Divide and Conquer

H.M.W. Verbeek*, W.M.P. van der Aalst*, J. Munoz-Gama†
*Department of Mathematics and Computer Science
Eindhoven University of Technology, Eindhoven, The Netherlands
E-mail: {h.m.w.verbeek,w.m.p.v.d.aalst}@tue.nl
†Department of Computer Science
Pontificia Universidad Católica de Chile, Santiago de Chile, Chile
E-mail: jmun@ing.puc.cl

***Abstract*—Process mining has been around for more than a decade now, and in that period several discovery and replay algorithms have been introduced that work fairly well on average-sized event logs. Nevertheless, these algorithms have problems dealing with big event logs. If the algorithms do not run out of memory, they will run out of time, because the problem handed to them is just too complex to be solved in reasonable time and space. For this reason, a generic approach has been developed which allows such big problems to be decomposed into a series of smaller (say, average-sized) problems. This approach offers formal guarantees for the results obtained by it, and makes existing algorithms also tractable for larger logs. As a result, discovery and replay problems may become feasible, or may become easier to handle. This paper introduces a tool framework, called *Divide and Conquer* that fully supports this generic approach, which has been implemented in ProM 6. Using this novel framework, this paper demonstrates that significant speed-ups can be achieved, both for discovery and for replay. This paper also shows that a *maximal* decomposition (that is, a decomposition into as many subproblems as possible) is not always preferable: Non-maximal decompositions may run as fast, and generally provide better results.**

I. INTRODUCTION

THE ultimate goal of *process mining* [1] is to gain process-related insights based on *event logs* created by a wide variety of systems. An event log then contains a sequence of *events* for every case that was handled by the system. As an example, Table I shows data related to a typical event recorded for some system, which can be interpreted as follows:

On October 1st, 2015, resource 112 has completed activity a_1.

A sequence of events contained in an event log is commonly referred to as a *trace*. From the data associated with the trace, we can derive for which particular case activity a_1 was completed.

TABLE I
EVENT e_1.

Key	Value
concept:name	a_1
lifecycle:transition	complete
org:resource	112
time:timestamp	2015-10-01T00:38:44.546

Typically, research done in the process mining area can be divided into three subfields: *process discovery*, *process conformance*, and *process enhancement*.

The field of *process discovery* [1], [4]–[7], [12]–[14], [19], [20], [32], [34], [35] deals with discovering a process model from an event log. Example process discovery algorithms include the Alpha Miner [5] and the ILP Miner [35]. The former was the first process discovery algorithm to discover concurrency adequately. The latter basically converts the discovery problem into many ILP (Integer Linear Problem) problems and solves the discovery problem by solving all these ILPs.

The field of *process conformance* [1], [3], [9], [11], [15], [16], [20], [26], [28], [31] deals with checking to what extent a process model and an event log conform to each other, that is, how well they match. One way to do this is to replay the event log on the process model as best as possible, which result in an *alignment* between both. Such an alignment relates events in the event log to activities in the process model. Based on this alignment, conclusions can then be drawn on important metrics like *fitness* (how well does the event log conform to the process model?), *precision* (how well does the process model conform to the event log?), and *generalization* (how well does the process model conform to the system?). An example replay algorithm is the cost-based replayer [3], which uses many ILPs to find a *cost-optimal* alignment between the event log and the process model.

The field of *process enhancement* [1], [24], [30] deals with enhancing the process model with data present in the event log. A typical example for this is adding time-related data to the process model, which allows us to detect for example bottlenecks in the process model. Note that for process enhancement an alignment is required, to position data from some event to an activity in the process model.

As indicated, both process discovery and process replay may use many ILPs to solve the problem at hand. The size of these ILPs is mainly determined by the number of different activities present in the event log, and much less by the number of traces in the event log. For example, consider the *57/52/n* event log from the *IS 2014* data set [28]. This log contains 2000 traces, 57 activities, an average trace length of 52, and noise. For this event log, we have created 9 smaller event logs by repeatedly filtering out the last 200 *traces*. Figure 1 shows the typical computation times needed by the ILP Miner on these logs as implemented in the leading process mining

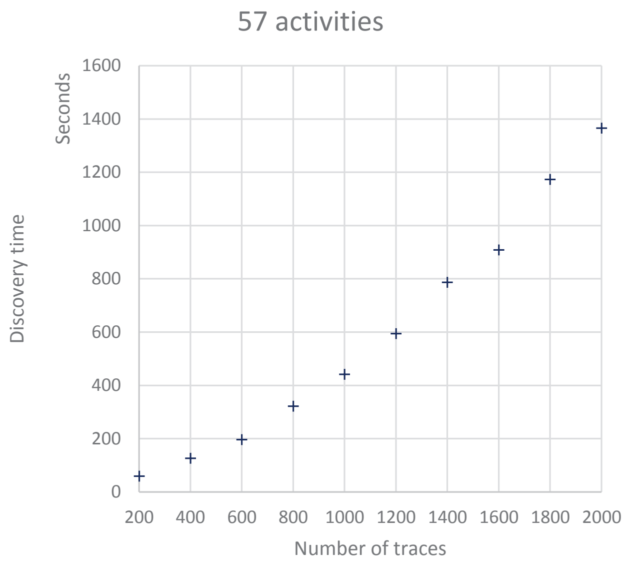


Fig. 1. The effect of different number of traces using the *ILP miner*.

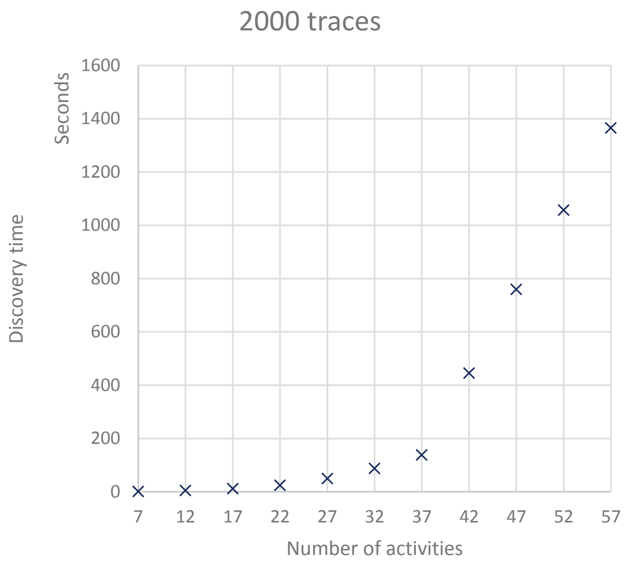


Fig. 2. The effect of different number of activities using the *ILP miner*.

framework ProM 6 [33]. The figure shows that splitting the log in this way does not really help in speeding up the ILP Miner. Granted, a sublog containing only 200 traces requires much less time than the overall log containing 200 traces, but if we have to run the ILP Miner on 10 such sublogs and then merge the results[1], we do not gain much. For the same event log, we also have created 10 smaller event logs by repeatedly filtering out 5 random *activities*[2] Figure 2 shows the typical computation times needed by the same ILP Miner on these logs. The figure shows that if we would be able to split the event log into five sublogs each containing 17 activities, the ILP Miner might only need 60 seconds instead of almost 1400.

To be able to deal with big event logs containing many different activities, [2] has proposed a *decomposition* approach for both process discovery and process replay. Instead of discovering a process model from the *overall* event log (the *monolithic* approach), the event log is first decomposed into a number of sublogs that each contains only a subset of the activities from the overall event log, second the discovery algorithm is employed on each of these sublogs resulting in as many process fragments, third all fragments are merged into an overall process model (the *decomposition* approach). This decomposition approach may be significantly faster than the monolithic approach. Likewise, instead of replaying the *overall* event log on the *overall* process model (the *monolithic* approach), the log and model can first be decomposed into sublogs and submodels, second the replayer can be employed on every sublog with corresponding submodel, third the resulting subalignments can be merged into an overall *pseudo-alignment* (the *decomposition* approach). The result of merging alignments does not always result in a proper alignment, as the merged alignment may not be executable on the original model [2]. For this reason, we will use the more relaxed concept of pseudo-alignments. Again, this decomposition approach may be significantly faster than the monolithic approach, and a pseudo-alignment can still be used to diagnose mismatches between the event log and the process model.

[2] has also shown that formal guarantees can be given for both decomposition approaches: Both preserve perfect fitness, that is, the decomposition approach can only result in perfect fitness if the monolithic approach can result in perfect fitness, and vice versa. Moreover, when a trace is not perfectly fitting, the pseudo-alignment helps to diagnose the problem.

This paper introduces the *Divide and Conquer* tool framework, which supports both decomposed discovery and replay, and which has been implemented in ProM 6. This framework offers an easy integration of existing discovery and replay algorithms, that is, existing algorithms can be *decomposed* in an easy way. Next to the decomposition approach for process models as introduced in [2], the framework also supports two *alternative* approaches. This paper also includes an evaluation of this framework for the ILP Miner [35] and the cost-based replayer [3]. Results show that (1) the decomposition approaches provide more results in cases where monolithic approaches fail, (2) for larger cases the decomposition approaches have

[1]Note that as the 10 results may disagree with each other, this merge may be (close to) impossible.

[2]The activities have been removed in the following batches of five: first $\{J, I4, L, O, X\}$, $\{AZ, AP, AG, AD, N\}$, $\{AS, R, AW, C, D\}$, $\{AN, AX, AK, AH, AY\}$, $\{Q, Y, I2, I, AA\}$, $\{AC, AT, W, H, I5\}$, $\{AB, AF, B, AR, F\}$, $\{AJ, T, V, S, AL\}$, $\{Z, I3, G, AQ, A\}$, last $\{AE, AI, P, AU, E\}$, which leaves the activities $\{AV, U, AO, AM, M, I1, K\}$ for the final log.

typically significant speed-ups over the monolithic approaches, (3) a non-maximal decomposing discovery is about as fast as the maximal decomposing discovery while providing better results, and (4) the overhead that results from the decomposition is significant for replay, but not for discovery.

The remainder of this paper is organized as follows. Section II introduces the concepts necessary for the other sections, these include activity logs and accepting Petri nets. Note that the remainder of this paper will use accepting Petri nets as process models. Section III introduces the decomposed discovery framework, which includes (a) different heuristics to split an overall event log into sublogs, (b) information on how to add an existing discovery algorithm to the framework, (c) an approach to merge many discovered subnets into an overall accepting Petri net, and (d) the implementation of the framework in ProM 6. Section IV introduces the decomposed replay framework, which includes (a) the approach as introduced in [2] to decompose an overall accepting Petri net into subnets, (b) two alternative approaches to do this, (c) an approach to decompose the overall event log into sublogs, (d) information on how to add an existing cost-based replayer to the framework, (e) an approach on how to merge subalignments into an overall pseudo-alignment, and (f) the implementation of the framework in ProM 6. Section V introduces an evaluation conducted with the decomposition framework, which uses a number of different data sets varying in size and complexity. Section VI concludes the paper.

II. Preliminaries

This section presents the key concepts informally. See [2] for formalizations of these concepts.

A. Logs

In this paper, we often consider *activity logs*, which are an abstraction of the *event logs* as found in practice.

An *activity log* is a collection of traces, where every trace is a sequence of *activity occurrences*. Table II shows the example activity log L_1, which contains information about 20 cases. For example, 4 cases followed the trace $\langle a_1, a_2, a_4, a_5, a_8\rangle$. In total, the log contains 8 activities ($\{a_1, \ldots, a_8\}$) and $13+17+9+2\times9+9+4\times5+9+9+5+5+17+3\times5+5+5 = 156$ activity occurrences.

An *event log* is a collection of traces, where every trace is a sequence of *events*. Table I shows a typical event from an *event* log, containing the following attributes:

concept:name The activity name of the event, in this case the events refers to the activity known as a_1;

lifecycle:transition The activity transition of the event, in this case activity a_1 has been completed (other options include starting, suspending, resuming, and aborting an activity [21]);

org:resource The resource that triggered the event, in this case the resource which is known by number 112 in the organization;

time:timestamp The date and time the event occurred, in this case some time on October 1st, 2015.

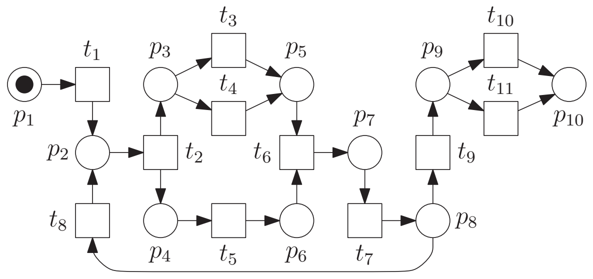


Fig. 3. A Petri net.

We assume that two events never have the same attribute values. This can be enforced by giving each event a unique identifier. An activity log can be obtained from an event log by using a so-called *classifier*, which is a set of attribute keys. Using such a classifier an activity log is obtained by replacing every event in the log with the combined values of the classifier. Typically, this will be the value of the **concept:name** attribute (see, for example, Table II), or the combined value of the **concept:name** and **lifecycle:transition** attributes.

In the remainder of this paper, a log corresponds to an activity log, unless it is explicitly stated that it is an event log.

B. Petri nets

A Petri net can model a process using three different types of elements: *places*, *transitions*, and *arcs*. Figure 3 shows an example Petri net containing 10 places ($\{p_1, \ldots, p_{10}\}$), 11 transitions ($\{t_1, \ldots, t_{11}\}$), and 24 arcs.

The dot in place p_1 is called a *token*. All tokens together indicate the current state of the Petri net, which is called a *marking*. In the example, the marking contains only a single token in place p_1, denoted $[p_1]$, but it could also contain the two tokens in place p_1 and three tokens in place p_2, denoted $[{p_1}^2, {p_2}^3]$. This latter marking would be visualized by putting two dots in place p_1 and three dots in place p_2.

As usual, a transition t is *enabled* in some marking M, denoted $M[t\rangle$, if all its input places (that is, places from which there is an arc to transition t) contain tokens in marking M. In the example Petri net, only transition t_1 is enabled in the example marking $[p_1]$, that is, $[p_1][t_1\rangle$. A transition enabled in a marking M may *fire*, resulting in a new marking M', denoted $M[t\rangle M'$, where M' equals M where one token is removed from every input place of t and one token is added to every output place of t. In the example Petri net, if transition t_1 fires at marking $[p_1]$, the new marking would be $[p_2]$, that is, $[p_1][t_1\rangle[p_2]$.

