Discovering Duplicate Tasks in Transition Systems for the Simplification of Process Models

Javier De San Pedro and Jordi Cortadella

Dept. of Computer Science, Universitat Politècnica de Catalunya, Barcelona, Spain

Abstract. This work presents a set of methods to improve the understandability of process models. Traditionally, simplification methods trade off quality metrics, such as fitness or precision. Conversely, the methods proposed in this paper produce simplified models while preserving or even increasing fidelity metrics. The first problem addressed in the paper is the discovery of duplicate tasks. A new method is proposed that avoids overfitting by working on the transition system generated by the log. The method is able to discover duplicate tasks even in the presence of concurrency and choice. The second problem is the structural simplification of the model by identifying optional and repetitive tasks. The tasks are substituted by annotated events that allow the removal of silent tasks and reduce the complexity of the model. An important feature of the methods proposed in this paper is that they are independent from the actual miner used for process discovery.

1 Introduction

Many factors can reduce the usefulness of a process model. Good quality models need to find a balance between all the common metrics by which a model can be evaluated against a log: *fitness*, *precision*, *simplicity* and *generalization* [1]. For example, an open problem in process mining is finding a middle point between *overfitting* and *underfitting* models [2]. Overfitting models only allow the behavior that has been observed, and thus may trade off simplicity and generalization, while underfitting models allow for more behavior, sacrificing precision. An unnecessarily overfit model may prevent the user from distilling more insight about the behavior of the process.

This paper presents a set of techniques to explore the trade off between *simplicity* and *precision*. More specifically, by introducing a small number of new elements, the proposed techniques result in tangible improvements in precision. They can work in combination with any existing discovery (mining) algorithm. While some of the techniques can be applied to different formal models, this work will focus on Petri nets.

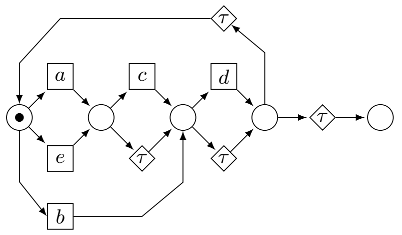
The first technique enables the discovery of duplicate tasks in process models. Duplicate tasks allow several nodes to refer to the same activity in the event log. While this is not a new concept in Process Mining [3,4,5,6], our proposal is novel in that the splitting criteria is based on properties of Labeled Transition Systems, thus allowing more precision than other existing techniques.

The second technique performs structural simplifications that do not modify the semantics of the model, thus preserving the quality metrics. We introduce extensions to the formalism that allow single nodes to represent more complex control-flow structures, such as loops or optional tasks.

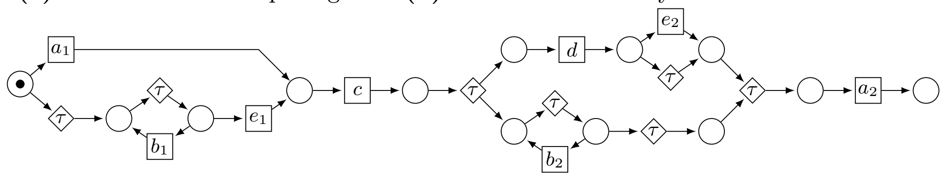
a c d e a
a c d b a
a c b d b e b a
a c b b d b b e b b a

b b b e c d e a
b b b e c d b a
b b b e c b d b e b a
b b b e c b b d b b e b b a

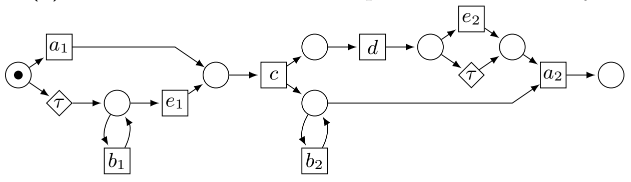
(a) Subset of the example log.



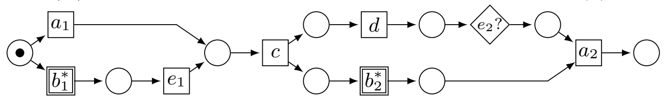
(b) Model discovered by the Inductive Miner.



(c) Model constructed after duplicate task discovery.



(d) Removal of unneeded silent transitions from (c).



(e) Using *meta-transitions* to simplify (d)

Fig. 1: Applying the method presented in this paper to a sample model discovered by the Inductive Miner.

1.1 Motivating example

Figure 1 will be used to illustrate the main contributions of this paper. We start from a simple log, a subset of which is shown in Fig. 1a. Figure 1b shows the model discovered by the Inductive Miner [7]. This model is highly imprecise (50%): while it is not a pure flower model, almost all the words are recognized.

The reason many discovery algorithms generate such a low-precision model is the presence of *duplicate tasks* in the original process. These may be introduced if, for example, different tasks have been improperly tagged with the same label.

Figure 1c shows the process model after the discovery of some duplicate tasks. The original process had two different tasks for each of the labels a, b, and e. This information is discovered automatically using the methods proposed in this work. Duplicate tasks also allow the discovery of more precise models. In this particular case, the new model has a precision of 90% and the workflow structure is clearer. However, the model has increased the total number of components, including silent transitions, which unnecessarily increase cognitive load.

Many of the silent transitions in Fig. 1c can be removed without affecting the semantics of the model, as shown in Fig. 1d. A method to remove silent transitions will also be presented in this work.

By applying some structural transformations to Fig. 1d, further reductions on the structure of the Petri net can be achieved. In this work, the alphabet of labels is enhanced to incorporate *meta-transitions*, which represent control flow patterns. For example, an $e?$ meta-transition can replace a choice between e and a silent transition, as in Fig. 1d. Similarly, a meta-transition $b*$ can sometimes replace a self-loop transition with label b. In this particular model, meta-transitions allow the removal of all silent transitions without altering its behavior.

The rest of this paper is structured as follows. Section 2 introduces the required background of this work. Section 3 describes the first proposed technique: a method to discover discover duplicate tasks. The second technique, a set of structural transformations to simplify a Petri net, is shown in Section 4. Both techniques are evaluated in Section 5. Finally, Section 6 discusses the related work, and Section 7 presents the conclusions.

