obtained using the given log and the model (Alg. 1: 3). Then, the algorithm builds the abstractions of the behaviour not covered by the frequent patterns (Alg. 1: 4). Finally, the log is abstracted with function `abstractLog` (Alg. 1: 5): each behavioural trace is converted to a trace —removing the behavioural information—, abstracting those behavioural events defined in the abstractions.

Algorithm 1. WoSimp algorithm.

Input: An event log $L = [\tau_1, ..., \tau_m]$ of traces, a process model M, and a threshold t.
Output: An event log $L' = [\tau'_1, ..., \tau'_m]$ with the infrequent behaviour of L abstracted into new activities.

1 **Algorithm** `WoSimp(`L, M, t`)`
2 $P \leftarrow$ `getFrequentPatterns(`L, M, t`) // using algorithm in [10]`
3 $L^{\pi} \leftarrow$ obtain the behavioural log of L and M `// Definition 5`
4 $\Lambda \leftarrow$ `buildAbstractions(`L^{π}, P`)`
5 $L' \leftarrow$ `abstractLog(`L^{π}, Λ`)`
6 **return** L'
7 **Function** `buildAbstractions(`L^{π}, P`)`
8 $\Lambda^{\emptyset} \leftarrow \emptyset$
9 **forall** $\pi \in L^{\pi}$ **do**
10 $\bar{\lambda} \leftarrow \{\beta \mid \beta \in \pi \wedge \beta \in p.executedEvents[\pi] \wedge p \in P\}$
11 $\Lambda^{\emptyset}_{\pi} \leftarrow$ `obtainEmptyAbstractions(`π, $\bar{\lambda}$`) // Alg. 2`
12 $\Lambda^{\emptyset} \leftarrow \Lambda^{\emptyset} \cup \Lambda^{\emptyset}_{\pi}$
13 **end**
14 $\Lambda \leftarrow$ `assignAbstractedEvents(`$\Lambda^{\emptyset}$`) // Alg. 3`
15 **return** Λ
16 **Function** `abstractLog(`L^{π}, Λ`)`
17 $L' \leftarrow L^{\pi}$
18 **forall** $\lambda \in \Lambda$ **do**
19 replace $\lambda.\mathcal{B}$ with $\lambda.\varepsilon^{abs}$ and insert it in L'
20 **end**
21 **return** L'

A naive example can be seen in Fig. 1 where, with a threshold of 25%, the frequent patterns obtained by WoMine cover the behaviour going through `DENIED - CANCELED` and through `ACCEPTED - SUCCESS`. These patterns allow to abstract the sequences starting in `INTERNAL ERR`, `WRONG DATA`, `CONN ERR` and `DENIED`, and going through `RETRY`. The abstraction technique encapsulates all these behaviour in one abstraction —named `ERROR AND RETRY` in Table 1c—, allowing to discover the process as shown in Fig. 1d.

The technique designed to build the abstractions is composed by two phases. The first phase (Alg. 1: 9-13) is an horizontal analysis, i.e. one trace at a time, creating the groups of behavioural events to abstract. For each trace, the behavioural events belonging to an execution of a frequent pattern are collected in their anti-abstraction (Alg. 1: 10). Then, function `obtainEmptyAbstractions` groups the behavioural events to be abstracted creating the empty abstractions —abstractions without an abstracted event assigned— corresponding to that trace (c.f. Sec. 3.1). In the second phase (Alg. 1: 14), a vertical analysis going over the log is performed to create the abstractions by assigning an abstracted event with the same activity to the empty abstractions with identical contextual

Algorithm 2. Get empty abstractions of a behavioural trace (Alg. 1: 11).

Input: A behavioural trace π and its anti-abstraction $\bar{\lambda}$.
Output: A set $\Lambda^{\emptyset}$ with the empty abstractions of the behavioural trace π.