A *firing sequence* is a sequence of markings and transitions such that every transition is enabled in its predecessor marking and results in its successor marking. For example, in the example net, the sequence $\langle [p_1], t_1, [p_2], t_2, [p_3, p_4], t_3, [p_4, p_5]\rangle$ is a firing sequence. A *transition sequence* is a firing sequence projected onto the transitions. For example, the example firing sequence yields $\langle t_1, t_2, t_3\rangle$ as transition sequence.

As usual in process mining, we extend Petri nets with labels, an initial marking, and a set of final markings, yielding an

TABLE II
ACTIVITY LOG L_1 IN TABULAR FORM.

Trace	Frequency
$\langle a_1, a_2, a_4, a_5, a_6, a_2, a_4, a_5, a_6, a_4, a_2, a_5, a_7\rangle$	1
$\langle a_1, a_2, a_4, a_5, a_6, a_3, a_4, a_5, a_6, a_4, a_3, a_5, a_6, a_2, a_4, a_5, a_7\rangle$	1
$\langle a_1, a_2, a_4, a_5, a_6, a_3, a_4, a_5, a_7\rangle$	1
$\langle a_1, a_2, a_4, a_5, a_6, a_3, a_4, a_5, a_8\rangle$	2
$\langle a_1, a_2, a_4, a_5, a_6, a_4, a_3, a_5, a_7\rangle$	1
$\langle a_1, a_2, a_4, a_5, a_8\rangle$	4
$\langle a_1, a_3, a_4, a_5, a_6, a_4, a_3, a_5, a_7\rangle$	1
$\langle a_1, a_3, a_4, a_5, a_6, a_4, a_3, a_5, a_8\rangle$	1
$\langle a_1, a_3, a_4, a_5, a_8\rangle$	1
$\langle a_1, a_4, a_2, a_5, a_6, a_4, a_2, a_5, a_6, a_3, a_4, a_5, a_6, a_2, a_4, a_5, a_8\rangle$	1
$\langle a_1, a_4, a_2, a_5, a_7\rangle$	3
$\langle a_1, a_4, a_2, a_5, a_8\rangle$	1
$\langle a_1, a_4, a_3, a_5, a_7\rangle$	1
$\langle a_1, a_4, a_3, a_5, a_8\rangle$	1

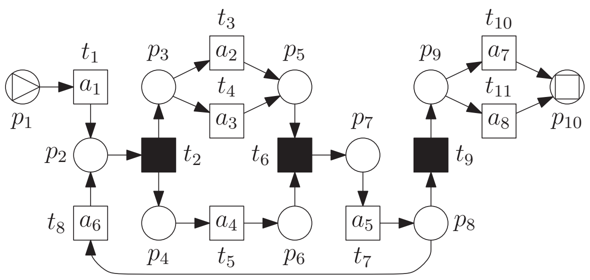


Fig. 4. An accepting Petri net N_1. Note that transitions are labeled and there is a well defined start and end.

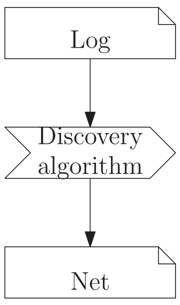


Fig. 5. Conceptual view on a discovery algorithm.

a_1	τ	a_2	a_3	$\gg$	τ	a_5	a_6	τ	a_7	a_8
t_1	t_2	t_3	$\gg$	t_5	t_6	t_7	$\gg$	t_9	t_{10}	$\gg$
0	0	0	10	10	0	0	10	0	0	10

Fig. 6. A trace alignment for the trace $\langle a_1, a_2, a_3, a_5, a_6, a_7, a_8\rangle$ and net N_1.

accepting Petri net. Figure 4 shows an accepting Petri net N_1 based on the example Petri net, with labels (like a_1 and a_8), an initial marking ($[p_1]$), and one final marking ($[p_{10}]$).

The labels are used to link transitions in the Petri net to activities in an activity log. As an example, transition t_1 is linked to activity a_1. Transitions that are linked to activities are called *visible* transitions. Transitions that are not linked to activities, like transition t_2, are called *invisible* transitions. As usual, invisible transitions are visualized using a black square.

As a result of the labeling, we can obtain an *activity sequence* from a transition sequence by removing all invisible transitions while replacing every visible transition with its label. For example, the example transition sequence $\langle t_1, t_2, t_3\rangle$ yields activity sequence $\langle a_1, a_2\rangle$ (because t_2 is invisible).

The initial marking and final markings are included to have a well defined start and end, just like the traces in the log. When replaying an activity log on a Petri net, the Petri net needs to have an initial marking to start with, and final markings to conclude whether the replay has reached a proper final state. In the example, a replay of some trace starts from marking $[p_1]$, and the replay will only be successful if marking $[p_{10}]$ is reached.

In the remainder of this paper, a net corresponds to an accepting Petri net, unless it is explicitly stated that it is a Petri net.

C. Discovery algorithms

A *discovery algorithm* (see Figure 5) is an algorithm that takes as input an overall log (like L_1) over some set of activities A and that creates as output a net (like net N_1) over the same set of activities A. Note that we do assume that the labeling function of the created net is surjective (there is at least one transition for every activity), but that we do not assume that it is injective (there may be multiple transitions labeled with the same activity). Example discovery algorithms that do result in an injective labeling function include the Alpha Miner [5], the Heuristics Miner [34], the Hybrid ILP Miner [36], the ILP Miner [35], and the Inductive Miner [23]. Example discovery algorithms that may result in a non-injective labeling function include the Evolutionary Tree Miner [10].

D. Alignments

A *trace alignment* links activities in a trace onto transitions in a net in a best-possible way. As an example, Figure 6 shows a possible trace alignment for the trace $\langle a_1, a_2, a_3, a_5, a_6, a_7, a_8\rangle$ and net N_1. The top row in this alignment contains the

a_1	τ	a_2	a_3	$\gg$	τ	a_5	a_6	τ	a_7	a_8
t_1	t_2	$\gg$	t_4	t_5	t_6	t_7	$\gg$	t_9	$\gg$	t_{11}
0	0	10	0	10	0	0	10	0	10	0

Fig. 7. Another optimal trace alignment for the trace $\langle a_1, a_2, a_3, a_5, a_6, a_7, a_8 \rangle$ and net N_1.

activities from the trace, the middle row the *transitions* from the net, and the bottom row the *costs* associated with that activity and transition. When reading this particular alignment from left to right, we encounter four different kinds of *legal moves*:

1) Activity a_1 in the log matches transition t_1. This is a so-called *synchronous move*, as both the trace and the net can advance (the trace by stepping over the activity, the net by executing the transition).
2) The *invisible* transition t_2 is not matched by any activity in the log, which is okay as this is an invisible transition. This is a so-called *invisible model move*, as only the net can advance (by executing the invisible transition). For such a move, we use τ to denote the lack of a matching activity.
3) Activity a_2 matches transition t_3. This is another synchronous move.
4) Activity a_3 is not matched by any transition. This is a so-called *log move*, as only the trace can advance (by stepping over the activity). For such move, we use $\gg$ to denote the lack of a matching transition.
5) The *visible* transition t_5 is not matched by the corresponding activity a_4 in the log. This is a so-called *visible model move*, as only the net can advance (by executing the visible transition). For such move, we use $\gg$ to denote the lack of a matching activity in the log.
6) And so on.

Note that we require the transition sequence in the middle row of the alignment to lead from the initial marking of the net to a final marking.

In this example, a synchronous move costs 0, a visible model move costs 10, an invisible model move costs 0, and a log move also costs 10. The total costs for the example alignment is $0+0+0+10+10+0+0+10+0+0+10 = 40$.

If no other alignment results in lower costs, the alignment is called *optimal*. There may exist multiple optimal alignments for a single trace. For example, the alignment as shown in Figure 6 is optimal, but the alignment as shown in Figure 7 is also optimal.

A *log alignment* is an optimal trace alignment for every trace in the activity log. As a result of a log alignment, any trace in the log can be mapped to the transition sequence that best matches this trace. As an example, Table III shows optimal trace sequences (it is straightforward to obtain the alignments from these sequences) for log L_1 and net N_1, where the same costs are used as mentioned above. Clearly, log L_1 can be perfectly aligned to net N_1, which results in no costs and perfect fitness. Using such a log alignment, it is possible to project the date and information that is present in an event log onto the net, and obtain average durations between activities and such.

E. Replay algorithms

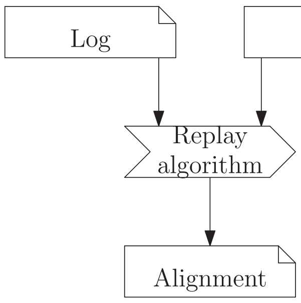


Fig. 8. Conceptual view on a replay algorithm.

A *replay algorithm* (see Figure 8) is an algorithm that, given the costs for all possible moves, takes a log (like L_1) over some set of activities A and a net (like N_1) over the same set of activities A, and creates a log alignment over the same set of activities A.

F. Data sets

Table IV shows the list of data sets (with their characteristics) that are used for the evaluation in this paper.

III. Decomposed discovery framework

The goal of decomposed discovery is to apply an existing discovery algorithm on a series of sublogs instead of one overall log, where every sublog contains significantly less different activities than the overall log. Under the assumption that the complexity of the discovery algorithm is significantly worse than linear, the decomposed discovery algorithm is expected to finish well before the monolithic discovery algorithm.

For this reason, the decomposed discovery algorithm first determines small sets of different activities that are expected to have direct causal relations among themselves. These sets of activities are referred to as activity clusters in the remainder of this paper. Figure 9 then shows a conceptual view on a decomposed discovery algorithm. First, the algorithm determines an as-best-as-possible set of activity clusters. Second, for every activity cluster, the algorithm filters the overall activity log into a sublog. Third, the algorithm discovers a subnet from the sublog using the provided discovery algorithm. Fourth and last, the subnets are merged into an overall net.

This section first introduces each of these steps in detail. Second, it introduces the implementation of the decomposed discovery algorithm in ProM 6.

A. Discover clusters

The goal of this step is to obtain an as-best-as-possible set of small activity clusters, where the activities within a single cluster have direct causal relations among themselves. Figure 10 shows the approach the decomposed discovery algorithm uses to achieve this. First, a matrix is discovered

TABLE III
OPTIMAL TRACE SEQUENCES FOR LOG L_1 AND NET N_1.

Trace sequence	**Costs**
$\langle t_1, t_2, t_3, t_5, t_6, t_7, t_8, t_2, t_3, t_5, t_6, t_7, t_8, t_5, t_2, t_3, t_6, t_7, t_9, t_{10} \rangle$	0
$\langle t_1, t_2, t_3, t_5, t_6, t_7, t_8, t_2, t_4, t_5, t_6, t_7, t_8, t_5, t_2, t_4, t_6, t_7, t_8, t_2, t_3, t_5, t_6, t_7, t_9, t_{10} \rangle$	0
$\langle t_1, t_2, t_3, t_5, t_6, t_7, t_8, t_2, t_4, t_5, t_6, t_7, t_9, t_{10} \rangle$	0
$\langle t_1, t_2, t_3, t_5, t_6, t_7, t_8, t_2, t_4, t_5, t_6, t_7, t_9, t_{11} \rangle$	0
$\langle t_1, t_2, t_3, t_5, t_6, t_7, t_8, t_5, t_2, t_4, t_6, t_7, t_9, t_{10} \rangle$	0
$\langle t_1, t_2, t_3, t_5, t_6, t_7, t_9, t_{11} \rangle$	0
$\langle t_1, t_2, t_4, t_5, t_6, t_7, t_8, t_5, t_2, t_4, t_6, t_7, t_9, t_{10} \rangle$	0
$\langle t_1, t_2, t_4, t_5, t_6, t_7, t_8, t_5, t_2, t_4, t_6, t_7, t_9, t_{11} \rangle$	0
$\langle t_1, t_2, t_4, t_5, t_6, t_7, t_9, t_{11} \rangle$	0
$\langle t_1, t_2, t_5, t_3, t_6, t_7, t_8, t_2, t_5, t_3, t_6, t_7, t_8, t_2, t_4, t_5, t_6, t_7, t_8, t_2, t_3, t_5, t_6, t_7, t_9, t_{11} \rangle$	0
$\langle t_1, t_2, t_5, t_3, t_6, t_7, t_9, t_{10} \rangle$	0
$\langle t_1, t_2, t_5, t_3, t_6, t_7, t_9, t_{11} \rangle$	0
$\langle t_1, t_2, t_5, t_4, t_6, t_7, t_9, t_{10} \rangle$	0
$\langle t_1, t_2, t_5, t_4, t_6, t_7, t_9, t_{11} \rangle$	0

TABLE IV
DATA SETS USED IN THE EVALUATION.

Data Sets	**Description**
DMKD 2006 [25]	20 synthetic events logs generated from 4 Petri nets, containing 12, 22, 32, and 42 activities, 1000 traces, and different noise levels.
IS 2014 [28]	32 synthetic event logs generated from 4 highly structured Petri nets, containing 59, 48, 32, and 57 activities, 2000 traces, 4 different average trace lengths (approx. 15–55), with and without noise.
BPM 2013 [27]	7 synthetic event logs (A–G) generated from 7 highly structured Petri nets, containing 317, 317, 317, 429, 275, 299, and 335 activities, log C contains 500 traces, all other logs contain 1200 traces, log B is 100% fitting its model, all other events logs do not fit 100%.
BPIC 2012 [17]	1 real-life event log, 13087 traces, 262200 events, and 36 activities.
BPIC 2015 [18]	5 real-life event logs, 832–1409 traces, 44354–59681 events, and 356–410 activities.