2 Preliminaries

2.1 Process Mining

Let Σ be an alphabet of *events*. A *trace* is a word $\sigma \in \Sigma^*$ that represents a finite sequence of events. An *event log* $L \in \mathcal{B}(\Sigma^*)$ is a multiset of traces[1]. Event logs are the starting point to apply process mining techniques, guided towards the discovery, analysis or extension of process models. *Process discovery* is an important discipline in process mining, concerned with learning a process model (e.g., a Petri net) from a log. Several discovery techniques are summarized in [1].

Process models are usually evaluated in four quality dimensions: *replay fitness*, *simplicity*, *precision*, and *generalization* [1]. A model with perfect replay fitness can replay all the traces in the log. On the other hand, a precise model does not replay any trace other than those contained in the log.

Among the different formalisms for process models, Petri nets are perhaps the most popular, due to its well-defined semantics. This paper focuses on Petri nets, although the work may be adapted to other formalisms.

2.2 Petri Nets

A labeled Petri Net [8] is a tuple $N = \langle P, \Sigma, T, \mathcal{L}, \mathcal{F}, m_0 \rangle$, where P is the set of places, Σ is the alphabet of labels (corresponding to events), T is the set of transitions, $\mathcal{L} : T \to \Sigma \cup \{\tau\}$ assigns a label (or the empty label τ) to every transition, $\mathcal{F} : (P \times T) \cup (T \times P) \to \{0, 1\}$ is the flow relation, and m_0 is the initial marking. A marking $m : P \mapsto \mathbb{N}$ is an assignment of a non-negative integer to each place. If $m(p) = k$, we say that p is marked with k tokens. Given a node $x \in P \cup T$, its pre-set and post-set are denoted by ${}^\bullet x$ and $x^\bullet$ respectively.

[1] $\mathcal{B}(A)$ denotes the set of all multisets over A.

A transition t is *enabled* in a marking m when all places in ${}^\bullet t$ are marked. When t is enabled, it can *fire* by removing a token from each place in ${}^\bullet t$ and putting a token to each place in $t^\bullet$. A marking m' is *reachable* from m if there is a sequence of firings $t_1 t_2 \ldots t_n$ that transforms m into m', denoted by $m[t_1 t_2 \ldots t_n\rangle m'$. A sequence $t_1 t_2 \ldots t_n$ is *feasible* if it is firable from m_0. A trace σ *fits* N if there exists a feasible sequence in N with the same labels.

A transition labeled with the empty label τ is called a *silent* transition. A *duplicate* task is a transition with the same label as some other transitions in N.

2.3 Transition Systems

A finite labeled transition system is a tuple $A = (S, \Sigma, T, s_0)$ where S is a finite set of states, Σ is the alphabet of labels, $T \in S \times \Sigma \times S$ are the transition relations between states, labeled with Σ, and s_0 is the initial state.

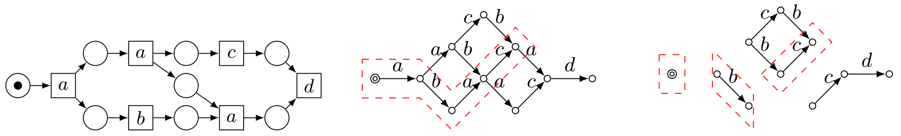
We use $s \xrightarrow{e} s'$ as a shorthand for the arc $(s, e, s') \in T$. A trace $\sigma = e_1 e_2 \ldots e_n$ *fits* A if there exists $s_1, s_2, \ldots, s_n \in S$ with $s_0 \xrightarrow{e_1} s_1 \xrightarrow{\cdots} s_{n-1} \xrightarrow{e_n} s_n$. An event $e \in \Sigma$ is enabled in a state $s_1 \in S$ if there exists $s_2 \in S$ with $s_1 \xrightarrow{e} s_2$.

Given two states s_1 and s_2 with $s_1 \xrightarrow{e} s_2 \in T$, we say e *triggers* another event f iff f is enabled in s_2, but not in s_1. In a sense, e triggering f implies a causality relation between the two events.

Excitation sets. For a given LTS $A = (S, \Sigma, T, s_0)$ and event $e \in \Sigma$, we define the *Excitation Set* of e as the set of states in which e is enabled, i.e., $ES(e) = \{s \in S \mid \exists s' \in S : s \xrightarrow{e} s'\}$.

Figure 2b shows an LTS constructed from the process in Fig. 2a. Notice how $ES(a)$ contains the states in which the three duplicate tasks of a are enabled. The concept of *local excitation set* distinguishes each such instance of a:

Definition 1 (Local excitation set). *Given LTS $A = (S, \Sigma, T, s_0)$ and event $e \in \Sigma$, the* local excitation sets *of e, $LES(e)_1, \ldots, LES(e)_k$ are the maximally connected subsets of $ES(e)$ such that, $\forall s_1 \xrightarrow{e} s_2 \in A$, if $s_1 \in LES(e)_i$ and $s_2 \in LES(e)_j$, then $i \neq j$.*



(a) Initial process. **(b)** $ES(a)$ in the LTS. **(c)** The three $LES(a)$ after removal of a-transitions.

Fig. 2: Calculation of Local Excitation Sets.

Notice that the definition does not allow both the source and target states of a transition with label e to be in the same $LES(e)_i$. The set of LES of an event can be efficiently computed with a simple algorithm, illustrated in Fig. 2c for event a. The algorithm has the following steps: (1) calculate $ES(a)$, (2) remove the transitions with label a from the LTS, (3) identify all $LES(a)$ as the maximally connected subsets of $ES(a)$ after the removal of the a-transitions.

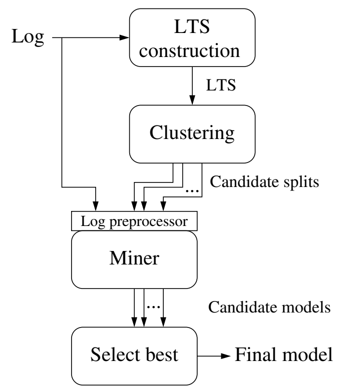
3 Discovering duplicate tasks

This section introduces a method that automatically discovers which events from an event log correspond most likely to duplicate tasks, i.e. should be represented by more than one task in order to enhance the quality of the model. The technique works with the LTS constructed from a log and can be combined with any discovery algorithm. By adding new tasks, the method slightly increases the element count of the model but results in tangible improvements in precision.