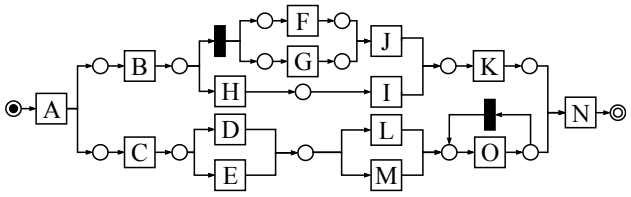
1 **Algorithm** `obtainEmptyAbstractions(`π, $\bar{\lambda}$`)`
2 $\quad \mathcal{B}_{infreq} \leftarrow \{\beta \mid \beta \in \pi \wedge \beta \notin \bar{\lambda}\}$
3 $\quad \mathcal{B}_{connected} \leftarrow$ `groupConnectedEvents(`$\mathcal{B}_{infreq}$`) // set of sets of` β
4 $\quad \Lambda^{\emptyset} \leftarrow \emptyset$
5 $\quad$ **forall** $\mathcal{B} \in \mathcal{B}_{connected}$ **do**
6 $\quad\quad \lambda^{\emptyset} \leftarrow$ `obtainEmptyAbstraction(`π, $\mathcal{B}$`)`
7 $\quad\quad \Lambda^{\emptyset} \leftarrow \Lambda^{\emptyset} \cup \{\lambda^{\emptyset}\}$
8 $\quad$ **end**
9 $\quad$ **return** $\Lambda^{\emptyset}$
10 **Function** `groupConnectedEvents(`$\mathcal{B}_{infreq}$`)`
11 $\quad \mathcal{B}_{connected} \leftarrow \emptyset$ `// set of sets of` β
12 $\quad$ **forall** $\beta \in \mathcal{B}_{infreq}$ **do**
13 $\quad\quad$ **if** $\beta \notin \cup\mathcal{B}_{connected}$ **then**
14 $\quad\quad\quad \mathcal{B}' \leftarrow \{\beta\} \cup \{\beta' \mid \beta' \in \mathcal{B}_{infreq} \wedge (\beta' \rightarrow \beta \vee \beta \rightarrow \beta')\}$
15 $\quad\quad\quad \mathcal{B}_{connected} \leftarrow \mathcal{B}_{connected} \cup \{\mathcal{B}'\}$
16 $\quad\quad$ **end**
17 $\quad$ **end**
18 $\quad$ **return** $\mathcal{B}_{connected}$
19 **Function** `obtainEmptyAbstraction(`π, $\mathcal{B}$`)`
20 $\quad A_I \leftarrow \{\beta'.activity \mid \beta' \in (\mathcal{B}^{\beta} \setminus \mathcal{B}) \wedge \beta \in \mathcal{B}\}$
21 $\quad A_O \leftarrow \{\beta'.activity \mid \beta' \in (\pi \setminus \mathcal{B}) \wedge \beta \in \mathcal{B}^{\beta'} \wedge \beta \in \mathcal{B}\}$
22 $\quad \lambda^{\emptyset} \leftarrow (\mathcal{B}, A_I, A_O)$
23 $\quad$ **return** $\lambda^{\emptyset}$

behaviour (c.f. Sec. 3.2). Finally, with all the information of the abstractions, function `abstractLog` abstracts the original behavioural log replacing the behavioural events of each abstraction with the corresponding abstracted event (Alg. 1: 16).

3.1 Create Abstractions of Infrequent Behaviour from a Trace

The objective of the first phase is to create the empty abstractions with the infrequent behaviour in each trace by grouping the corresponding behavioural events. Alg. 2 shows this abstraction process over a trace. First, the behavioural events to abstract are collected, i.e., those not present in the anti-abstraction (Alg. 2: 2). Then, these behavioural events are grouped, where each group contains those connected between them (Alg. 2: 3). Afterwards, an empty abstraction is created for each group (Alg. 2: 5-8). Function `obtainEmptyAbstraction` creates this empty abstraction with *i)* the set of behavioural events to abstract; *ii)* the inputs of this group, i.e., for each behavioural event from the group, the activities of its input behavioural events not contained in the abstraction group (Alg. 2: 20); and *iii)* the outputs of this group, i.e., the activities of the behavioural events of the trace having as inputs any of the behavioural events in the group (Alg. 2: 21).