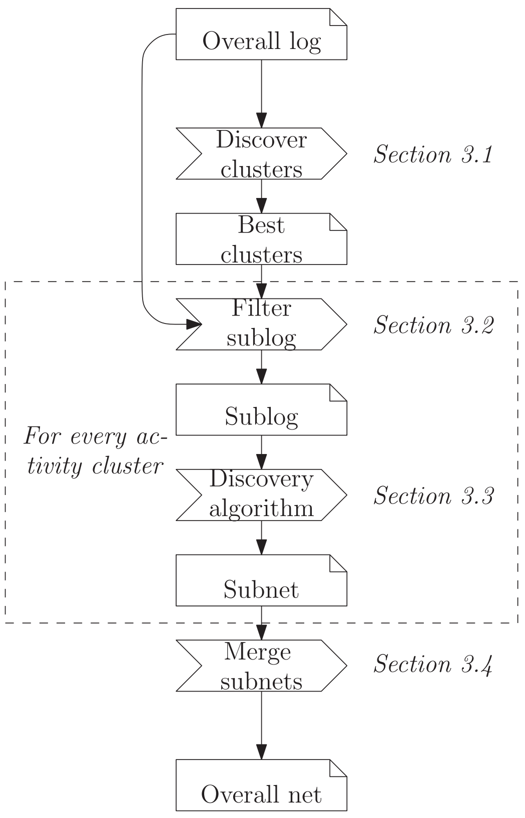


Fig. 9. Conceptual view on a decomposed discovery algorithm.

from the overall log indicating for every pair of activities how strong the direct causal relation is from the first to the second. Second, a graph is derived from this matrix containing only the strongest relations. Third, an initial set of activity clusters is derived from this graph. Fourth, a set of grouped activity clusters is derived from the initial set of clusters by grouping very small or very coherent clusters together. These four steps are executed using twelve different settings, leading to a collection of 12 sets of clusters. Fifth and last, the best set from the collection is selected and returned as result.

1) Discover matrix: This step discovers a matrix (using some heuristics) that contains for every pair of two activities the estimated strength of the direct causal relation from the first activity to the second. We will refer to such a matrix as a *causal activity matrix*. Table V shows an example causal activity matrix M_1 for log L_1.

The strengths in a causal activity matrix range from -1.0 to 1.0, which should be interpreted as follows:

- A value of 1.0 indicates that it is sure that there is a direct causal relation.
- A value of 0.5 indicates that it is likely that there is a direct causal relation.
- A value of 0.0 indicates that we do not know whether there is a direct causal relation or not.
- A value of -0.5 indicates that it is likely that there is *no* direct causal relation.
- A value of -1.0 indicates that it is sure that there is *no* direct causal relation.

TABLE V
EXAMPLE CAUSAL ACTIVITY MATRIX M_1 FOR LOG L_1.

From/To	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
a_1	-0.41	0.91	0.75	0.88	-1.00	-1.00	-1.00	-1.00
a_2	-1.00	-0.79	-1.00	0.29	0.88	-1.00	-1.00	-1.00
a_3	-1.00	-1.00	-0.76	0.10	0.86	-1.00	-1.00	-1.00
a_4	-1.00	-0.29	-0.13	-0.86	1.00	-1.00	-1.00	-1.00
a_5	-1.00	-1.00	-1.00	-1.00	-1.00	0.93	0.90	0.92
a_6	-1.00	0.75	0.83	0.86	-1.00	-0.60	-1.00	-1.00
a_7	-1.00	-1.00	-1.00	-1.00	-1.00	-1.00	-0.62	-1.00
a_8	-1.00	-1.00	-1.00	-1.00	-1.00	-1.00	-1.00	-0.63

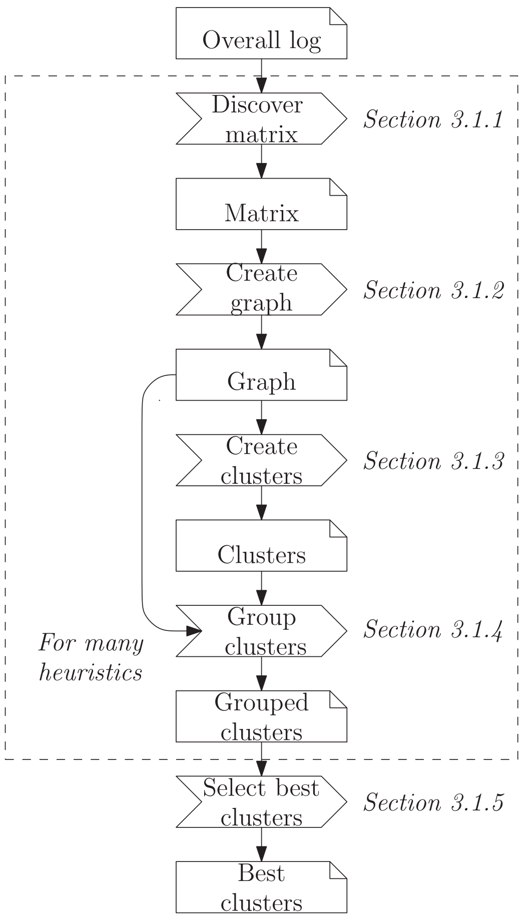


Fig. 10. Conceptual view on discovering activity clusters.

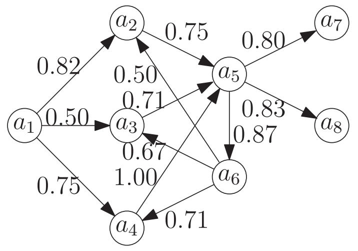


Fig. 11. Example causality graph G_1 for matrix M_1.

For example, based on M_1, we are sure that there is a direct causal relation from a_4 to a_5 (as $M_1(a_5, a_5) = 1.0$), and we are sure that there is no direct causal relation from a_2 to a_1 (as $M_1(a_2, a_1) = -1.0$).

Table VI shows an overview of the heuristics currently implemented in the decomposed discovery framework for discovering a causal activity matrix from a log.

2) Create graph: In this step, we create a graph containing the strongest direct causal relations, by filtering those strongest relations from the casual activity matrix. We will refer to such graph as a *causal activity graph*. Figure 11 shows a causal activity graph G_1 created from causal activity matrix M_1. The strengths of the arcs in the causal activity graph range from 0.0 (exclusive) to 1.00 (inclusive). The closer the strength is to 1.0, the more confident we are there there is indeed such a direct causal relation. For example, based on G_1, we are sure that there is a causal relation from a_4 to a_5 (weight is 1.0), and there might be a causal relation from a_6 to a_2 (weight is 0.5).

To create a causal activity graph from a causal activity matrix M, we simply take all values from M that exceed 0.0. However, prior to doing this, we first apply two transformations on M, which might affect the outcome.

The first transformation is the *zero value* transformation, which takes a new zero value $z \in (-1.0, 1.0)$ and transforms M to $M\bot_z$ using the following rule:

$$M\bot_z(a, a') = \begin{cases} 1.0 & \text{if } M(a, a') = 1.0; \\ \frac{M(a,a')-z}{1.0-z} & \text{if } M(a, a') \in (z, 1.0); \\ 0.0 & \text{if } M(a, a') = z; \\ \frac{M(a,a')-z}{1.0+z} & \text{if } M(a, a') \in (-1.0, z); \\ -1.0 & \text{if } M(a, a') = -1.0. \end{cases}$$

Figure 12 shows this transformation graphically: The selected value will become the new zero value, and all other values are scaled linearly. Clearly, this transformation has an effect on which values in the matrix will be selected for arcs in the graph: Any value exceeding value z will be selected, any other value will not. As an example, causal activity graph G_1 was obtained from matrix $M_1\bot_{0.5}$.

The second transformation is the *concurrency threshold* transformation, which takes a concurrency threshold $c \in (0.0, 1.0]$ and transforms M to $M\|_c$ using the following rule:

$$M\|_c(a, a') = \begin{cases} -0.5 & \text{if } |M(a, a') - M(a', a)| < c; \\ M(a, a') & \text{otherwise.} \end{cases}$$

TABLE VI
HEURISTICS TO DISCOVER A CAUSAL ACTIVITY MATRIX.

Heuristic	**Description**
Heuristics	A simple heuristic based on how often a is directly followed by a' in L and vice versa.
Fuzzy	A more involved heuristic that also takes second-order effects (like how often a is directly followed by a' compared to how often a it directly followed by a'').
Alpha	A heuristic based on the Alpha miner. If the Alpha miner creates a place between the transitions labeled with a and a', then this heuristics returns 1.0, otherwise -1.0.
Random	A heuristic that returns a random value (for testing purposes only).
Average	A meta heuristic that returns the average value of the *Heuristics*, *Fuzzy*, and *Alpha* heuristics.
Mini	A meta heuristic that returns the minimal value of the *Heuristics*, *Fuzzy*, and *Alpha* heuristics.
Midi	A meta heuristic that returns the middle value of the *Heuristics*, *Fuzzy*, and *Alpha* heuristics.
Maxi	A meta heuristic that returns the maximal value of the *Heuristics*, *Fuzzy*, and *Alpha* heuristics.

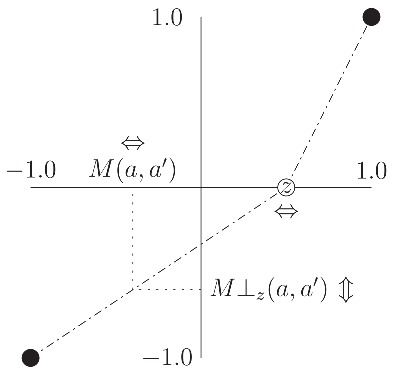


Fig. 12. Zero value parameter.

This transformation can be used to downplay values in the matrix in case the relation between activities are balanced, which may be caused because both activities can be executed concurrently. In case of concurrent activities, direct causal relations are not wanted.

3) Create clusters: This step creates an initial set of activity clusters from a causal activity graph. These activity clusters are created by first assigning an equivalence class on the arcs in the graph, where the following restrictions apply:

Input arcs Input arcs of the same node are equivalent.
Output arcs Output arcs of the same node are equivalent.

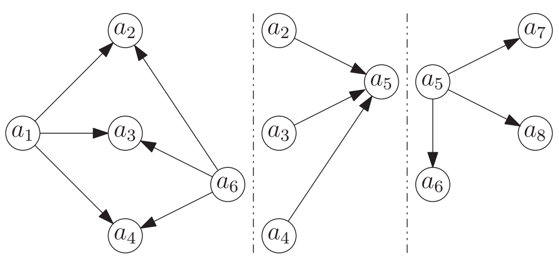


Fig. 13. Example activity clusters C_1 for graph G_1.

As an example, Figure 13 shows the set of activity clusters created from causal activity graph G_1.

To prevent any confusion in the remaining steps, the set of activity clusters are ordered. As a result, there will be a first cluster, a second clusters, etc. This ordering allows us to keep track of which subnet was discovered from which sublog, and, later on, which sublog should be replayed on which subnet.

4) Group clusters: This step changes the initial set of activity clusters by grouping clusters that are strongly related to each other. As an example, given that the two leftmost clusters as shown in Figure 13 have three activities in common, it might be a good idea to join both clusters. Figure 14 shows the resulting set of activity clusters.

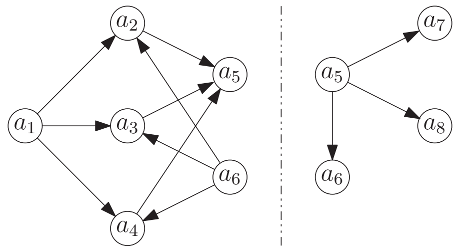


Fig. 14. Example grouped activity clusters C_2.

This grouping of clusters, along with the reason for doing this, has been described in detail in [22]. In short, a set of activity clusters is considered to be better if it scores better on the weighted quality metrics as shown in Table VII. Each of these metrics provides a value between 0.0 and 1.0, and using the provided relative weights, an end score is determined.

This step starts with the initial set of clusters and requires a *percentage of clusters* as input. As long as the number of clusters divided by the number of initial clusters exceeds the given percentage, this step selects the best two different activity clusters to be merged, and merges them.

5) Select best clusters: Instead of relying on a single heuristic, the decomposed discovery algorithm relies on three different discovery heuristics with four different zero values each. Table VIII shows an overview of the discovery heuristics used for selecting the best activity clusters. For each of these combinations, the set of activity clusters is determined. From these sets of clusters, the best one is selected.

TABLE VII
QUALITY METRICS FOR ACTIVITY CLUSTERS. ALTHOUGH OTHER METRICS ARE POSSIBLE AS WELL, WE LIMIT OURSELVES HERE TO THOSE DEFINED IN [22].

Metric	**Description**
Cohesion	The causal relation strengths within every single cluster should be maximal.
Coupling	The causal relation strengths between every two different clusters should be minimal.
Size	The sizes of the clusters should be distributed evenly.
Overlap	The overlap (common activities) between every two different clusters should be minimal.

TABLE VIII
COLLECTION OF HEURISTICS TO SELECT THE BEST ACTIVITY CLUSTERS FROM.

Heuristic	**Zero values**	**Concurrency threshold**
Heuristics	$\{-0.5, 0.0, 0.5, 0.9\}$	$\{0.005\}$
Fuzzy	$\{-0.6, -0.5, 0.0, 0.5\}$	$\{0.005\}$
Midi	$\{-0.5, 0.0, 0.5, 0.9\}$	$\{0.005\}$

The reason for selecting these heuristics is that we have seen that sometimes one works best, and sometimes another. The reason for using the values -0.5, 0.0, and 0.5 as zero values is to have some coverage of the entire space of this parameters, that is, $(-1.0, 1.0)$. The reason for adding the values -0.6 and 0.9 is that we have seen that for these values these heuristics often provide good results.

B. Filter sublog

The goal of this step it so split the overall activity log into a sublog for every activity cluster. The sublogs are ordered in the same way as the activity clusters are ordered. As a result, the first sublog corresponds to the first cluster etc. As an example, Table IX shows the sublogs resulting from filtering log L_1 using the activity clusters C_1.