Given a log L, the goal of this procedure is to generate, for every activity $a \in L$, a *partition* of all the events in L referring to a. When mining a process model, every different partition will be represented by a different task. We will generally refer to each task by $a_1, a_2, \ldots, a_n$. A partition that, for every activity a, maps all events into a single task a_1 results in a model with no duplicate tasks. Figure 3b shows an example partition for the log in Fig. 1a.

An overview of the proposed method is shown in Fig. 3a. At the core of the proposal lies a clustering process that generates a small set of candidate partitions. An existing mining algorithm is used to generate a process model for each of these partitions, and the best model is selected out of these discovered models. This way, the method adapts to the subtleties of the different mining algorithms. Even for miners that automatically discover duplicate tasks, the proposed method may help improving the results.

The clustering method uses a bottom-up (*agglomerative*) approach: starting from the trivial partition which maps every event to a different task, the procedure iteratively selects pairs of *similar* events, grouping them into the same task. To find similar events, the algorithm uses causality relationships between events as discovered in a LTS, instead of using log information directly (e.g. direct predecessors or successors of an event). An LTS can be built from the log with a variety of methods [2]. Section 3.1 describes how the procedure finds similar events in the LTS, while Section 3.2 details the actual clustering algorithm.



(a) Overview of the flow.

```
a1 c d e2 a2
a1 c d b2 a2
a1 c b2 d b2 e2 b2 a2
a1 c b2 b2 d b2 b2 e2 b2 b2 a2

b1 b1 b1 e1 c d e2 a2
b1 b1 b1 e1 c d b2 a2
b1 b1 b1 e1 c b2 d b2 e2 b2 a2
b1 b1 b1 e1 c b2 b2 d b2 b2 e2 b2 b2 a2
```

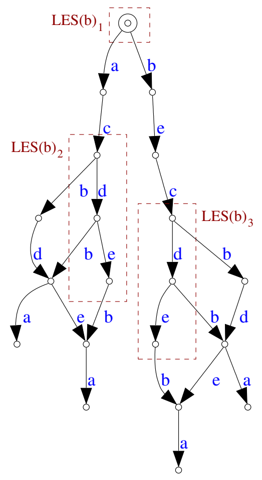
(b) Split version of the log from Fig. 1a.

Fig. 3: Summary of the duplicate task discovery process.

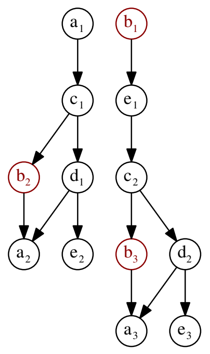
3.1 Partitioning based on Excitation Sets

A significant difference between this proposal and previous approaches to duplicate tasks is that the proposed method works at the Transition System level. The log is firstly converted into an LTS, and the clustering procedure generates a partition *based on causality relationships between excitation sets in this LTS*, rather than directly using the preceding and successor events in the log. Because of this, the approach is resilient to processes where duplicate tasks are combined with concurrency and choice. The use of clustering-based methods [9] and similarity metrics rather than looking for exact matches also allow the proposed flow to gracefully handle noise and incompleteness in the log.

Let us use an example to show the benefits of using ESs. Figure 4a shows the LTS constructed from the log in Fig. 1a, with no duplicate task detection performed. For simplicity, loops have been removed (allowing one iteration only). As per the definition of LES, there are 3 LESs for activity b, shown in Fig. 4a.



(a) Constructed LTS, highlighting all $LES(b)$.



(b) Excitation set graph of the LTS in (a).

LES	Triggers	Triggered by
a_1	$\{c\}$	$\emptyset$
a_2	$\emptyset$	$\{b,d\}$
a_3	$\emptyset$	$\{b,d\}$
b_1	$\{e\}$	$\emptyset$
b_2	$\{a\}$	$\{c\}$
b_3	$\{a\}$	$\{c\}$
c_1	$\{b,d\}$	$\{a\}$
c_2	$\{b,d\}$	$\{e\}$

(c) Trigger relations between LES.

Fig. 4: Example excitation set graph of a subset of Fig. 1a (loops removed).

Notice how the LESs of b provide an intuitive view of the correct partition for activity b (as shown in Fig. 3b): $LES(b)_1$ corresponds to the events of task b_1, while $LES(b)_2 \cup LES(b)_3$ would correspond to b_2. Our proposal classifies these LES by their relationships with other LES. The *excitation set graph* represents all the LES of a TS as well as the causality relationships between those:

Definition 2 (Excitation Set Graph). *Given a Labeled Transition System $A = (S, \Sigma, T, s_0)$, the excitation set graph of A is a graph $ESG(A)$ where:*

- *The set of vertices $V(ESG(A))$ corresponds to the set of LES of A.*
- *For every pair $(LES(a)_i, LES(b)_j)$ of A, with $a, b \in \Sigma$, there is an edge $(LES(a)_i, LES(b)_j) \in E(ESG(A))$ iff for any $s_1 \in LES(a)_i$ and $s_2 \in LES(b)_j$, $s_1 \xrightarrow{a} s_2$ triggers b.*

Figure 4b shows the corresponding excitation set graph of the example LTS, while 4c summarizes the immediate trigger relations. Notice how this information allows us to trivially distinguish between $\mathit{LES}(b)_1$ and $\{\mathit{LES}(b)_2, \mathit{LES}(b)_3\}$, since $\mathit{LES}(b)_1$ triggers a different set of events.

Compare this to using predecessor and successor information from the log directly, without constructing an LTS first. It is difficult to distinguish events of b by looking at the immediately following event. For example, an event b followed by e may indicate an instance of task b_1 as discovered in the previous section, but it may also be caused by an instance of b_2, since it is concurrent with e. Thus, using log information only, it would be difficult to construct an accurate partition for b. The use of excitation sets avoids this problem.