As an example, the process model and the two traces from Fig. 3 are going to be used. Assuming a balanced distribution in the selections, and a threshold for the patterns of 70%, WoMine recovers as frequent patterns the initial AND-split (A, B and C) and the final AND-join without the loop (K, O and N). Table 1 shows the results of the main steps of the first phase over the two traces of Fig. 3. To create the groups with the connected behavioural events not present in the anti-abstractions —those unmarked in the trace description— the algorithm performs a forward iteration over them adding each behavioural event to the set where its inputs are. The results can be seen in the $\mathcal{B}_{connected}$ elements. Then, an empty abstraction is created for each group (e.g. $\lambda_1^{\emptyset}$) with the behavioural events of the group (e.g. $\{F, G, J\}$), the input activities of these behavioural events (e.g. $\{B\}$), and the activities of the behavioural events from π whose inputs are in the group (e.g. $\{K\}$). For instance, the input activity for $\lambda_1^{\emptyset}$ is only B because is



(a) Petri net of a process to abstract.

$\langle A, B, F, C, D, G, L, J, O, K, N \rangle$

(b) Trace example.

$\langle A, C, E, B, H, L, I, O, O, K, O, N \rangle$

(c) Trace example.

Fig. 3. Petri net and two traces to exemplify the abstraction process.

Table 1. Key elements obtained in the first phase of the algorithm for the traces in Fig. 3 —events with a hat in each trace are those belonging to the anti-abstraction. π: the corresponding behavioural trace. $\mathcal{B}_{connected}$: the groups of behavioural events to abstract —to ease the visualization the behavioural events from each $\mathcal{B}_{connected}$ are shown as simple events. $\Lambda^{\emptyset}$: the empty abstractions created from these groups.

$\tau_1 = \langle \hat{A}, \hat{B}, F, \hat{C}, D, G, L, J, \hat{O}, \hat{K}, \hat{N} \rangle$	
π_1	$\langle(\emptyset, A), (\{A\}, B), (\{B\}, F), (\{A\}, C), (\{C\}, D), (\{B\}, G), (\{D\}, L), (\{F, G\}, J),$ $(\{L\}, O), (\{J\}, K), (\{O, K\}, N)\rangle$
$\mathcal{B}_{connected}$	$\{F, G, J\}$ and $\{D, L\}$
$\Lambda^{\emptyset}$	$\lambda_1^{\emptyset} = (\{F, G, J\}, \{B\}, \{K\})$ $\lambda_2^{\emptyset} = (\{D, L\}, \{C\}, \{O\})$
$\tau_2 = \langle \hat{A}, \hat{C}, E, \hat{B}, H, L, I, O, O, \hat{K}, \hat{O}, \hat{N} \rangle$	
π_2	$\langle(\emptyset, A), (\{A\}, C), (\{C\}, E), (\{A\}, B), (\{B\}, H), (\{E\}, L), (\{H\}, I), (\{L\}, O), (\{O\}, O),$ $(\{I\}, K), (\{O\}, O), (\{K, O\}, N)\rangle$
$\mathcal{B}_{connected}$	$\{E, L, O, O\}$ and $\{H, I\}$
$\Lambda^{\emptyset}$	$\lambda_3^{\emptyset} = (\{E, L, O, O\}, \{C\}, \{O\})$ $\lambda_4^{\emptyset} = (\{H, I\}, \{B\}, \{K\})$

the firing behavioural event of F and G, and the firing behavioural events of J are inside the group. For the output activities, the behavioural events of π_1 are inspected, searching for those whose firing behavioural events are in the group, i.e., K.