This filtering works for most of the existing discovery algorithms, but the Alpha Miner is a known exception. As

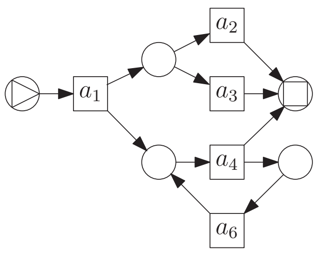


Fig. 15. Result of the Alpha Miner on the sublog obtained for cluster $\{a_1, a_2, a_3, a_4, a_6\}$.

an example of this, Figure 15 shows the result of running the Alpha Miner on the first sublog, that is on the log that corresponds to the cluster $\{a_1, a_2, a_3, a_4, a_6\}$. Obviously, the Alpha Miner is unable to properly handle the activity a_4 correctly, which is caused by the fact that it appears both as a final activity (like in the trace $\langle a_1, a_2, a_4\rangle$) and in the middle of a trace (like in the trace $\langle a_1, a_4, a_2\rangle$) [2]. For the second cluster, the fact that activity a_4 appears both as an initial activity and in the middle of a trace, results in a similar problem.

TABLE X
FILTERED TRACES WITH ARTIFICIAL START AND END ACTIVITIES ADDED FOR THE FIRST CLUSTER IN TABLE IX.

Cluster $\{a_1, a_2, a_3, a_4, a_6\}$
$\langle \alpha, a_1, a_2, a_4, a_6, a_2, a_4, a_6, a_4, a_2, \omega\rangle$
$\langle \alpha, a_1, a_2, a_4, a_6, a_3, a_4, a_6, a_4, a_3, a_6, a_2, a_4, \omega\rangle$
$\langle \alpha, a_1, a_2, a_4, a_6, a_3, a_4, \omega\rangle$
$\langle \alpha, a_1, a_2, a_4, a_6, a_3, a_4, \omega\rangle$
$\langle \alpha, a_1, a_2, a_4, a_6, a_4, a_3, \omega\rangle$
$\langle \alpha, a_1, a_2, a_4, \omega\rangle$
$\langle \alpha, a_1, a_3, a_4, a_6, a_4, a_3, \omega\rangle$
$\langle \alpha, a_1, a_3, a_4, a_6, a_4, a_3, \omega\rangle$
$\langle \alpha, a_1, a_3, a_4, \omega\rangle$
$\langle \alpha, a_1, a_4, a_2, a_6, a_4, a_2, a_6, a_3, a_4, a_6, a_2, a_4, \omega\rangle$
$\langle \alpha, a_1, a_4, a_2, \omega\rangle$
$\langle \alpha, a_1, a_4, a_2, \omega\rangle$
$\langle \alpha, a_1, a_4, a_3, \omega\rangle$
$\langle \alpha, a_1, a_4, a_3, \omega\rangle$

The typical work-around to overcome this problem is to introduce an *artificial start activity* α and an *artificial end activity* ω. These two artificial transitions prevent that an initial or a final activity also occurs in the middle of a trace. Table X shows the result of adding these two artificial activities to the first sublog. Figure 16 shows the result of running the Alpha miner on this sublog. Clearly, the resulting net now handles activity a_4 in a proper way.

For this reason, when filtering the overall log for a sublog using an activity cluster, we include the option to add artificial start and end activities to the resulting sublog. Obviously, this creates the obligation to remove the transitions labeled with these activities later on, that is, when merging the subnets into an overall net.

TABLE IX
FILTERED TRACES FOR ACTIVITY LOG L_1 AND ACTIVITY CLUSTERS C_1 IN FIGURE 13 IN TABULAR FORM.

Cluster $\{a_1, a_2, a_3, a_4, a_6\}$	**Cluster** $\{a_2, a_3, a_4, a_5\}$	**Cluster** $\{a_5, a_6, a_7, a_8\}$
$\langle a_1, a_2, a_4, a_6, a_2, a_4, a_6, a_4, a_2\rangle$	$\langle a_2, a_4, a_5, a_2, a_4, a_5, a_4, a_2, a_5\rangle$	$\langle a_5, a_6, a_5, a_6, a_5, a_7\rangle$
$\langle a_1, a_2, a_4, a_6, a_3, a_4, a_6, a_4, a_3, a_6, a_2, a_4\rangle$	$\langle a_2, a_4, a_5, a_3, a_4, a_5, a_4, a_3, a_5, a_2, a_4, a_5\rangle$	$\langle a_5, a_6, a_5, a_6, a_5, a_6, a_5, a_7\rangle$
$\langle a_1, a_2, a_4, a_6, a_3, a_4\rangle$	$\langle a_2, a_4, a_5\rangle$	$\langle a_5, a_6, a_5, a_7\rangle$
$\langle a_1, a_2, a_4, a_6, a_3, a_4\rangle$	$\langle a_2, a_4, a_5, a_3, a_4, a_5\rangle$	$\langle a_5, a_6, a_5, a_8\rangle$
$\langle a_1, a_2, a_4, a_6, a_4, a_3\rangle$	$\langle a_2, a_4, a_5, a_4, a_3, a_5\rangle$	$\langle a_5, a_6, a_5, a_7\rangle$
$\langle a_1, a_2, a_4\rangle$	$\langle a_2, a_4, a_5\rangle$	$\langle a_5, a_8\rangle$
$\langle a_1, a_3, a_4, a_6, a_4, a_3\rangle$	$\langle a_3, a_4, a_5, a_4, a_3, a_5\rangle$	$\langle a_5, a_6, a_5, a_7\rangle$
$\langle a_1, a_3, a_4, a_6, a_4, a_3\rangle$	$\langle a_3, a_4, a_5, a_4, a_3, a_5\rangle$	$\langle a_5, a_6, a_5, a_8\rangle$
$\langle a_1, a_3, a_4\rangle$	$\langle a_3, a_4, a_5\rangle$	$\langle a_5, a_8\rangle$
$\langle a_1, a_4, a_2, a_6, a_4, a_2, a_6, a_3, a_4, a_6, a_2, a_4\rangle$	$\langle a_4, a_2, a_5, a_4, a_2, a_5, a_3, a_4, a_5, a_2, a_4, a_5\rangle$	$\langle a_5, a_6, a_5, a_6, a_5, a_6, a_5, a_8\rangle$
$\langle a_1, a_4, a_2\rangle$	$\langle a_4, a_2, a_5\rangle$	$\langle a_5, a_7\rangle$
$\langle a_1, a_4, a_2\rangle$	$\langle a_4, a_2, a_5\rangle$	$\langle a_5, a_8\rangle$
$\langle a_1, a_4, a_3\rangle$	$\langle a_4, a_3, a_5\rangle$	$\langle a_5, a_7\rangle$
$\langle a_1, a_4, a_3\rangle$	$\langle a_4, a_3, a_5\rangle$	$\langle a_5, a_8\rangle$

TABLE XI
DISCOVERY ALGORITHMS IN PROM 6 [33] SUPPORTED BY THE FRAMEWORK. THE CONVERSION PLUG-INS LISTED ARE NECESSARY TO CONVERT A NATIVE RESULT (LIKE A HEURISTICS NET OR A PROCESS TREE) INTO A PETRI NET.

Discovery algorithm	**Discovery Plug-in**	**Conversion Plug-in**
Alpha Miner [5]	"Alpha Miner"	
Heuristics Miner [34]	"Mine for a Heuristics Net using Heuristics Miner"	"Convert Heuristics net into Petri net"
Hybrid ILP Miner [36]	"ILP-Based Process Discovery"	
ILP Miner [35]	"ILP Miner"	
Inductive Miner [23]	"Mine Petri net with Inductive Miner, with parameters"	
Evolutionary Tree Miner [10]	"Mine a Process Tree with ETMd using parameters and classifier"	"Convert Process Tree to Petri Net"

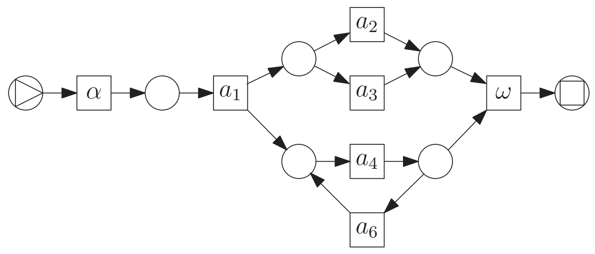


Fig. 16. Result of the Alpha Miner on the first sublog with artificial start and end activities.

C. Discovery algorithm

The goal of this step is to discover a subnet from every sublog by using the provided discovery algorithm. Table XI shows a list of existing discovery algorithms in ProM 6 that are currently supported by the framework. Some of the existing discovery algorithms do not discover a Petri net, and require a conversion algorithm to convert the discovered model into a Petri net. Although the ideas in this paper are not Petri-net specific, the framework is tailored towards Petri nets to allow for a modular approach. Without this, we would need to customize things for every discovery approach.

The main problem with this step is that these existing algorithms discover a Petri net rather than an accepting Petri net. The initial marking is often clearly defined, but usually the set of final markings is left implicit or unknown. The framework offers two solutions for this problem:

1) Some discovery algorithms do in fact discover a Petri net with an *explicit initial marking* and a collection of *explicit final markings*. Example discovery algorithms for which this holds include the Inductive Miner and the Evolutionary Tree Miner. For such algorithms a wrapper is available that first finds these initial and final markings for the Petri net at hand, and second constructs an accepting Petri net from them.
2) Other discovery algorithms only discover a Petri net with an *implicit initial marking* (containing a single token in every source place) and a collection of *implicit final markings* (where each final marking contains a single token in a single sink place). Example discovery algorithms for which this holds include the Alpha Miner, the Heuristics Miner, the ILP Miner, and the Hybrid ILP Miner. The Alpha Miner and Heuristics Miner always discover a Petri net with a single source place and a single sink place, with the underlying assumption that the initial marking contains a single token in the source place and the only final marking contains a single token in the sink

place. The ILP Miner and the Hybrid ILP Miner always discover a Petri net with any number of source places and no sink places, with the underlying assumption that the initial marking contains a token in every source place, and that the only final marking is the empty marking. For these algorithms a different wrapper is available that first creates these initial and final markings from the net at hand, and second constructs an accepting Petri net from them.

Using these two wrappers, all discovery algorithms mentioned in Table XI could be added with ease to the framework. In case a discovery algorithm does not provide any initial and final markings (be it implicit or explicit), or in case the algorithm has different implicit markings than the ones mentioned, then a specific wrapper needs to be created for it. This is allowed by the framework but it will take some effort.

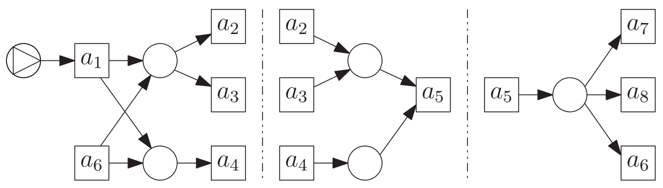


Fig. 17. Result of the Hybrid ILP Miner on all sublogs from Table IX.

As an example, Figure 17 shows the resulting subnets that the Hybrid ILP Miner discovered from the sublogs that are shown in Table IX.

Please note that all these discovery algorithms are oblivious to the fact that the sublog may contain artificial activities. These algorithms will just discover a Petri net from the sublog that was provided to them.

D. Merge subnets

The goal of this step is to merge the discovered subnets into one overall net. This merge is done in three steps: joining the subnets, hiding all transitions labeled with artificial activities, and reducing the net. The result after reduction will be the accepting Petri net that results from the merge.

1) Join subnets: This step joins a collection of subnets into one overall net using the following rules (cf. [2]):

Place Every place from every subnet is copied into the overall net.

Invisible transitions Every invisible transition from every subnet is copied into the overall net.

Visible transitions For every label, a single visible transition with that label is selected as proxy for all other transitions in all subnets with that label. Only the proxy transition is copied into the overall net.

Arc Every arc from every subnet is copied into the overall net, where a transition is replaced by its proxy if it has a proxy.

Initial marking The initial markings of all subnets are combined into the overall initial marking.

Final markings For every possible combination of final markings in the small net, an overall final marking will be created. Note that markings are multisets of tokens that can be combined easily.

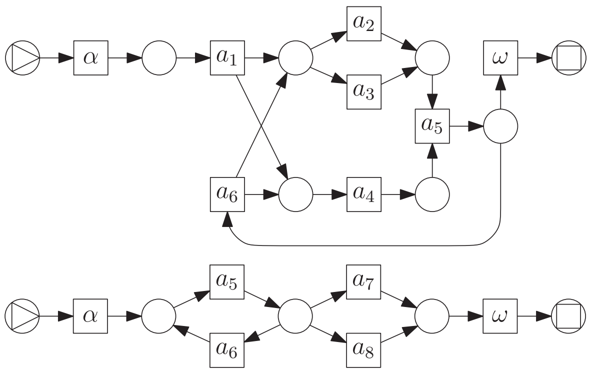
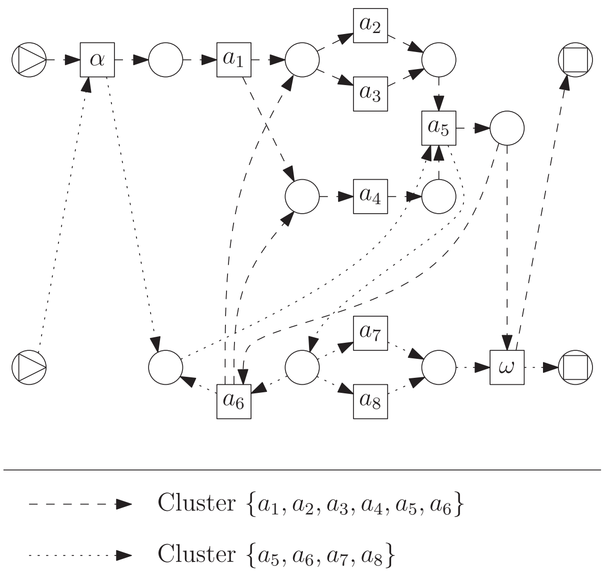


Fig. 18. Possible nets resulting from discovery algorithm.

Assume, for the sake of argument, that some discovery algorithm has discovered the two subnets as shown in Figure 18. Joining these two subnets results in the overall net shown in Figure 19.


α
a_1
a_2
a_3
a_5
a_4
a_7
a_6
a_8
ω
Cluster $\{a_1, a_2, a_3, a_4, a_5, a_6\}$
Cluster $\{a_5, a_6, a_7, a_8\}$
Fig. 19. Net that result from joining the subnets as shown in Figure 18.