Even when using excitation sets, the combination of choice, loops and/or incomplete logs may introduce LES that have related but slightly different sets of predecessors/successors, yet should be mapped to the same task. For this reason, the proposed flow includes a clustering method that combines *similar* LES. This method is described in the following section.

3.2 Hierarchical clustering algorithm

This section describes the method used by our proposal to classify local excitation sets into groups with similar causality relationships. The described clustering method is *agglomerative* [9], discovering clusters using a bottom up approach: the algorithm starts by assuming every that, for every activity a, every $\mathit{LES}(a)_i$ belongs to its own cluster. In this initial solution, each LES maps to its own duplicate task. Then, the algorithm considers the pairwise similarity of all the LES, and combines the two closest $(\mathit{LES}(a)_i, \mathit{LES}(a)_j)$ (of the same activity a) into the same task a_i. The entire process iterates until no further clustering can be performed. On every iteration, the algorithm explores a solution with exactly one duplicate task less than the previous solution.

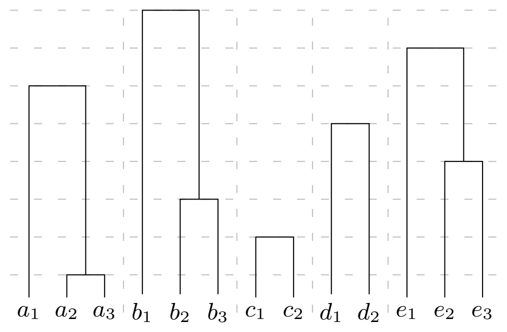
Algorithm 1 Discovery flow with duplicate tasks

```
 1: function DuplicateTaskDiscovery($L, M$)
    ▷ $L$ is the input log, $M$ is a miner algorithm.
 2:     $A \leftarrow$ ConstructLTS($L$)
 3:     $G \leftarrow$ ESG($A$)
 4:     $R \leftarrow M(L_i)$          ▷ Stores the best result (process model) discovered so far
 5:     while $|V(G)| > |\mathit{Activities}(A)|$ do        ▷ While there is some duplicate task
 6:         $v_i, v_j \leftarrow$ FindMostSimilarNodes(G)
 7:         merge $v_i, v_j$ into single node in $G$
 8:         $L_i \leftarrow$ TagLog($L, G$) ▷ Tag events in the log according to current partition
 9:         $N_i \leftarrow M(L_i)$                    ▷ Discover a temporary model for evaluation
10:         if $N_i$ is better than $R$ then
11:             $R \leftarrow N_i$
12:     return $R$
```

The full discovery algorithm can be seen in Algorithm 1. The input is a log L. A is an LTS constructed from L (for example using the methods described in [2]), while G is the initial ESG, constructed using the rules seen in Definition 2. The output R is a process model with duplicate tasks.

In every iteration, procedure FINDMOSTSIMILARNODES selects two vertices of G with the most similar *context vectors*, a numeric way to represent their causality relations which will be explained in the following section. The selected vertices are then merged into a single new vertex, representing the new cluster, which inherits the causality relationships of the merged vertices. Note that only vertices with the same activity label will be selected. The loop ends when there is only vertex in G for every activity, i.e. there are no duplicate tasks.

To select which partition of tasks will be returned by our procedure, we construct a temporary process model N_i at every iteration. The provided miner is called using a log where events have been tagged according to the currently evaluated partition. The details of how models are compared will be described in a later section. Note that the total maximum number of models to evaluate (i.e. the number of iterations in the procedure) is limited by the number of excitation sets in the LTS. However, most processes contain only a few duplicate tasks. Limiting to 4 or 5 tasks per activity reduces the number of models that need to be evaluated to a few, depending on the number of different activities.



a_1 a_2 a_3 b_1 b_2 b_3 c_1 c_2 d_1 d_2 e_1 e_2 e_3

Dup. tasks	$\lvert P\rvert + \lvert T\rvert$	Precision	
0	17	0.53	
1	24	0.60	
2	25	0.65	
3	**22**	**0.95**	←
4	23	0.86	
5	26	0.88	
6	33	0.88	
7	36	0.95	
8	38	0.95	

Fig. 5: Dendogram showing clustering of LTS in Fig. 4a.

Figure 5 visualizes the clustering procedure. The initial solution, where every LES is partitioned into its own duplicate task, is shown at the bottom row. The following row represents one iteration of the clustering process, in which a_2, a_3 were the most similar LES and were merged. Thus, the number of duplicate tasks, in the first column, is reduced by 1. The top row shows the result after all nodes have been merged and thus there are no duplicate tasks left. Columns 2 and 3 show sample metrics of the evaluation model for each row: Petri net size and precision. The selected model has the best precision and smallest size.

Representing excitation set relations in vector space. In order to find the closest two groups of LES, a distance metric capable of evaluating the similarity of the relationships of two LES is required. For this, we will first provide a way to represent, as a numeric vector, the causality relationships of a given vertex (representing a LES or cluster of LES) in a ESG.

This representation needs to satisfy several requirements: a) it needs to be normalized, allowing meaningful comparisons between different vertices, b) it needs to distinguish vertices by their immediate predecessors/successors, but also more distant neighbors. Otherwise, duplicate tasks sharing the same set of immediate predecessors or successors would not be distinguishable. However, similarity of closer neighbors should have more weight than distant neighbors.

Definition 3 (Context vector). *Given LTS A, $ESG(A)$, and a vertex $v \in ESG(A)$, the* forward context vector *of v, $\overrightarrow{C_v}$, is a function $E \mapsto \mathbb{R}$ that maps an activity $e \in \Sigma$ to*

$$\overrightarrow{C_v}(e) = \frac{|\operatorname{Succ}(v,e)|}{2|\operatorname{Succ}(v)|} + \frac{\sum_{v' \in Succ(v)} \overrightarrow{C_{v'}}(e)}{4|Succ(v)|}$$

where $Succ(v)$ is the set of immediate successors of v and $Succ(v,e)$ is the set of immediate successors of v of activities with label e. Similarly, we can define the backwards context vector*, $\overleftarrow{C_v}$, using predecessors instead of successors.*

For a given vertex v and event e, the value of $\overrightarrow{C_v}(e)$ depends on the number of e-successors of v relative to the total number of successors of v. Notice the function gives decreasing weight to more distant successors using the pattern $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots$. Thus, the function is normalized between $[0 \ldots 1)$, allowing for numeric comparisons between different vectors.