3.2 Activity Assignment to Each Abstraction

Once each trace has its infrequent behaviour grouped in the different empty abstractions, the second phase starts (Alg. 3). In this phase, all the empty abstractions of the log are compared to assign an event with the same activity to those with identical contextual behaviour —coming from the same activities or going to the same activities in the model. For this, the empty abstractions are first grouped by their input activities (Alg. 3: 3-8). Then, these groups are merged by their output activities, i.e., the groups sharing the output activities of all their empty abstractions are merged (Alg. 3: 10-15). Finally, an activity is created for each group of empty abstractions and assigned to each of them (Alg. 3: 17-23).

Continuing with the example in Table 1, the second phase groups all the empty abstractions first by their input activities obtaining two groups: $\{\lambda_1, \lambda_4\}$ and $\{\lambda_2, \lambda_3\}$. The grouping by their outputs does not merge any group because the output activities of the empty abstractions in the first group are $\{K\}$, and the output activites of the second group are $\{O\}$. Once the empty abstractions are grouped, the assignation of artificial activities is performed. An event with the activity Abs_1 is assigned to the empty abstractions $\lambda_1^{\emptyset}$ and $\lambda_4^{\emptyset}$, and other event with activity Abs_2 to $\lambda_2^{\emptyset}$ and $\lambda_3^{\emptyset}$, obtaining the corresponding abstractions. With the second phase finished the abstraction process in the log is performed,

Algorithm 3. Assign an event with an abstracted activity to each empty abstraction (Alg. 1: 14).

Input: A set $\Lambda^{\emptyset}$ of empty abstractions.
Output: The set Λ of abstractions with the events of the abstracted activities.

Algorithm `assignAbstractedEvents(`$\Lambda^{\emptyset}$`)`
  $\Lambda_I^{\emptyset} \leftarrow \emptyset$ `// set of sets of` $\lambda^{\emptyset}$ `with identical inputs`
  forall $\lambda^{\emptyset} \in \Lambda^{\emptyset}$ **do**
    if $(\lambda^{\emptyset} \notin \cup\Lambda_I^{\emptyset})$ **then**
      $\tilde{\Lambda}^{\emptyset} \leftarrow \{\tilde{\lambda}^{\emptyset} \mid \tilde{\lambda}^{\emptyset} \in \Lambda^{\emptyset} \wedge \tilde{\lambda}^{\emptyset}.A_I = \lambda^{\emptyset}.A_I\}$
      $\Lambda_I^{\emptyset} \leftarrow \Lambda_I^{\emptyset} \cup \{\tilde{\Lambda}^{\emptyset}\}$
    end
  end
  $\Lambda_O^{\emptyset} \leftarrow \emptyset$ `// set of those sets in` $\Lambda_I^{\emptyset}$ `with identical outputs`
  forall $\Lambda_i^{\emptyset} \in \Lambda_I^{\emptyset}$ **do**
    if $(\Lambda_i^{\emptyset} \not\subset \cup\Lambda_O^{\emptyset})$ **then**
      $\tilde{\Lambda}^{\emptyset} \leftarrow$ sets in $\Lambda_I^{\emptyset}$ with identical output activities than $\Lambda_i^{\emptyset}$
      $\Lambda_O^{\emptyset} \leftarrow \Lambda_O^{\emptyset} \cup \{\tilde{\Lambda}^{\emptyset}\}$
    end
  end
  $\Lambda \leftarrow \emptyset$ `// set with the abstractions with events assigned`
  forall $\Lambda_o^{\emptyset} \in \Lambda_O^{\emptyset}$ **do**
    $\alpha \leftarrow$ create new activity
    forall $\lambda^{\emptyset} \in \Lambda_o^{\emptyset}$ **do**
      $\lambda \leftarrow \lambda^{\emptyset}$ with α as activity of $\lambda.\varepsilon^{abs}$
      $\Lambda \leftarrow \Lambda \cup \{\lambda\}$
    end
  end
  return Λ

producing the traces of Fig. 4. With this abstracted log, it is possible to mine the model shown in Fig. 4c.