Note that in this step, we join *all* visible transition with the same label by selecting a proxy and by rerouting all arcs to and from this proxy. However, this will not work in case one (or more) of the subnets contains duplicate transitions (that is, multiple visible transitions sharing the same label). As a result of the rules, these two transitions would be joined as well. As an example, consider the subnet as shown in Figure 20. In this net, the visible transitions t_4 and t_6 share label a_6. Obviously, joining these transitions is not desired: t_4 and t_6 cannot both occur at the same time, so merging them leads to a deadlock. As a result, before joining the subnets, we need to make sure that every subnet does not have duplicate transitions.

Figure 21 shows the solution used to solve this problem: In every subnet, if duplicate transitions exist, then the construct as shown in this figure is applied. Every firing of transitions t_4

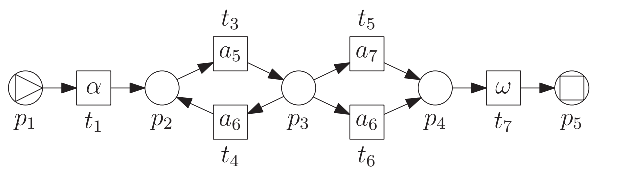


Fig. 20. Possible discovered net that contains two duplicate transitions labeled a_6.

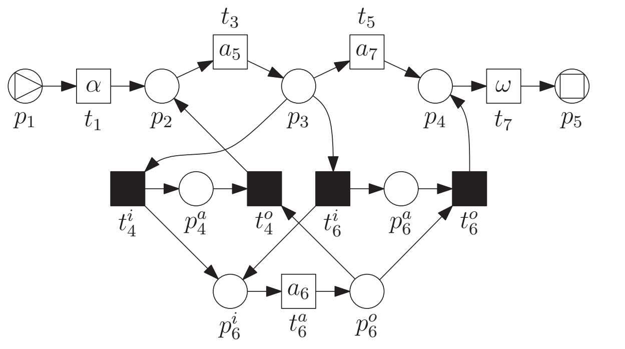


Fig. 21. Similar net that contains only one visible transition labeled a_6.

and t_6 in the subnet is now replaced by the transition sequences $\langle t_4^i, t_6^a, t_4^o\rangle$ and $\langle t_6^i, t_6^a, t_6^o\rangle$ and vice versa. The places p_4^a and p_6^a guarantee that any firing of any transition labeled with a_6 gets routed into the right direction. As an example, transition t_4^o can only fire if t_4^i has fired before. As a result, we obtain an adapted subnet that has similar behavior but which does not contain duplicate transitions.

To avoid joining duplicate transitions in a single subnet, before joining all subnets, all duplicate transition is a single subnet are removed first by adapting every subnet.

2) Hide transitions labeled with artificial activities: This step removes the labels of the artificial activities that may have been inserted into the sublogs in an earlier step. To remove these labels, the corresponding transitions are simply made invisible. As an example, Figure 22 shows the result of

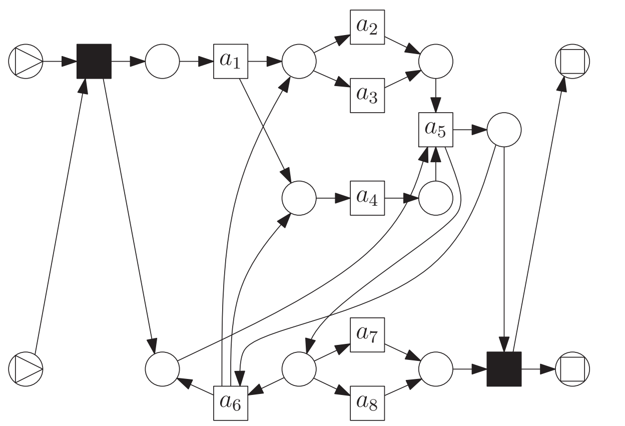


Fig. 22. Net from Figure 19 with artificial labels made invisible.

performing this step on the net as shown in Figure 19.

3) Reduce net: This step reduces the size of the overall Petri net by applying variants on classical behavior-preserving reduction rules [29] and by removing places that are structurally redundant [8]. The classical behavior-preserving reduction rules had to be adapted to take initial markings and visible transitions into account. Note that we only reduce invisible transitions and need to keep track of initial and final markings.

As a result of applying a reduction rule, the initial marking of the overall net may need to be updated. Consider, for example, the silent transition in Figure 22 that corresponds to the transition labeled α in Figure 19. This transition and its input places can be removed from the net, but then the tokens from the initial marking need to be moved from the input places to the output places. Otherwise, the initial marking would get lost.

No reduction step should remove a visible transition. Only invisible transitions and places may be reduced by these rules, but all visible transition should remain. Consider, for example, the transition labeled a_2 in Figure 19. This transition has the same input places and the same output places as the transition labeled a_3. As a result, the so-called *Fusion of Parallel Transitions* reduction rule [29] could remove one of these activities. Clearly, this is not desired, as these transitions are there to explain the behavior as found in the log. Removing them now would defeat the purpose of the process discovery from event data.

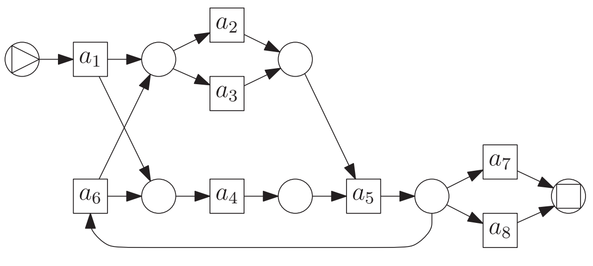


Fig. 23. Net from Figure 22 after reductions.

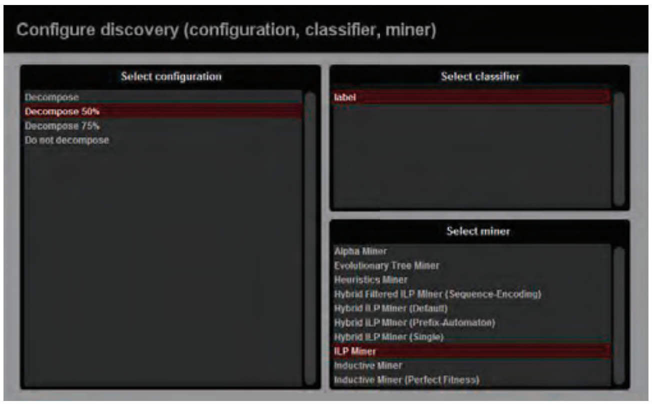
As an example, Figure 23 shows the result of performing this step on the net as shown in Figure 22. In this example, the reduction was able to remove all invisible transitions. However, in general, this might not be the case (e.g. skip transitions).

E. Implementation

The decomposed discovery algorithm has been implemented as the *Discover using Decomposition* action in the *DecomposedMiner* package of ProM 6. Figure 24 shows the dialog for this action, which allows the user to select the *configuration*, the classifier (see Section II-A), and the discovery algorithm (see Section III-C).

A configuration of the algorithm determines predefined values for the settings in the algorithm. The following configurations can be selected:

Decompose This configuration uses all steps (as described before) with default values. For discovering a matrix (see Section III-A1), the aforementioned 12 configurations


Configure discovery (configuration, classifier, miner)

Select configuration
- Decompose
- Decompose 50%
- Decompose 75%
- Do not decompose

Select classifier
- label

Select miner
- Alpha Miner
- Evolutionary Tree Miner
- Heuristics Miner
- Hybrid Filtered ILP Miner (Sequence-Encoding)
- Hybrid ILP Miner (Default)
- Hybrid ILP Miner (Prefix-Automaton)
- Hybrid ILP Miner (Single)
- ILP Miner
- Inductive Miner
- Inductive Miner (Perfect Fitness)

Fig. 24. Dialog for *Discover using Decomposition* action in ProM 6.

(see Table VIII) are used. For creating a graph (see Section III-A2), the zero value is set to 0.0 and the concurrency threshold to 0.005. For creating initial clusters (see Section III-A3), no parameters are required. For filtering the overall log (see Section III-B), empty traces are not removed, and artificial start and end activities (called " |start> " and " [end] ") are added only in case the Alpha Miner is selected as discovery algorithm. For discovering the subnets from the sublogs (see Section III-C), the selected discovery algorithm is used. Merging the subnets into an overall net (see Section III-D) first removes the structural redundant places and then reduces the result using the improved classical reduction rules.

Decompose 75% This configuration is identical to the *Decompose* configuration, except that the clusters are now grouped to 75% of the original number of clusters (see Section III-A5). As an example, if the best initial clusters contained 20 clusters, then these clusters would be grouped into 15 clusters by this configuration.

Decompose 50% This *default* configuration is identical to the *Decompose* configuration, except that the clusters are now grouped to 50% of the original number of clusters. This results in 10 clusters in case there are 20 clusters in the best clustering.

Do not decompose This configuration creates an activity cluster array with a single cluster containing all activities, and it does not add artificial start and end events. As a result, the selected miner is run with the selected classifier on the original log. This configuration corresponds to the monolithic approach, and offers a baseline for comparison.

The first three configurations allow the user to select the level of decomposition from maximal (*Decompose*) to three-quarters of maximal (*Decompose 75%*) and half of maximal (*Decompose 50%*). The last configuration allows the user to check the result of applying the regular (monolithic, or '*Decompose 0%*') discovery algorithm in an easy way.

IV. DECOMPOSED REPLAY FRAMEWORK

After presenting a decomposed discovery approach and implementation, we now shift our focus to conformance checking based on replay. The goal of the decomposed replay is to apply an existing *cost-based* replay algorithm on a series of sublogs and subnets instead of on one overall log and one overall net. Like with discovery, the idea is that every subnet contains significantly fewer activities than the overall net. As it is assumed that the sublogs correspond to the subnets, a sublog then also contains significantly fewer activity labels than the overall log does. Under the assumption that the complexity of the replay algorithm is significantly worse than linear in terms of the different activities, the decomposed replay algorithm is expected to finish well before the monolithic replay algorithm.

For this reason, the decomposed replay algorithm first decomposes the overall net using the decomposition approach as described in [2]. Hence, instead of having to *estimate* using all kinds of *heuristics* like the decomposed discovery algorithm needs to do, the decomposed replay algorithm can just derive the maximal subnets from the overall net as provided as input.

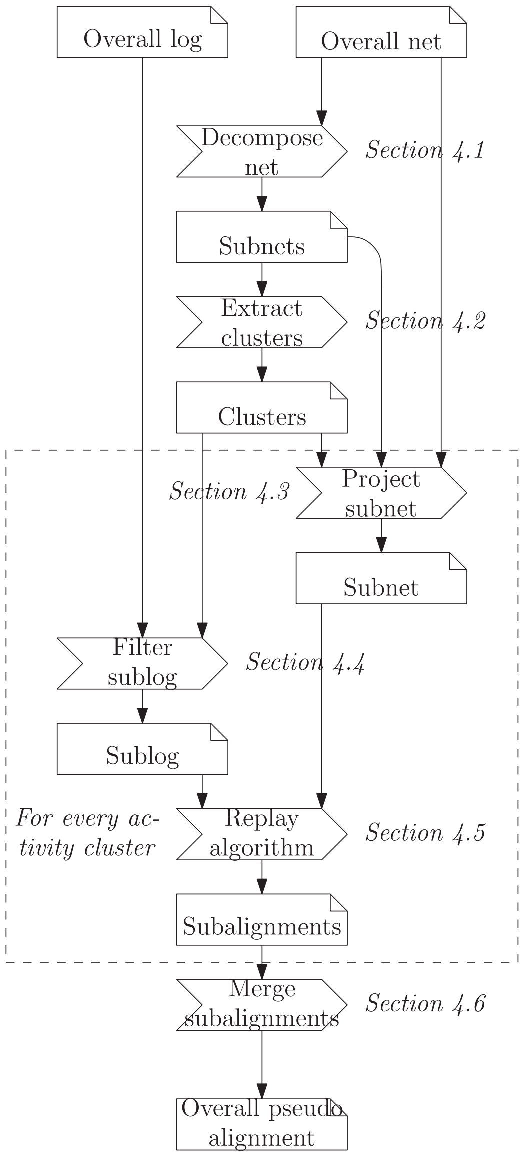


Fig. 25. Conceptual view on a decomposed replay algorithm.

Figure 25 shows a conceptual view on the decomposed replay algorithm. First, the algorithm decomposes the overall net into a series of subnets. Second, it extracts the activity clusters from these subnets. Third, for every activity cluster, the algorithm filters the overall log into a sublog, and it either takes the corresponding subnet from the decomposed subnets or it filters the overall net for a subnet. Fourth, it replays the sublog on the corresponding subnet, which results in a subalignment. Fifth and last, it merges these subalignments into an overall *pseudo*-alignment, where a pseudo-alignment is an alignment except that we drop the requirement that its transition sequence leads from the initial to some final marking. In general, this requirement cannot be guaranteed

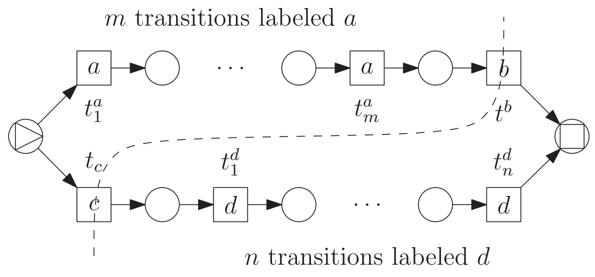


Fig. 26. An example for which merging the subalignments does not result in an overall alignment.

as the example in Figure 26 shows. The dashed line in this net indicates the only way this net can be decomposed into subnets: The first subnet (N_a) contains (among others) all m duplicate transitions labeled a and the second subnet (N_d) contains (among others) all n duplicate transitions labeled d. Now assume that the trace at hand is the empty trace and that every model move costs 10. The optimal alignment for N_a contains only the visible model move $(\gg, t_c)$ (with costs $10/2 = 5$, as t_c is distributed over two subnets [2]), whereas the optimal alignment for N_d contains only the visible model move $(\gg, t_b)$ (also with costs 5). Clearly, there is no alignment in the overall net that has lower costs than the accumulated costs $(5 + 5 = 10)$ of these two model moves: Selecting the upper branch would cost $10 \times m + 10$, selecting the lower branch would cost $10 + 10 \times n$. This example also shows that we cannot give an upper bound for the costs of an overall alignment using the subalignments, as m and n could be arbitrary high. Nevertheless, note that we can still provide a lower bound for the costs.