Imposing a limit k to the recursion depth, context vectors are easy to compute with a single pass over the graph. As the weight of successors decreases with distance, this limit does not impact the quality of the metric. An example list of context vectors for the graph in Fig. 4b is shown in Table 1, assuming $k = 2$.

Table 1: Context vectors for the ESG in Fig. 4b.

	Forward					Backward				
LES	a	b	c	d	e	a	b	c	d	e
a_1	0	1/8	1/2	1/8	0	0	0	0	0	0
a_2	0	0	0	0	0	0	1/4	0	1/8 + 1/8	1/4
a_3	0	0	0	0	0	0	1/4	0	1/8 + 1/8	1/4
b_1	0	0	1/4	0	1/2	0	0	0	0	0
b_2	1/2	0	0	0	0	1/4	0	1/2	0	0
b_3	1/2	0	0	0	0	0	0	1/2	0	1/4
c_1	1/16 + 2/16	1/4	0	1/4	1/16	1/2	0	0	0	0
c_2	1/16 + 2/16	1/4	0	1/4	1/16	0	1/4	0	0	1/2

Distance function. To measure the similarity (distance) between two vertices $v_1, v_2 \in ESG(A)$, the following formula is used, where d is the Euclidean distance:

$$\operatorname{dist}(v_1, v_2) = \min(\operatorname{d}(\overrightarrow{C_{v_1}}, \overrightarrow{C_{v_2}}), \operatorname{d}(\overleftarrow{C_{v_1}}, \overleftarrow{C_{v_2}}))$$

Using the minimal distance between the forward and backward vectors allows proper detection of duplicate tasks in the first and last iterations of loops. For tasks in a loop, several LESs may exist in the LTS for different iterations of the same task. The causality relations of the LESs corresponding to the first and last iterations will be different of those from inner iterations. For example, only the LES corresponding to the last iteration will not trigger other LESs of the same task. By centering on either the backward or forward context vector, depending on which pair is the closest, these LESs will still be clustered into a single task.

Comparing candidate models. Traditional hierarchical clustering algorithms use various criteria to determine which clustering solution is more suited to the data, such as for example the *elbow* criteria [10]. However, the flow proposed in this work produces more than one candidate model, allowing the exploration of the trade-off between precision and simplicity. By limiting the maximum number of allowed duplicate tasks, the set of candidate models can be kept under manageable sizes. Therefore, conventional conformance checking strategies may be used to accurately compare the candidate models, e.g. measuring fitness, precision, generalization or simplicity. Generally, a combination of these parameters will be used, depending on user preference. For example, maximizing precision while constraining the simplicity to a minimum threshold value.

Figure 5 shows that the precision increases with every duplicate task until 95% with 3 duplicate tasks, and then decreases, revealing that more duplicate tasks introduce unnecessary choices and are not necessary for this process. The result, with 3 duplicate tasks, exactly matches the model shown in Fig. 1c.

4 Structural simplification

This section introduces the structural simplifications proposed in this work: substituting common control flow patterns with special *meta-tasks* that represent optional or iterative behavior.

The simplifications are especially suitable for Petri nets. They reduce the complexity of the net while still allowing the expressiveness of Petri nets. In addition, the proposed simplifications exactly preserve the semantics of the models, and thus, conformance metrics such as fitness and precision.

The simplifications center on two aspects. First, the removal of unnecessary silent transitions. While silent transitions are a useful construct, many mining algorithms or conversions from other modeling languages often generate silent transitions that may be unnecessary [11]. Second, we introduce a series of *meta-transitions* which extend the language of Petri nets and represent simple flow control operations such as optional or iterative behavior.

Removal of silent transitions. Our proposal removes unnecessary silent transitions by following the transformations shown in Fig. 6. The objective of these transformations is to eliminate as many silent transitions as possible without impacting the semantics of the Petri net, so that the set of traces fitting the original net is identical to the traces fitting the transformed Petri net. The transformations proposed are similar to the liveness and safeness-preserving transformations

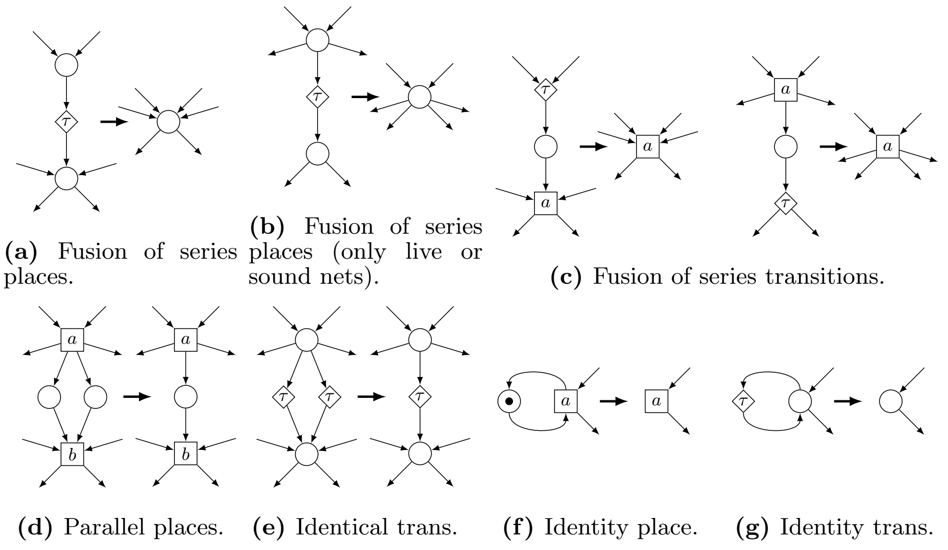

(a) Fusion of series places. (b) Fusion of series places (only live or sound nets). (c) Fusion of series transitions. (d) Parallel places. (e) Identical trans. (f) Identity place. (g) Identity trans.

Fig. 6: Reduction rules for behavior-preserving removal of silent transitions.