4 Experimentation

In this section we evaluate the performance of WoSimp. These experiments have been executed in a computer with an Intel Core i7-2600 and 16GB of RAM[1].

4.1 Datasets

For the experimentation a real log from the health domain —sepsis cases from a hospital [12]— and multiple Business Process Challenge logs [13–16] have been used. The characteristics of these logs are presented in Table 2.

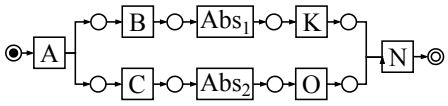
[1] The algorithm, datasets and results can be downloaded from `http://tec.citius.usc.es/processmining/WoSimp/`

$\langle A, B, Abs_1, C, Abs_2, O, K, N \rangle$

(a) Abstracted trace of Fig. 3b.

$\langle A, C, Abs_2, B, Abs_1, K, O, N \rangle$

(b) Abstracted trace of Fig. 3c.



(c) Abstracted Petri net for the process in Fig. 3.

Fig. 4. Petri net and two traces to exemplify the abstraction process.

Although the abstraction of infrequent behaviour is usually useful to visualize what is happening in the process, there are some scenarios where the penalization it causes in terms of quality metrics makes it worse than other simplification techniques. Two log features are the most relevant to describe in which scenarios the abstraction of infrequent behaviour might produce a better process model. One of these features is the number of activities. The penalization due to the inclusion of abstracted activities —not present in the log— is too high when the number of activities is low —e.g. BPIC13-clo and BPIC13-op. The other feature is the percentage of the log covered by the most frequent activity sequences — variants. In logs where few variants cover a high percentage of the log traces, the discovery of a model with those variants may already lead to a better and

Table 2. Characteristics of the logs used in the experimentation: number of traces (*#Traces*); number of events (*#Events*); number of activities (*#Activities*); number of variants —traces with the same activity sequence— (*Variants*), and the percentage of the log covered by the three variants with more traces. All the logs have been modified by adding both single start and end activities to each trace. All event names have been combined with its lifecycle to discern between different phases of the same activity (*START*, *COMPLETE*, etc.).

	#Traces	#Events	#Activities	Variants			
				#	% 1st	% 2nd	% 3rd
BPIC11	1143	152577	626	981	3.59%	1.49%	1.40%
BPIC12-financial	13087	288374	38	4366	26.20%	14.30%	2.07%
BPIC13-clo	1487	9634	9	327	32.62%	8.68%	7.40%
BPIC13-inc	7554	80641	15	2278	23.15%	6.94%	4.66%
BPIC13-op	819	3989	7	182	21.49%	15.02%	6.72%
BPIC15_1	1199	54615	400	1170	0.67%	0.50%	0.33%
BPIC15_2	832	46018	412	828	0.24%	0.24%	0.24%
BPIC15_3	1409	62499	385	1349	1.06%	0.85%	0.71%
BPIC15_4	1053	49399	358	1049	0.28%	0.19%	0.19%
BPIC15_5	1156	61395	391	1153	0.17%	0.17%	0.17%
sepsis-cases	1050	17314	18	846	3.33%	2.29%	2.10%

simpler process model. Regarding this feature, note that logs from BPIC12 and BPIC13 contain more than a third of the traces in three variants.

4.2 Results

We have compared our approach with two state of the art techniques: *Matrix Filter*[2] [5], and *Activity Filter*[3] [6]. We have also considered a naive simplification technique such as the removal of the variants with lower percentage of coverage —henceforth referred to as *Repetitions*.