In the remainder of this section, we introduce each of the abovementioned steps in detail, followed by a description of the implementation of the decomposed replay algorithm in ProM 6.

A. Decompose net

The goal of this step is to decompose the overall net into a series of subnets using the decomposition approach from [2]. This approach decomposes the overall net by first assigning an *equivalence class* to the arcs, where the following restrictions apply:

Places Arcs connected to the same place are equivalent.

Invisible transitions Arcs connected to the same invisible transition are equivalent.

Duplicate visible transitions Arcs connected to all duplicate transitions with the same label are equivalent.

Second, for every equivalence class, it projects the overall net into a subnet that contains (1) all the arcs in this equivalence class and (2) all places and transitions that are connected by these arcs.

Figure 27 shows the resulting subnets ($N_1^a, \ldots, N_1^e$) of this approach on net N_1 (see Figure 4).

B. Extract clusters

The goal of this step is to create a set of activity clusters from the provided subnets. This is done in a straightforward way: For every subnet an activity cluster will be created that contains exactly those activities that are supported by the subnet (that is, for which there is a transition that refers to it). As a result, for subnet N_1^a the activity cluster $\{a_1\}$ will be created, for subnet N_1^b the cluster $\{a_1, a_2, a_3, a_4, a_6)\}$, and so on.

C. Project subnet

The goal of this step is to project an overall net into a collection of subnets, if needed. As input, it takes the overall net, a set of activity clusters, and the subnets that resulted from decomposing the overall net. As output, it delivers a collection of subnets, one net for every activity cluster. The decomposed replay framework supports three possible projections: *Decompose*, *Hide*, and *Hide and reduce*.

1) Decompose: The *Decompose* projection simply takes the corresponding subnet from the subnets that resulted from decomposing the overall net. As such, this projection fully supports the decomposition approach from [2]. The resulting subnets are already shown in Figure 27.

Using this projection, the combination of the initial markings of all subnets corresponds to the initial marking of the overall net. As a result, the initial marking of a subnet is only a subset of the initial marking of the overall net. In the example as shown in Figure 27, the initial marking of subnet N_1^a does correspond to the initial marking of the overall net, but the initial markings of all other subnets are all the empty marking. In case the initial marking covers several places that end up in different subnets, none of the subnets may have the overall initial marking as initial marking. Something similar holds for the final markings.

2) Hide: The *Hide* projection projects the overall net into a subnet by *hiding* visible transitions that are not covered by the cluster at hand. Figure 28 shows the result of this projection on net N_1 and the cluster $\{a_1, a_2, a_3, a_4, a_6\}$. By definition, this projection maintains the structure of the net, as it only hides some transitions. For this reason, the initial and final markings do not change when applying this projection. However, this projection does lead to the duplication of places and invisible transitions, as every subnet will contain its own copy of the silent transition t_6 (called t_6^b in Figure 28). Furthermore, this projection also leads to invisible transitions that correspond to visible transitions in the overall net. Transition t_7^b is an example of such a transition.

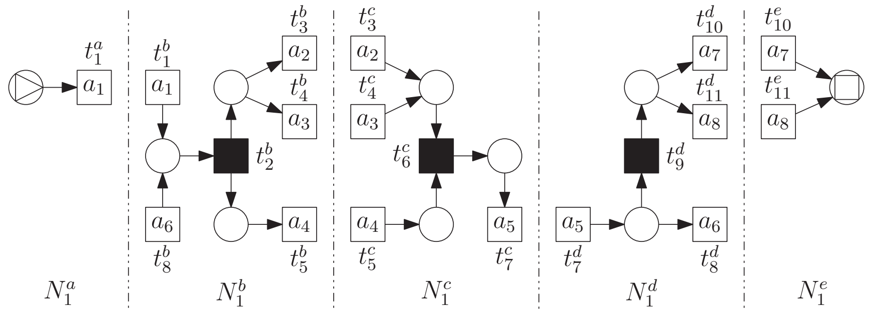


Fig. 27. The result of applying the decomposition approach on net N_1.

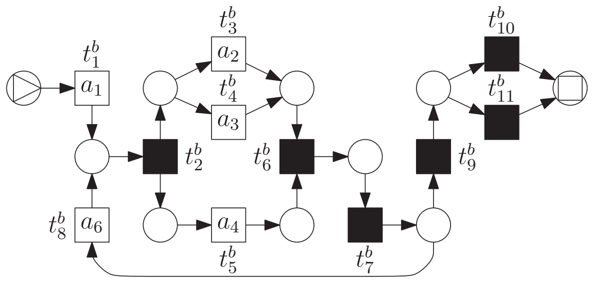


Fig. 28. An example subnet that results from applying the *Hide* projection on net N_1 and cluster $\{a_1, a_2, a_3, a_4, a_6\}$ (cf. subnet N_1^b in Figure 27).

3) Hide and reduce: The *Hide and reduce* projection projects the net into a subnet by *hiding* visible transitions that are not covered by the cluster at hand, and by applying classical behavior-preserving *reduction rules* afterwards. Figure 29

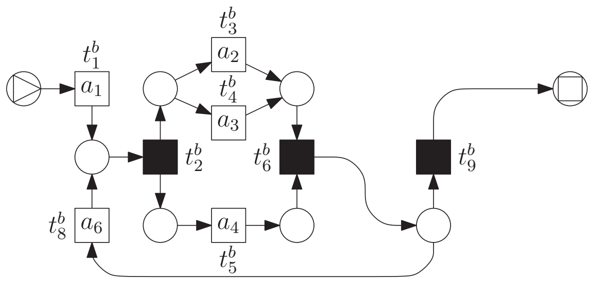


Fig. 29. An example subnet that results from applying the *Hide and reduce* projection on net N_1 and cluster $\{a_1, a_2, a_3, a_4, a_6\}$ (cf. subnet N_1^b in Figure 27).

shows the result of this projection approach on net N_1 and cluster $\{a_1, a_2, a_3, a_4, a_6\}$. Unlike the *Hide* projection, this projection does not maintain the *structure* of the net, as it may remove places, transitions, and arcs. Like the *Hide* projection, this projection maintains the *behavior* of the net, as it only uses behavior-preserving reduction rules. Note that the initial and final markings may need to be updated when using this projection. Like the *Hide* projection, this projection may lead to the duplication of places and invisible transitions, and to invisible transitions that correspond to visible transitions in the overall net.

D. Filter sublog

This step is similar to the step as described in Section III-B. The only exception is that no artificial start and end activities are added in this step, the overall log is just split according to the activity clusters.

E. Replay algorithm

The goal of this step is to replay all sublogs on the corresponding subnets using the provided cost-based replay algorithm, thereby obtaining subalignments. To be able to accumulate the costs from the subalignments into costs for the overall pseudo-alignment later on, we need to change the cost structure for these subreplays. As explained in [2], we divide the costs of a particular activity move (that is, a synchronous move, a log move, or a visible model move) in the alignment by the number of clusters that contain the corresponding activity. As a result, if the subreplays all agree on the same move as the overall replay, then the accumulated costs for the subreplays match the costs of the overall replay. We refer to [2] for the details on this *splitting of costs over clusters*. For this paper, it suffices to mention that the subreplays indeed use such an updated cost structure.

Any replayer algorithm that takes the same set of inputs and that returns a similar alignment with similar costs, can be added with ease to the framework. For other replayers, more work needs to be done, depending on the required inputs or created outputs. The only real requirements on a replayer algorithm (from our framework point-of-view) is that it should be able take a net, a log, and a cost structure, and that it should return an alignment with costs (based on the provided cost structure) associated to every trace.

At the moment, the framework supports the most-used replay algorithm: the cost-based replayer that underlies the existing "Replay a Log on Petri net for Conformance Analysis" plug-in in ProM 6. This plug-in takes (1) a Petri net, (2) a log, (3) a mapping that assigns an activity to every transition (where invisible transitions can be mapped to a special dummy activity), and (4) additional parameters for the cost-based replayer. These additional parameters include the initial marking of the net, the final markings of the net, and the cost structure to be used. The replayer then returns an optimal alignment including (raw fitness) costs for every trace.

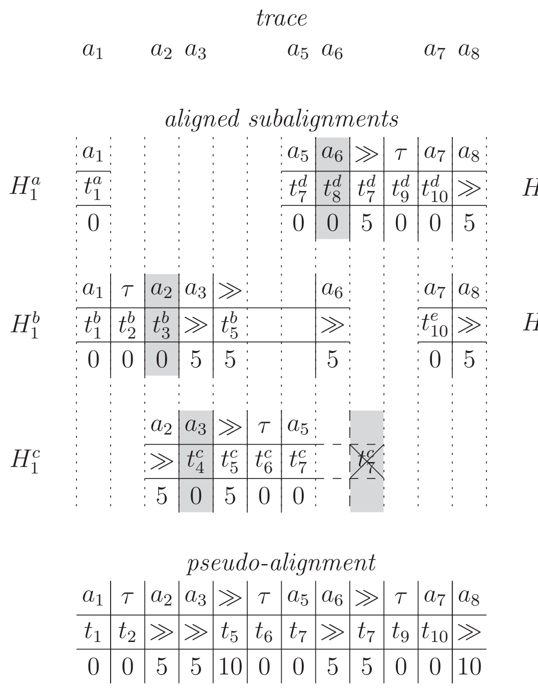
F. Merge subalignments

The goal of this step is to merge the subalignments that resulted from replaying the sublogs on the subnets, and to accumulate the costs of these subalignments. Each trace can be correctly classified as fitting or not in the decomposed approach (that is, exact results). However, because of the updated costs structure, as explained in [2], this accumulated costs will be a *lower bound* for the costs obtained by replaying the overall log on the overall net.

To explain issues at hand for this step, we assume that we need to replay the trace $\langle a_1, a_2, a_3, a_5, a_6, a_7, a_8 \rangle$ on the overall net, that is on net N_1 (see Figure 4). Figure 6 shows an optimal overall alignment for this trace, which shows that the optimal costs for replaying this trace is 40. Figure 30 shows a set of possible optimal subalignments ($H_1^a, \ldots, H_1^e$), obtained by replaying the sublogs on the subnets obtained using the *Decompose* projection ($N_1^a, \ldots, N_1^e$, see Figure 27). Accumulating the costs from these small alignments yields costs 40, which is in accordance with the costs of the overall alignment.

We need to merge these five subalignments into one overall pseudo-alignment. If possible, this pseudo-alignment should be an alignment. To do so, we take the trace and an empty pseudo-alignment, and work our way through the trace (from left to right) and the subalignments while building up the pseudo-alignment:

1) Activity a_1 is covered by two subalignments: H_1^a and H_1^b. Fortunately, both subalignments agree on a synchronous move on a_1, so we add this synchronous move on a_1 and t_1 to the pseudo-alignment and advance both the trace and subalignments H_1^a and H_1^b.
2) Activity a_2 is also covered by two subalignments: H_1^b and H_1^c. However, subalignment H_1^b is not yet ready to accept a_2 as it first needs to do an invisible model move on t_2. Therefore, we first add this invisible model move into the pseudo-alignment and advance the state of subalignment H_1^b. Then, unfortunately, we see that both subalignments disagree on the move on a_2, as H_1^b suggests a synchronous move (on transition t_3^b) while H_1^c suggests a log move. In case of such a conflict, we either take an *optimistic* approach (by selecting the least expensive move) or a *pessimistic* approach (by selecting the most expensive move). In the remainder of this paper, we will use the pessimistic approach, as the optimistic approach tends to mask mismatches by selecting, in case of a conflict, moves without costs. Clearly, when diagnosis is the goal, one should not mask possible problems. So, we add the log move $(a_2, \gg)$ into the pseudo-alignment and advance the state of the trace and both subalignments.
3) Activity a_3 is handled in a similar way as a_2, as both subalignments again disagree. Note that as a result, we now have added two log moves for a_2 and a_3 to the pseudo-alignment, which leads to a transition sequence that is indeed not executable in the overall net, and to a pseudo-alignment which is (by definition) not an alignment.
4) And so on.



a_1	τ	a_2	a_3	$\gg$	τ	a_5	a_6	$\gg$	τ	a_7	a_8
t_1	t_2	$\gg$	$\gg$	t_5	t_6	t_7	$\gg$	t_7	t_9	t_{10}	$\gg$
0	0	5	5	10	0	0	5	5	0	0	10

Fig. 31. Alignment of the subalignments for obtaining the overall pseudo-alignment. Less expensive moves that are ignored in the pseudo-alignment have grey backgrounds.

Figure 31 shows the *alignment of alignments* that results from merging the subalignments. In this figure, the top row shows the activity sequence, that is, the trace, the middle row shows how the subalignments can be aligned properly, and the bottom rows show the resulting overall pseudo-alignment. Moves that have been ignored (because they were in conflict and were less expensive) are indicated with a grey background. Note that his includes transition t_7^c in H_1^c as its absence conflicts with the visible model move on t_7^d in H_1^d. The overall pseudo-alignment follows directly from this alignment of subalignments by taking in every column an optimal move, that is, a move that has no grey background.