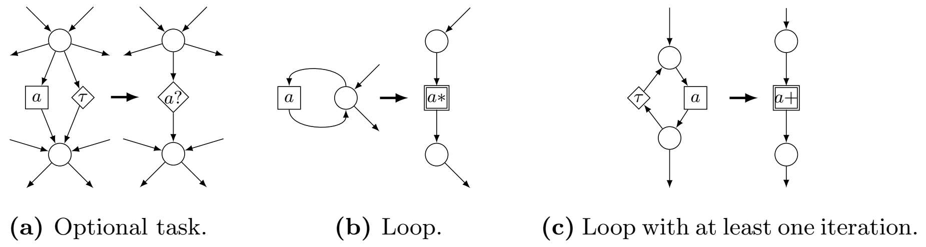

(a) Optional task. (b) Loop. (c) Loop with at least one iteration.

Fig. 7: Rules for transformation using *meta-transitions*.

proposed in [8], that have been already used in previous work [11,12] also with the goal of removing silent transitions. However, the existing set of transformations is not exhaustive. For example, it is not possible to remove all the silent transitions from the model in Fig. 1c using only the rules defined in [8].

By centering on a commonly used structural type of Petri nets, sound workflow nets [13], we are able to introduce additional transformations covering the removal of more silent transitions. For example, Fig. 6b proposes that fusion of serial places can be performed even if the first place has other outgoing arcs. However, this transformation does not fully preserve the behavior of general Petri nets, as it may remove deadlocks present in the original Petri net. Full preservation of behavior, including liveness, is only guaranteed in the case of live Petri nets or nets with deadlocks only on specific states, such as sound workflow nets. For the former subtype of Petri nets, deadlocks only appear in states where the output sink place is marked [13], and the output place will never be modified by the transformation rule.

Meta-transitions. A *meta-transition* replaces common a Petri net substructure (e.g., a self-loop) with a single transition that is defined to have identical behavior. By transforming a Petri net, replacing instances of these structures by meta-transitions, the element count of a Petri net can be reduced while completely preserving its behavior. The transformed net will fit exactly the same traces as the original net. In addition, the transformation may open the door to further simplifications such as removal of additional silent transitions.

In Figure 7 we show the proposed new meta-transitions, as well as the behavior represented by each meta-transition. These specific patterns have been selected because of their high frequency in real-life processes. In addition, the well-known regular expression-like syntax used in the meta-transitions makes their meaning familiar.

The first meta-transition, $a?$, models an *optional* event: it is equivalent to a choice between the empty label τ and trace a. The other two meta-transitions represent iterative behavior. $a*$ is equivalent to a self-loop. Thus, it fits the empty trace, but also $\{a, aa, aaa, \ldots\}$. Meta-transition $a+$ similarly represents a loop of a, but requires at least one iteration.

5 Experimental evaluation

The algorithms described in this work have been implemented using PMLAB [14]. To construct an LTS from the input log, the *multiset* abstraction from [2] is used. Our implementation of the clustering procedure uses the centroid linkage functionality of [10] to avoid recomputing context vectors on every iteration.

For a set of benchmarks, we compare the quality metrics of the models obtained with and without the proposed duplicate task discovery algorithm, as well as the reduction in complexity after the structural simplifications and use of meta-transitions. All benchmarks are available at `http://www.cs.upc.edu/~jspedro/pnsimpl/`. In order to demonstrate the ability of the proposal to work with multiple miners, two different miners will be used: Inductive Miner [7] (IM) and Petrify [15]. While the current version of the Inductive Miner does not support duplicate tasks, Petrify contains some support for automatic discovery of duplicate tasks [4]. Thus, models discovered by Petrify may already contain duplicate task before the clustering method proposed in this article takes place.

Precision and generalization are measured using the available ProM plugins [16,17]. To measure complexity, we will show the size of the Petri nets. For non-workflow Petri nets, such as those generated by Petrify, we will also use a complexity metric closely related to the concept of planarity: the minimal number of *crossings* required to embed the graph on a plane. This number is estimated using GraphViz [18].

The method used to select a model from the list of candidates produced by the duplicate task discovery method depends on the miner used. When using the IM, the smallest model (in terms of places and transitions) out of all models with highest precision will be selected. When using Petrify, the model with lowest number of crossings, out of those with highest precision, is used instead.

Artificial benchmarks. To evaluate our duplicate task discovery workflow and compare to the results presented by previous work, we reuse an existing dataset comprising a combination of logs [19,6,5] whose source processes are well-known and reproduce behavior commonly found in real-life. Because these benchmarks have no noise, the miners were configured to generate perfectly fitting models.

Table 2: Comparison using artificial benchmarks.

	Inductive Miner					With duplicate tasks						After simpl.		
	$\|P\|$	$\|T\|$	$\|\tau\|$	Prec.	Gen.	$\|P\|$	$\|T\|$	$\|\tau\|$	Prec.	Gen.		$\|P\|$	$\|T\|$	$\|\tau\|$
alpha	11	17	6	68%	100%	11	16	4	70%	100%	†	9	13	1
betaSimpl	14	21	8	62%	86%	14	16	1	94%	73%		14	15	0
Fig5p19	9	14	6	67%	89%	12	14	5	85%	76%		12	12	3
Fig5p1AND	9	8	3	83%	28%	10	8	2	100%	0%		9	7	1
Fig5p1OR	5	6	1	70%	33%	6	6	0	100%	0%		6	6	0
Fig6p10	15	24	13	63%	100%	19	25	10	77%	100%		18	19	4
Fig6p25	22	35	14	76%	100%	24	35	12	84%	100%		23	27	4
Fig6p31	6	10	1	63%	72%	9	11	0	100%	42%		9	11	0
Fig6p33	7	11	1	67%	70%	10	12	0	100%	38%		10	12	0
Fig6p34	17	24	12	58%	100%	19	20	4	93%	100%		17	18	2
Fig6p38	13	11	4	62%	84%	12	14	6	66%	87%		11	11	3
Fig6p39	12	12	5	90%	94%	12	12	5	90%	94%	†	10	9	2
Fig6p42	7	18	4	23%	100%	26	32	12	75%	96%	†	24	29	9
Fig6p9	10	15	8	67%	82%	9	12	3	83%	72%		9	9	0
flightCar	10	14	4	67%	64%	10	14	4	67%	64%	†	11	9	1
RelProc	21	28	12	71%	100%	21	28	11	74%	100%	†	19	21	4