We have run these techniques in each log with 9 simplification thresholds, from 10% to 90% with a step of 10. In *Repetitions* this threshold means the minimum percentage of traces covered with the most frequent variants to be maintained, in *Activity Filter* it refers to the percentage of activities of the log to be maintained[4], and in *Matrix Filter* it means the threshold to consider an event as outlier. For each simplified log, 5 process models have been discovered: one with the Inductive Miner [18], and 4 with the Inductive Miner Infrequent [19] (thresholds 10%, 20%, 30% and 40%). Finally, to check the simplification level of these techniques and how good are the process models they obtain, we have measured the fitness —Alignment-based fitness [20]—, precision —Negative Event Precision [21]— and simplicity —Weighted P/T average arc degree [22]— of each simplified model.

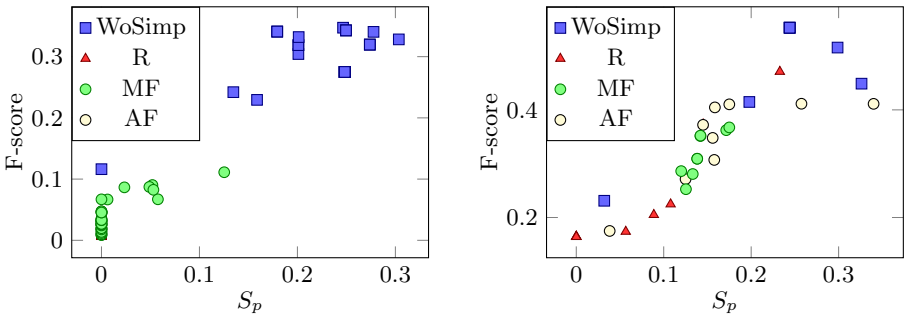
We aim to obtain a simple process model allowing to understand the frequent behaviour happening in the process while both fitness and precision are maintained at desirable levels —a model with an extremely low precision allows too many behaviour not recorded in the log, obfuscating the real behaviour. For this reason, both metrics have been summarized in the F-score, penalizing low values in any of them. Regarding simplicity, we have transformed it into a metric with values in $[0, 1]$, where a greater value is better —as the F-score. We use the percentage of simplification w.r.t. the simplicity of the discovered model with the original log ($S_p = 1 - \frac{min(S_{raw}, S_s)}{S_{raw}}$). Being S_p the percentage of simplification, S_{raw} and S_s the simplicity of the models mined with the original log, and with the simplified log, respectively.

Fig. 5 shows the F-score and S_p of the models discovered with two simplified logs as inputs. Fig. 5a shows a clear overcoming of WoSimp over the other techniques: for high values of S_p, it obtains models with higher values of F-score. However, there are cases, such as the ones depicted in Fig. 5b, where not all the models from other techniques are overcome by a model obtained with WoSimp. For this reason, to make a fairer comparison between the different techniques, we have used the area covered by the dominant points.

[2] Using plugin *Matrix Filter* in ProM with `Mean` as the `Threshold adjusting Method`.

[3] Using the plugin *Activity Filter: Indirect Entropy optimized with Greedy Search* in ProM [17].

[4] *Activity Filter* takes more than 24h to converge in datasets with more than 300 activities, thus, no results of this technique are shown in those datasets.



(a) S_p vs. F-score of the models discovered by the IM with the simplified logs of BPIC15_1.

(b) S_p vs. F-score of the models discovered by the IMf with the simplified logs of sepsis cases.

Fig. 5. Scatter plots of S_p against F-score for the models mined with the simplified logs for each technique (R stands for *Repetitions*, MF for *Matrix Filter* and AF for *Activity Filter*).

Fig. 6 shows for each dataset the area covered by the dominant points of the models obtained with both the Inductive Miner (top) and Inductive Miner Infrequent (bottom), having the logs simplified with each technique as inputs. As it was commented in Sec. 4.1, for datasets with few activities (BPIC13-op and BPIC13-clo), the addition of abstracted unmapped activities is not worth due to its penalization. Furthermore, due to the high quantity of behaviour covered by the more frequent variants, the results of WoSimp in these datasets are overcome by all the approaches, being *Repetitions* the best option. In other datasets where the number of activities is higher, but the more frequent variants still cover more than a third of the log traces (BIC13-inc and BPIC12-fin), WoSimp is only overcome by *Repetitions*. The result in BPIC11 is a particular case. Here, the 10% of more repeated traces contain enough common behaviour to compensate the penalization that WoSimp receives for adding abstracted unmapped activities. Nevertheless, this only happens using the IM, and WoSimp allows IMf to discover better process models than all other techniques.