A crucial observation for this merging of alignments, is that it requires the invisible transitions to occur in only a single subnet. For the *Decompose* projection, this requirement holds by definition, but this is not the case for the *Hide* and *Hide and reduce* projections. Almost by definition, these projections result in subnets that share invisible transitions, as shown in Figure 28 and Figure 29. For this reason, we first project a subalignment as obtained by these latter two projections on the corresponding subnet as obtained by the *Decompose* projection. The projection of a subalignment on a net keeps only those moves in the subalignment that are related to some transition in the subnet, while all other moves are discarded. As an example, Figure 32 shows a possible optimal alignment for the trace $\langle a_1, a_2, a_3, a_5, a_6, a_7, a_8 \rangle$ on the subnet shown in Figure 28. Projection of this alignment on the corresponding subnet N_1^b (see Figure 27) leads to alignment H_1^b (see Figure 30), as the moves related to the

a_1	
t_1^a	
0	
H_1^a	

a_1	τ	a_2	a_3	$\gg$	a_6
t_1^b	t_2^b	t_3^b	$\gg$	t_5^b	$\gg$
0	0	0	5	5	5

H_1^b

a_2	a_3	$\gg$	τ	a_5
$\gg$	t_4^c	t_5^c	t_6^c	t_7^c
5	0	5	0	0

H_1^c

a_5	a_6	$\gg$	τ	a_7	a_8
t_7^d	t_8^d	t_7^d	t_9^d	t_{10}^d	$\gg$
0	0	5	0	0	5

H_1^d

a_7	a_8
t_{10}^e	$\gg$
0	5

H_1^e

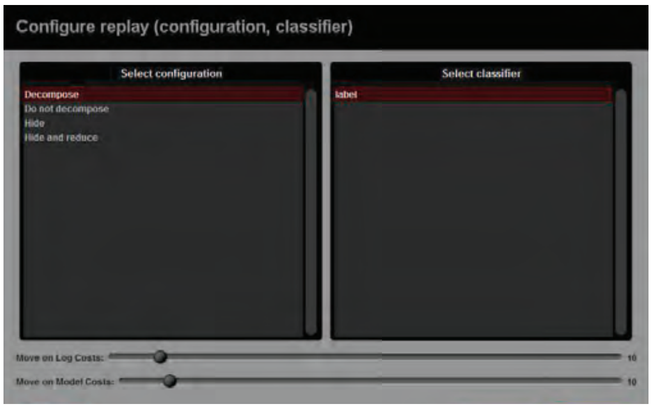
Fig. 30. Possible optimal subalignments using *Decompose* approach.

a_1	τ	a_2	a_3	$\gg$	τ	τ	a_6	τ	τ
t_1^b	t_2^b	t_3^b	$\gg$	t_5^b	t_6^b	t_7^b	$\gg$	t_9^b	t_{10}^b
0	0	0	5	5	0	0	5	0	0

Fig. 32. Possible optimal subalignment using *Replace* projection.

transitions t_6^b, t_7^b, t_9^b, and t_{10}^b will be discarded. After this projection, the alignment can be merged as described above.

G. Implementation



Configure replay (configuration, classifier)

Select configuration
- Decompose
- Do not decompose
- Hide
- Hide and reduce

Select classifier
- label

Move on Log Costs: 10
Move on Model Costs: 10

Fig. 33. First dialog for *Replay using Decomposition* action in ProM 6.

The decomposed replay algorithm has been implemented as the *Replay using Decomposition* action in the *Decompose-dReplayer* package in ProM 6. Figure 33 shows the *first* dialog for this action, which allows the user to select the *configuration* to use (which determines, amongst other things, which net projection to use), which classifier to use, and two sliders to determine the cost structure to be used during the replay. The top slider (labeled *Move on Log Costs*) determines the cost of a log move, that is, the costs of skipping an activity from the log because there is no matching transition in the net. The bottom slider (labeled *Move on Model Costs*) determines the cost of a visible model move, that is, the costs of executing a visible transition in the net while there is no matching activity in the log.

Like with the decomposed discovery, a configuration of the algorithm determines predefined values for the settings of the algorithm. The following configurations can be selected:

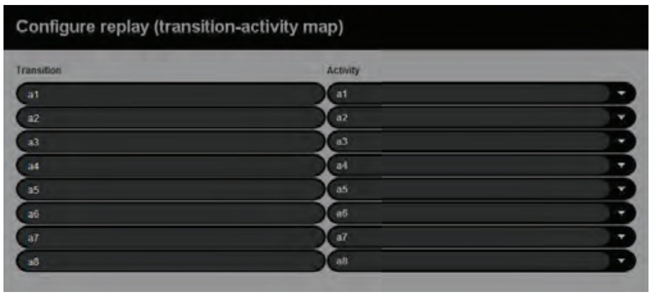
Decompose This configuration uses all steps (as described before) with default values. For decomposing the net, extracting the clusters, filtering the sublogs, and merging the subalignments, no parameters are required. For projecting the net into subnets, the *Decompose* projection is selected. For the replay algorithm, (1) the classifier is as selected by the user, (2) the costs of synchronous moves and invisible model moves are set to 0, while the costs of log moves and visible log moves are set as selected by the user, and (3) a transition label is mapped onto an activity if and only the user has selected so (in the second dialog).

Hide This configuration is identical to the *Decompose* configuration, except that for projecting the net into subnets the *Hide* projection is used.

Hide and reduce This configuration is identical to *Decompose* configuration, except that for projecting the net into subnets the *Hide and reduce* projection is used.

Do not decompose This configuration creates a single subnet and a single sublog. As a result, the replayer is run with the selected classifier, costs, and transition-activity mapping on the original log and net. This configuration corresponds to the monolithic approach, and offers a baseline for comparison.

Like with the decomposed discovery, the *Do not decompose* configuration has been added to be able to run the underlying algorithm in a monolithic setting.



Configure replay (transition-activity map)

Transition	Activity
a1	a1
a2	a2
a3	a3
a4	a4
a5	a5
a6	a6
a7	a7
a8	a8

Fig. 34. Second dialog for *Replay using Decomposition* action in ProM 6.

Figure 34 shows the *second* dialog for this action, which allows the user to select which *transition label* corresponds to which activity. Note that this dialog causally depends on the classifier as set in the first dialog: Changing the classifier in the first dialog will change the available activities in this second dialog. If an activity exists with the same name as the transition label, then this activity will be preselected for the transition label.

As mentioned in Section IV-E, the only replayer implemented in the framework is the cost-based replayer that underlies the existing "Replay a Log on Petri net for Conformance Analysis" plug-in in ProM 6. Our framework imposes an initial deadline of 10 minutes on this replayer (see also [28]). As soon as a single run of this replayer exceeds this deadline, this run is stopped and the deadline for *subsequent* runs is set to 0 minutes. In case of a non-decomposed replay, there are no subsequent runs, but in case of decomposed replay there may be. Assume, for the sake of argument, that the log and net at hand have been decomposed by the framework into five sublogs and subnets, and that the replayer takes (in the given order) 5, 15, 10, 5, and 15 minutes to replay these sublogs on these subnets. The first replay finishes in time, but the second does not. As a result, this second replay fails, which causes the entire decomposed replay to fail. As such, there is no need to spend much time in the subsequent third, fourth, and fifth replay. For this reason, after the second replay has failed, we set the deadline to 0 minutes, which causes these subsequent replays to return as soon as possible.

V. Evaluation

We have evaluated the implemented decomposed discovery and replay algorithms by running them on the data sets as mentioned in Section II-F. This section discusses the results of these experiments. For the decomposed discovery algorithm, the reported computation times for the monolithic discovery (that is, for the *Do not decompose* configuration) include only the computation time needed for the *discovery algorithm* itself (see Figure 9), that is, it excludes the computation times for discovering activity clusters, filtering the overall log, and merging the subnets. For all other configurations, the computation times include all these steps. In a similar fashion, for the decomposed replay algorithm, the reported computation times for the monolithic replay (that is, for the *Do not decompose* configuration) include only the computation time needed for the *replay algorithm* itself (see Figure 25), that is, it excludes the computation times for the creation of the activity clusters, the filtering of both the overall net and the overall log, and the merging of the subalignments. Hence, overhead is only counted for the decomposed approaches and not for the baseline (no decomposition).

For the *DMKD 2006* data set, the results use case labels like *a32f0n10*, where *a32* indicates that this case contains 32 activities, and *n10* indicates that in 10% of the traces noise was introduced. For the *IS 2014* data set, the results use case labels like *59/55/n*, where *59* indicates the reported number of activities in [28], *55* indicates the average trace length, and *n* indicates that this log contains noise. For the *BPM 2013* data set, the results use case labels like *prAm6*, which directly relates to the case from the data set. For the *BPIC* data sets, the result use case labels like *BPIC15_1*, where *15* indicates that this case is from the *BPIC 2015* data set, and *1* indicates that this is the first case (log) from this data set.

All plug-ins used for doing this evaluation are available in the *DivideAndConquerTest* package in ProM 6. This package can be downloaded from https://svn.win.tue.nl/repos/prom/Packages/DivideAndConquerTest/Trunk[3], which is a folder in our Subversion repository.

[3]See the file evaluation/readme.txt in this download for additional details on the evaluation.

All tests are performed on a desktop computer with an Intel Core i7-4770 CPU at 3.40 GHz, 16 GB of RAM, running Windows 7 Enterprise (64-bit), and using a 64-bit version of Java 7 where 4 GB of RAM was allocated to the Java VM. Note that the approach can be distributed over multiple computers, but we only use one computing node.

A. Discovery

We have evaluated the implemented decomposed discovery algorithm using the *ILP Miner*, as this discovery algorithm is known to have a bad complexity in the number of different activities in the log. Although other discovery algorithms are supported by the framework (see also Table XI), in this paper we focus only on the ILP miner.

To evaluate the decomposed discovery algorithm for a single case, that is, for a given event log, a given configuration, and a given discovery algorithm, we perform the following steps:

1) We import the event log. We assume that the first classifier in that event log provides us with the activity log.
2) We run the decomposed discovery algorithm using the given configuration and the given discovery algorithm. Any computation time reported relates only to this step, and not to any of the other steps. In the end, this results in an *accepting Petri net*, which is saved to file.
3) We replay the given event log on the discovered net using the monolithic replay. This provides us with the log alignment needed for the next step.
4) We estimate the *generalization* and *precision* of the log and the net using the *Measure Precision/Generalization* plug-in as available in ProM 6.
5) We output a text file containing the diagnostic results in condensed form.

These steps have been implemented as the *Evaluate Decomposed Discovery* plug-in.

First, we compare the monolithic discovery algorithm (as implemented by the *Do not decompose* configuration, see Section III-E) with the maximal-decomposition discovery algorithm (as implemented by the *Decompose* configuration). Second, we compare the maximal-decomposition discovery algorithm with both the non-maximal-decomposition discovery algorithms (as implemented by the *Decompose 75%* configuration and the *Decompose 50%* configuration).

1) Monolithic vs. Maximal-decomposition: First, we show for which logs in the data sets both configurations are feasible. Second, we show the computation times required by both configurations, and compare them where possible. Next, we provide results for the precision and generalization metrics for both configurations. Note that, as we are using the ILP miner, fitness is guaranteed to be 1, so we do not discuss the fitness metric here. To compute precision and generalization, we need to replay the log on the discovered net *without using decomposition*. Therefore, third, we show for which logs this replay was feasible. Fourth, we show the feasible precision values obtained by both configurations, and compare them where possible. Fifth, we do the same for generalization. Finally, we summarize our findings.

a) Feasibility: The *Do not decompose* configuration (simply called *Do not decompose* henceforth) is feasible for all logs in the *DMKD 2006* data set, all logs in the *IS 2014* data set, for the *prAm6*, *prBm6*, *prCm6*, and *prEm6* logs from the *BPM 2013* data set, and for the only log in the *BPIC 2012* data set. *Do not decompose* runs out of memory for the *prDm6* and *prFm6* logs, while it runs out of time for all other infeasible logs (that is, they were stopped after a week).

The *Decompose* configuration (simply called *Decompose* henceforth) is feasible for all logs in the *DMKD 2006* data set, all logs in the *IS 2014* data set, for the *prAm6*, *prBm6*, *prCm6*, *prEm6*, and *prGm6* logs from the *BPM 2013* data set, for the only log in the *BPIC 2012* data set, and for all logs in the *BPIC 2015* data set. Like *Do not decompose*, *Decompose* runs out of memory for the *prDm6* and the *prFm6* logs.

As a result, we can only compare computation times for those logs that are feasible with *Do not decompose*.

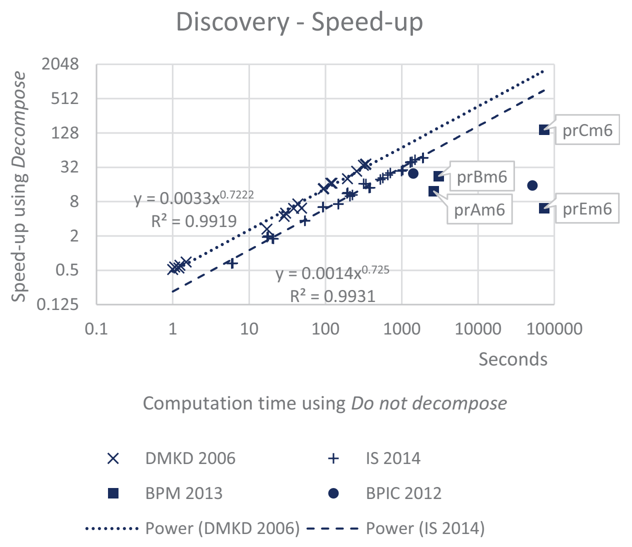


Fig. 35. Comparison of feasible computation times. The speed-up by decomposition tends to be high when computation times are long.

b) Computation times: Figure 35 shows the feasible computation times for *Do not decompose*, and the speed-ups obtained by using *Decompose*. For example, this figure shows that *Do not decompose* took almost 75.000 seconds (more than 20 hours) to discover a net from the *prCm6* log, and it also shows that *Decompose* is about 150 times as fast, needing only about 500 seconds (less than 10 minutes). *Decompose* outperforms *Do not decompose* for all cases where the latter takes more than ten seconds.

Figure 35 clearly shows that the speed-up obtained by *Decompose* depends on the computation time of *Do not decompose*. This is especially clear for the *DMKD 2006* and *IS 2014* data sets: The higher the computation time needed by *Do not decompose*, the higher the speed-up of *Decompose*. The figure finally shows that the speed-up also depends on the data set the log originates from. For example, the speed-up for a log from the *DMKD 2006* data set is typically higher than the speed-up for a log from the *IS 2014* data set. This is surprising, as we believed that both data sets contained logs with a similar complexity.