	Petrify					With duplicate tasks						After simpl.		
	$\|P\|$	$\|T\|$	Cros.	Prec.	Gen.	$\|P\|$	$\|T\|$	Cros.	Prec.	Gen.		$\|P\|$	$\|T\|$	Cros.
alpha	13	11	11	92%	100%	12	12	1	92%	100%	†	12	12	1
betaSimpl	11	13	1	80%	77%	14	15	0	97%	39%		15	15	0
Fig5p19	8	8	2	100%	74%	9	9	1	100%	58%		9	9	1
Fig5p1AND	8	5	0	100%	0%	7	6	0	100%	0%		7	6	0
Fig5p1OR	5	5	3	100%	0%	5	6	0	100%	0%		5	6	0
Fig6p10	7	11	1	39%	100%	13	15	1	91%	100%		13	15	1
Fig6p25	14	21	6	80%	100%	14	23	0	80%	100%		18	23	0
Fig6p31	7	9	12	100%	42%	8	11	0	100%	42%		8	11	0
Fig6p33	8	10	7	100%	38%	9	12	0	100%	38%		9	12	0
Fig6p34	9	12	4	39%	100%	14	16	0	89%	100%		14	16	0
Fig6p38	8	7	0	71%	85%	10	8	0	100%	64%		10	8	0
Fig6p39	6	7	0	72%	98%	7	8	1	86%	86%	†	8	8	0
Fig6p42	11	14	20	37%	98%	21	23	3	96%	94%	†	21	23	3
Fig6p9	9	7	9	100%	54%	8	9	0	100%	54%		8	9	0
flightCar	6	8	0	58%	72%	6	8	0	58%	72%	†	7	8	0
RelProc	16	16	11	87%	100%	15	17	2	87%	100%	†	15	17	2

Table 2 summarizes the results. For every benchmark, there are three different runs: in the first one, the log is mined with the default miner configuration. In the second run, the flow with duplicate task discovery as presented in this work is used. In the third result, we apply structural simplifications (silent transition elimination and meta-transitions) on top of the model discovered on the second run. For each run, we evaluate the size of the model (number of places, transitions and silent (τ) transitions) as well as its precision and generalization.

The proposed method significantly increases the precision on all the benchmarks. In some examples, generalization is reduced, yet still shows that the method results in models that are not overfitting. In tests with the Inductive Miner, using duplicate tasks allows removing most of the silent transitions, and thus the overall complexity of the model decreases. Using meta-transitions, additional silent tasks can be removed. On the other hand, when combining our discovery flow with Petrify, the discovery of duplicate tasks allows for models with fewer crossings. However, results after simplification are not as remarkable as with the IM, since Petrify does not discover silent transitions.

For the majority of benchmarks, the partition of tasks discovered by the proposed flow exactly matched the duplicate tasks in the original process. The exceptions are marked with †. These cases are usually situations where, e.g., duplicate tasks are concurrent with themselves. Despite the fact that the partition is not exactly correct, the increase in quality metrics is still significant.

Logs with noise. An additional experiment shows the resilience of the proposed method to noise. We used Process Log Generator (PLG) [20] to generate a set of 3 random processes using a process depth of 3 and uniform probabilities for all control flow operators. Then, for each of these processes, we generated 10 logs containing 1000 traces each. In each log a different amount of random control-flow noise was injected using PLG, ranging from 0% to 10%.

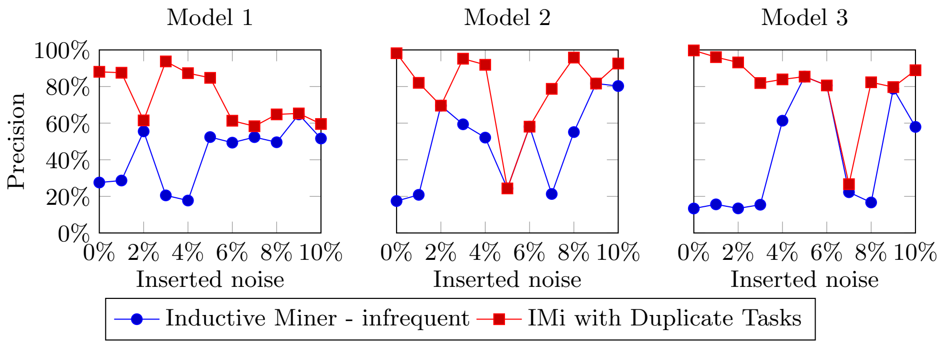


Fig. 8: Resilience of duplicate task discovery to different artificial noise levels.

Figure 8 compares the precision of the models obtained using the Inductive Miner – infrequent [7] (IMi) miner, configured with a 20% threshold, with the models obtained by the combination of our duplicate task discovery flow and the IMi. For the 3 evaluated processes, our flow can discover duplicate tasks and thus increase the precision even when confronted with noise. The differences in fitness were always smaller than 5% between both versions.

On a Intel Core i5-2520m, our implementation of the clustering procedure is able to provide a set of candidate partitions in less than 4 seconds, even for the largest of these logs. The runtime of the miner, required to evaluate each candidate, is usually much larger than the clustering process. However, the number of candidates to be evaluated can be limited by setting an upper bound to the number of allowed duplicate tasks per label.

6 Related work

Several methods already exist for duplicate task detection. In [6], a set of heuristics creates a candidate set of duplicate tasks, which is then explored by a local search procedure working in tandem with an arbitrary mining algorithm.

The method produces high-quality results in combination with advanced miners. However, since the miner influences the direction of the search, it is difficult to predict the runtime of the discovery process. In this work, the miner algorithm is only used to evaluate the set of candidate results. The number of results is exactly bounded by the maximum number of allowed duplicate tasks per event. The work in [5] proposes a clustering approach based on the context of events similar to the one described in this work. Analogously, finding repeating patterns in the log [21] may be used to discover potential duplicate tasks. However, our work uses excitation sets to identify the context of events, which allows for more accurate detection that using the log directly.