As commented in Sec. 4.1, in datasets where the trace variability is high (BPIC15_1, BPIC15_2, BPIC15_3, BPIC15_4, BPIC15_5 and sepsis cases) and a naive technique as *Repetitions* is not useful, and WoSimp outperforms the state of the art techniques as Fig. 6 shows. Note that, if the variability in traces is high, the abstraction of WoSimp is the best option for logs with both high (BPIC15) and low (sepsis-cases) number of activities.

5 Conclusions

We have presented WoSimp, a novel algorithm to simplify process logs abstracting the infrequent behaviour, allowing to discover a simpler process model. The

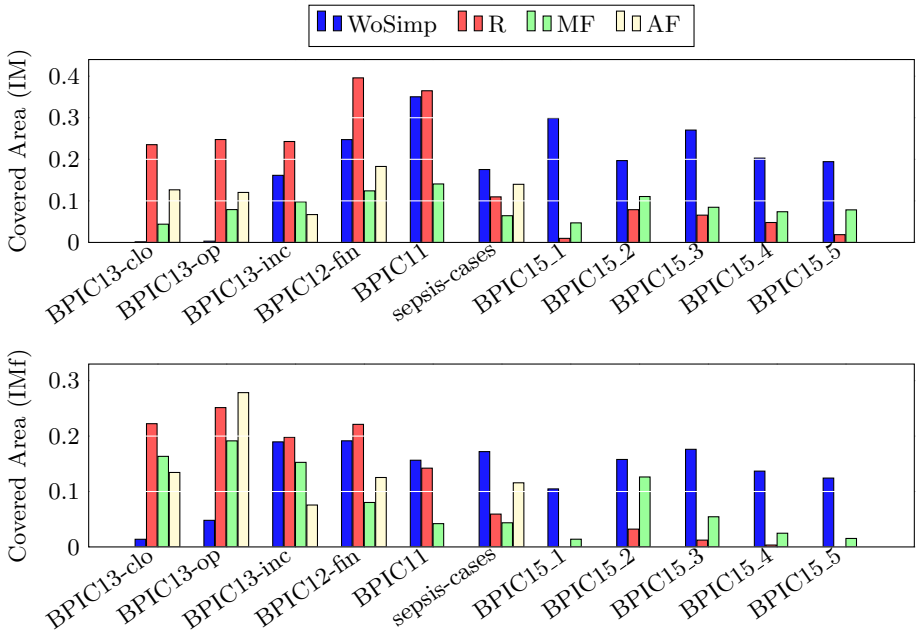


Fig. 6. Area covered by the dominant points (S_p vs. F-score) for the models discovered —IM (top), IMf (bottom)— with the simplified logs of each technique, for each dataset.

proposal is able to detect, using WoMine, the infrequent behaviour which obfuscates a process and abstract it allowing to discover a simpler and comprehensible process model. We have compared WoSimp with the state of the art approaches showing that WoSimp outperforms the state of the art in complex processes.

Acknowledgments.

This research was funded by the Spanish Ministry of Economy and Competitiveness under grant TIN2017-84796-C2-1-R, and the Galician Ministry of Education, Culture and Universities under grant ED431G/08. These grants are co-funded by the European Regional Development Fund (ERDF/FEDER program). D. Chapela-Campa is supported by the Spanish Ministry of Education, under the FPU national plan (FPU16/04428).