A different family of methods to perform duplicate task detection are tied to specific mining technologies. For example, Fodina [22], Genetic Miner [3], AGNEs [23], InWoLvE [19], region theory [4], α^*-algorithm [24]. The proposal in this work works with any mining algorithm, and does not require e.g. workflow nets or other specific process models.

For the second proposal in this work, structural simplifications, a potential comparable work is the use of other process modeling notations, such as BPMN [1]. The formalisms presented in this paper still allow the expressiveness of Petri nets, yet hide the complexity of common flow control operators. Other methods to simplify Petri nets do so at the cost of accuracy [25,26].

7 Conclusions

This work has presented methods for simplification of process models that improve the quality of discovered models, in both simplicity and precision, while using different mining algorithms.

As future work, we envision methods that work even in the presence of concurrent duplicate tasks, which are currently handled with unsatisfactory results. In addition, the language of structural tasks can be extended, for example, to allow simple regular expressions in nodes, e.g., $(a|bc)*$.

Acknowledgments. This work has been partially supported by funds from the Spanish Ministry for Economy and Competitiveness and the European Union (FEDER funds) under grant TIN2013-46181-C2-1-R, and the Generalitat de Catalunya (2014 SGR 1034 and FI-DGR 2015).

References

1. van der Aalst, W.M.P.: Process Mining - Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
2. van der Aalst, W., Rubin, V., Verbeek, H., van Dongen, B., Kindler, E., Gnther, C.: Process mining: a two-step approach to balance between underfitting and overfitting. Software & Systems Modeling **9**(1) (2010) 87–111
3. de Medeiros, A.K.A.: Genetic Process Mining. PhD, Technische Universiteit Eindhoven, Eindhoven, The Netherlands (2006)

4. Carmona, J.: The Label Splitting problem. In: Transactions on Petri Nets and Other Models of Concurrency VI. Volume 7400 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 1–23
5. Song, J.L., Luo, T.J., Chen, S., Liu, W.: A clustering based method to solve duplicate tasks problem. Journal of University of Chinese Academy of Sciences **26**(1) (2009) 107
6. Vázquez-Barreiros, B., Mucientes, M., Lama, M.: Mining duplicate tasks from discovered processes. In: Proc. of Algorithms & Theories for the Analysis of Event Data. Volume 1371., Brussels, Belgium, CEUR (June 2015) 78–82
7. Leemans, S., Fahland, D., van der Aalst, W.: Discovering block-structured process models from incomplete event logs. In: Application and Theory of Petri Nets and Concurrency. Volume 8489 of LNCS. Springer (2014) 91–110
8. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4) (April 1989) 541–574
9. Johnson, S.C.: Hierarchical clustering schemes. Psychometrika **32**(3) (September 1967) 241–254
10. Jones, E., Oliphant, T., Peterson, P., et al.: SciPy: Open source scientific tools for Python (2001–) [Online; accessed 2016-03-18].
11. van der Aalst, W.M.P., Dumas, M., Ouyang, C., Rozinat, A., Verbeek, E.: Conformance checking of service behavior. ACM Trans. Internet Technol. **8**(3) (May 2008) 13:1–13:30
12. Dongen, B.F., Medeiros, A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M., Aalst, W.M.P.: The ProM Framework: A new era in process mining tool support. In: Proceedings of the Applications and Theory of Petri Nets 2005 26th International Conference, Berlin, Heidelberg, Springer Berlin Heidelberg (June 2005) 444–454
13. van der Aalst, W.M.P., van Hee, K.M., ter Hofstede, A.H.M., Sidorova, N., Verbeek, H.M.W., Voorhoeve, M., Wynn, M.T.: Soundness of workflow nets: classification, decidability, and analysis. Formal Aspects of Computing **23**(3) (2011) 333–363
14. Carmona, J., Sol, M.: PMLAB: An scripting environment for Process Mining. In: Proceedings of the BPM Demo Sessions 2014. (October 2014) 16–21
15. Carmona, J., Cortadella, J., Kishinevsky, M.: A region-based algorithm for discovering Petri nets from event logs. In: Business Process Management. Volume 5240 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2008) 358–373
16. Adriansyah, A., Munoz-Gama, J., Carmona, J., van Dongen, B., van der Aalst, W.: Measuring precision of modeled behavior. Information Systems and e-Business Management **13**(1) (2015) 37–67
17. Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: On the role of Fitness, Precision, Generalization and Simplicity in Process Discovery. In: On the Move to Meaningful Internet Systems, Rome, Italy, Springer (2012) 305–322
18. Gansner, E.R., Koutsofios, E., North, S.C., Vo, K.: A technique for drawing directed graphs. IEEE Trans. Software Eng. **19**(3) (1993) 214–230
19. Herbst, J., Karagiannis, D.: Workflow mining with InWoLvE. Computers in Industry **53**(3) (2004) 245 – 264 Process / Workflow Mining.
20. Burattin, A., Sperduti, A.: PLG: A framework for the generation of business process models and their execution logs. In: Business Process Management Workshops. Volume 66 of LNBIP., Hoboken, NJ, USA, Springer (September 2010) 214–219
21. Bose, R.P.J.C.: Process mining in the large: preprocessing, discovery, and diagnostics. PhD thesis, Technische Universiteit Eindhoven (2012)
22. vanden Broucke, S.K.L.M.: Advances in Process Mining. Ph.D., Katholieke Universiteit Leuven (2014)
23. Goedertier, S., Martens, D., Vanthienen, J., Baesens, B.: Robust Process Discovery with Artificial Negative Events. J. Mach. Learn. Res. **10** (June 2009) 1305–1340
24. Li, J., Liu, D., Yang, B.: Process mining: Extending α-algorithm to mine duplicate tasks in process logs. In: Advances in Web and Network Technologies, and Information Management, Huang Shan, China, Springer (June 2007) 396–407
25. De San Pedro, J., Carmona, J., Cortadella, J.: Log-Based Simplification of Process Models. In: Business Process Management. Volume 9253 of Lecture Notes in Computer Science., Springer International Publishing (2015) 457–474
26. Fahland, D., van der Aalst, W.M.P.: Simplifying discovered process models in a controlled manner. Information Systems **38**(4) (2013) 585–605