References

1. van der Aalst, W.M.P.: Process Mining - Data Science in Action, Second Edition. Springer (2016)
2. Polyvyanyy, A., García-Bañuelos, L., Dumas, M.: Structuring acyclic process models. Inf. Syst. **37**(6) (2012) 518–538

3. Fahland, D., van der Aalst, W.M.P.: Simplifying discovered process models in a controlled manner. Inf. Syst. **38**(4) (2013) 585–605
4. de San Pedro, J., Carmona, J., Cortadella, J.: Log-based simplification of process models. In Motahari-Nezhad, H.R., Recker, J., Weidlich, M., eds.: BPM 2015. Volume 9253 of LNCS., Springer (2015) 457–474
5. Sani, M.F., van Zelst, S.J., van der Aalst, W.M.P.: Improving process discovery results by filtering outliers using conditional behavioural probabilities. In Teniente, E., Weidlich, M., eds.: BPM 2017 International Workshops. Revised Papers. Volume 308 of LNBIP., Springer (2017) 216–229
6. Tax, N., Sidorova, N., van der Aalst, W.M.P.: Discovering more precise process models from event logs by filtering out chaotic activities. J. Intell. Inf. Syst. **52**(1) (2019) 107–139
7. Mannhardt, F., de Leoni, M., Reijers, H.A., van der Aalst, W.M.P., Toussaint, P.J.: From low-level events to activities - A pattern-based approach. In Rosa, M.L., Loos, P., Pastor, O., eds.: BPM 2016. Volume 9850 of LNCS., Springer (2016) 125–141
8. Mannhardt, F., Tax, N.: Unsupervised event abstraction using pattern abstraction and local process models. In Gulden, J., Nurcan, S., et al., eds.: BPMDS 2017. Volume 1859 of CEUR Workshop Proceedings., CEUR-WS.org (2017) 55–63
9. Tax, N., Sidorova, N., Haakma, R., van der Aalst, W.M.P.: Mining local process models. CoRR **abs/1606.06066** (2016)
10. Chapela-Campa, D., Mucientes, M., Lama, M.: Mining frequent patterns in process models. Inf. Sci. **472** (2019) 235–257
11. Desel, J., Reisig, W.: Place or transition petri nets. In Reisig, W., Rozenberg, G., eds.: Lectures on Petri Nets I: Basic Models, Advances in Petri Nets. Volume 1491 of LNCS., Springer (1996) 122–173
12. Mannhardt, F. (Felix): Sepsis cases - event log (2016)
13. Van Dongen, B.: Real-life event logs - hospital log (2011)
14. Van Dongen, B.: BPI Challenge 2012 (2012)
15. Steeman, W.: BPI Challenge 2013 (2013)
16. Van Dongen, B.: BPI Challenge 2015 (2015)
17. van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M., van der Aalst, W.M.P.: The prom framework: A new era in process mining tool support. In Ciardo, G., Darondeau, P., eds.: ICATPN 2005. Volume 3536 of LNCS., Springer (2005) 444–454
18. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs - A constructive approach. In Colom, J.M., Desel, J., eds.: PETRI NETS 2013. Volume 7927 of LNCS., Springer (2013) 311–329
19. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs containing infrequent behaviour. In Lohmann, N., Song, M., Wohed, P., eds.: BPM 2013 International Workshops. Revised Papers. Volume 171 of LNBIP., Springer (2013) 66–78
20. Adriansyah, A., van Dongen, B.F., van der Aalst, W.M.P.: Conformance checking using cost-based fitness analysis. In: EDOC 2011, IEEE Computer Society (2011) 55–64
21. vanden Broucke, S.K.L.M., Weerdt, J.D., Vanthienen, J., Baesens, B.: Determining process model precision and generalization with weighted artificial negative events. IEEE Trans. Knowl. Data Eng. **26**(8) (2014) 1877–1889
22. vanden Broucke, S.K.L.M., Weerdt, J.D., Vanthienen, J., Baesens, B.: A comprehensive benchmarking framework (cobefra) for conformance analysis between procedural process models and event logs in prom. In: IEEE Symposium on Computational Intelligence and Data Mining, CIDM 2013, IEEE (2013) 254–261