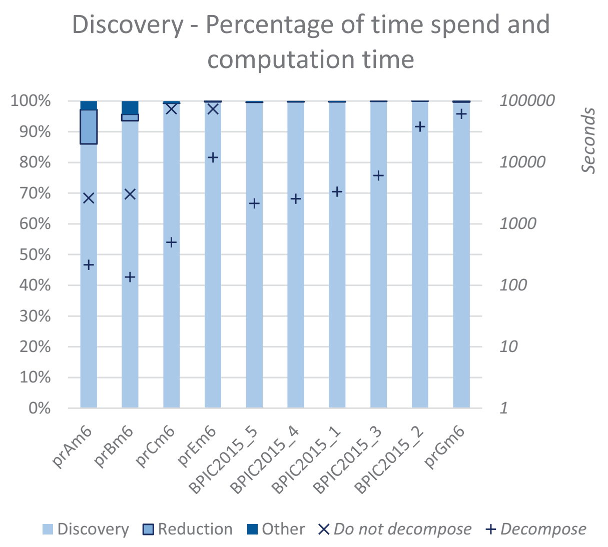


Fig. 36. Categorized percentages of computation times for the most-hard feasible cases together with their computation times.

Figure 36 shows, for the 10 most time-consuming logs, the feasible computation times for both configurations, and also where time was spend. First, time can be spend on the discovery of the subnets, that is, on running the discovery algorithm on the sublogs. Figure 36 shows the percentage of time spend on this (see the bottom-most bars, labeled *Discovery*). Second, time can be spend in the reduction of the discovered overall net, which is shown using the middle bars, labeled *Reduction*. Clearly, *Decompose* spends the majority of its time (more than 86% for the cases shown in Figure 36, more than 83% for all feasible cases) in the decomposed discovery, only a fraction is spend on the overhead of the decomposition approach. This shows that there is no urgent need to improve on, for example, the reduction of the net, as the entire approach would hardly benefit from this.

Figure 36 also shows the computation times using *Decompose* for those logs for which *Do not decompose* was infeasible. For example, it took *Decompose* 2167 seconds (about 36 minutes) to discover a net from the *BPIC2015_5* log. Given the fact that *Do not decompose* for this log was stopped after a week, the speed-up of *Decompose* for this log is at least 280.

c) Feasibility of monolithic replay: The monolithic replay on the nets discovered using *Do not decompose* is feasible for all logs in the *DMKD 2006* data set, all logs in the *IS 2014* data set, the only log in the *BPIC 2012* data set, and for the *prAm6* and *prBm6* logs in the *BPM 2013* data set. The monolithic replay runs out of time (that is, it takes more than 10 minutes) for the *prCm6* and *prEm6* logs.

The monolithic replay on the net discovered using *Decompose* is feasible for all logs in the *DMKD 2006* data set, all logs in the *IS 2014* data set, the *prCm6*, *prDm6*, *prFm6* logs in the *BPM 2013* data set, the only log in the *BPIC 2012* data set, and the *BPIC2015_3* and *BPIC2015_4* logs in the *BPIC*

2015 data set. The monolithic replay runs out of memory for the *BPIC2015_2* log, and runs out of time for all remaining logs.

As a result, we can only compare precision and generalization for all logs in the *DMKD 2006* data set, all logs in the *IS 2014* data set, and the log in the *BPIC2012* data set.

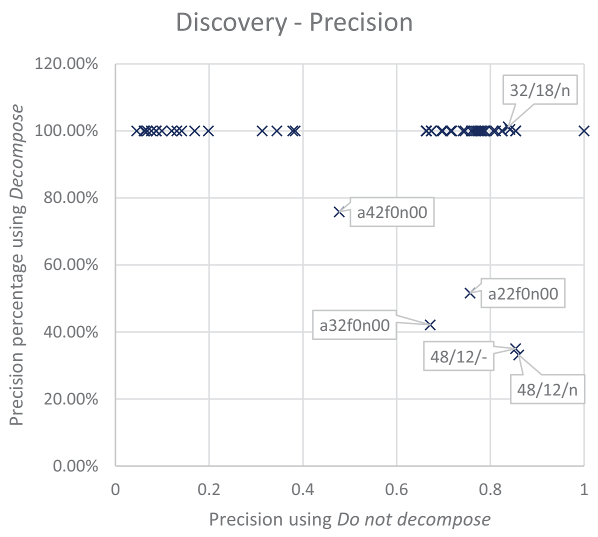


Fig. 37. Comparison of precision metrics on all feasible data sets.

d) Precision: Figure 37 shows the precision values obtained using *Do not decompose*, and the percentage of precision as obtained using *Decompose*. As an example, the precision obtained with *Do not decompose* on the *48/12/n* log is about 0.86, and *Decompose* results in a precision of about 33% of 0.86 (about 0.28). This figure also shows that, in general, *Decompose* results in the same or less precision as *Do not decompose*. The only exceptions to this are the *32/18/n* log (101.23%) and the *32/18/-* log (100.42%).

Figure 38 shows why precision can be lower when using *Decompose*. The net that is discovered with *Decompose* contains three source transitions (transitions without incoming arcs), which are always enabled. As these transitions are enabled in all possible states, but only executed in few states, this net is less precise.

In contrast, Figure 39 shows that precision can also be higher when using *Decompose*. The net that is discovered with *Decompose* contains two additional source places (places without incoming arcs), which are initially marked. One of these places effectively prevents the transition labeled *I2+complete* from being executed more than once, which is possible in the net discovered by *Do not decompose*, but which does not occur in the log. As a result, the net discovered with *Decompose* is more precise.

e) Generalization: Figure 40 shows the generalization values obtained using *Do not decompose*, and the percentage of precision as obtained using *Decompose*. This figure shows that, in general, *Decompose* results in a better generalization than *Do not decompose*, although the differences are typically very small.

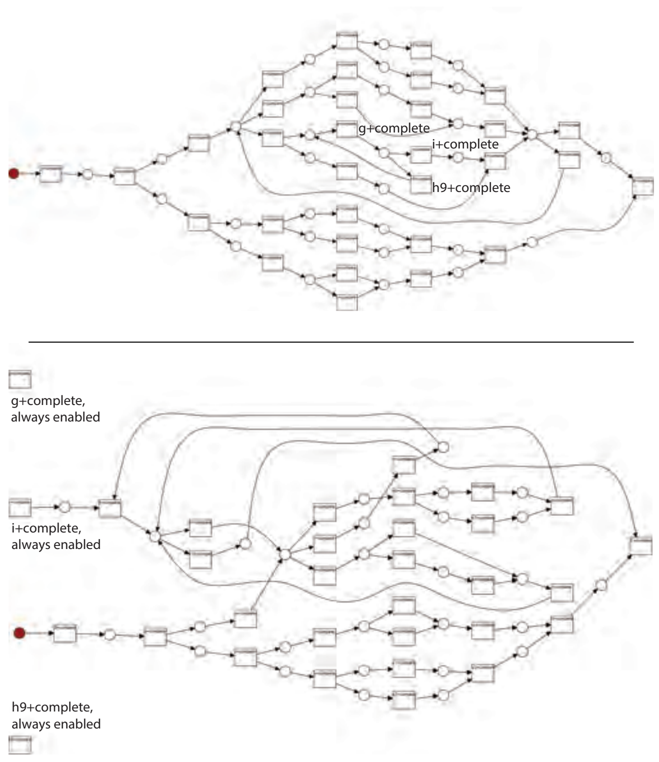

g+complete
i+complete
h9+complete
g+complete, always enabled
i+complete, always enabled
h9+complete, always enabled

Fig. 38. Example *a32f0n00* explaining why precision can be lower when using *Decompose*. The top net is the result from *Do not decompose*, the bottom net from *Decompose*.

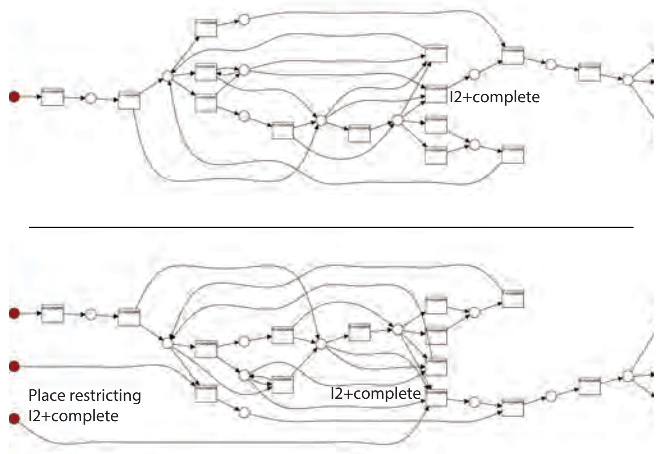


Fig. 39. Example *32/18/n* explaining why precision can be higher when using *Decompose*. The top net is the result from *Do not decompose*, the bottom net from *Decompose*.

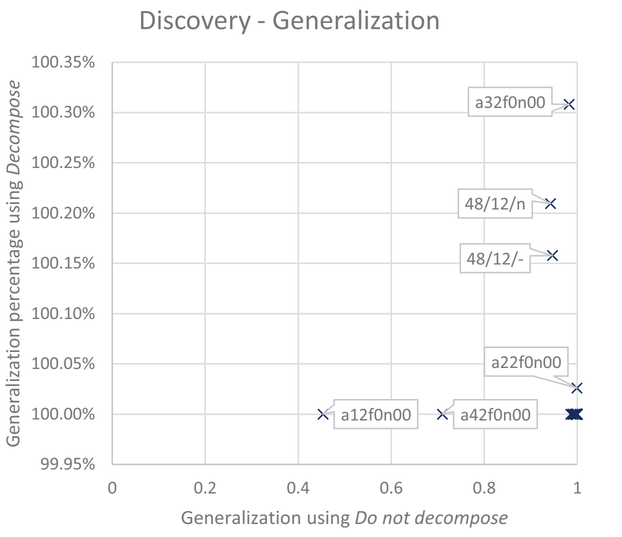


Fig. 40. Comparison of generalization metrics on all feasible data sets. *Decompose* results in very similar generalization values.

f) Conclusions: If the monolithic discovery algorithm (that is, *Do not decompose*) can discover a net from a log, then the decomposition discovery algorithm (that is, *Decompose*) can also discover a net from this log. However, the decomposition algorithm can also discover nets from logs on which the monolithic algorithm fails. As such, the decomposition algorithm can be applied on much larger and more complex logs than the monolithic algorithm.

The decomposition algorithm is typically faster than the monolithic algorithm. If the monolithic algorithm takes more than 100 seconds, then the speed-up is at least 7.5, but can be more than 100.

The decomposition algorithm typically results in nets that have an equal or worse value for precision, where the latter is typically due to the introduction of additional source transitions. However, it is also possible that the decomposition algorithm results in a slightly higher precision, as a result of the introduction of additional initially marked source places.

The decomposition algorithm typically results in a net that has an equal or better value for generalization, although the improvements are minor.

2) Maximal-decomposition vs Non-maximal-decomposition: First, we show for which logs all three configurations are feasible. Second, we show the computation times required by *Decompose* and the speed-ups obtained using the other two configurations. Third, we show for which logs (and discovered nets) the monolithic replay was feasible, as again this is needed to compute the precision and generalization. Fourth, we show the precision values obtained using *Decompose* and the percentages obtained by the other two configurations. Fifth, we do the same for generalization. Finally, we summarize our findings.

a) Feasibility: The *Decompose 75%* configuration (simply called *Decompose 75%* henceforth) and the *Decompose 50%* configuration (simply called *Decompose 50%* henceforth) are feasible for exactly the same set of logs as *Decompose*, that is, they only fail for the *prDm6* and *prFm6* logs (by also running out of memory).

As a result, we can compare computation times for all logs that are feasible with *Decompose*.

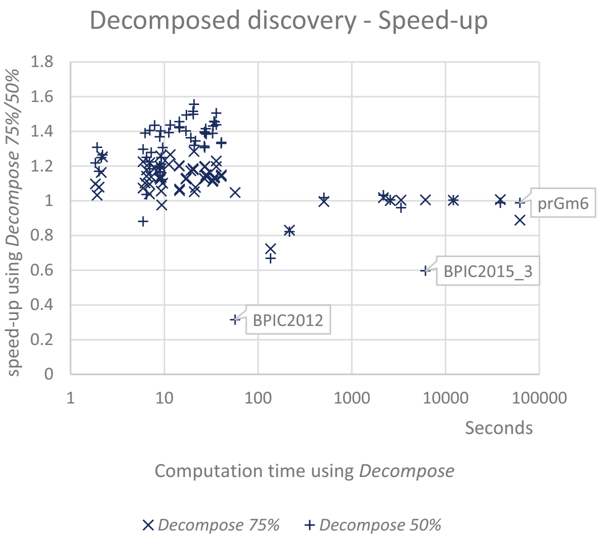


Fig. 41. Comparison of decomposed computation times.

b) Computation times: Figure 41 shows the computation times required by *Decompose*, and the speed-ups obtained using *Decompose 75%* and *Decompose 50%*. As an example, it takes *Decompose* about 210,000 seconds (about 58 hours) to discover a net from the *prGm6* log, and the speed-up of *Decompose 50%* is about 1.09, resulting in a required computation time of about 193,000 seconds (about 54 hours). For the easier cases, *Decompose 50%* outperforms *Decompose*, but for the harder cases there seems to be no improvement. In fact, there are two cases (*BPIC2012* and *BPIC2015_3*) for which *Decompose 50%* needs significantly more time than the other two configurations.

On average, *Decompose 50%* requires about 96% of the time required by *Decompose*, and *Decompose 75%* requires about 98%. Nevertheless, the differences between the configurations seem only minor, especially when compared to the difference with *Do not decompose*.

c) Feasibility of monolithic replay: The monolithic replay on the nets discovered using *Decompose 75%* is feasible for all logs in the *DMKD 2006* data set, all logs in the *IS 2014* data set, the *prCm6* log in the *BPM 2013* data set, the only log in the *BPIC 2012* data set, the only log in the *BPIC 2012* data set, and the *BPIC2015_1*, *BPIC2015_3*, and *BPIC2015_4* logs in the *BPIC 2015* data set. The monolithic replay runs out of memory for the *BPIC2015_2* log and out of time for all remaining logs.

The monolithic replay on the nets discovered using *Decompose 50%* is feasible for exactly the same set of logs as for which *Decompose 75%* is feasible. Again, the monolithic replay runs out of memory for the *BPIC2015_2* log, while it runs out of time for all remaining logs.

d) Precision: Figure 42 shows the precision values obtained using *Decompose*, and the percentages using *Decom-*

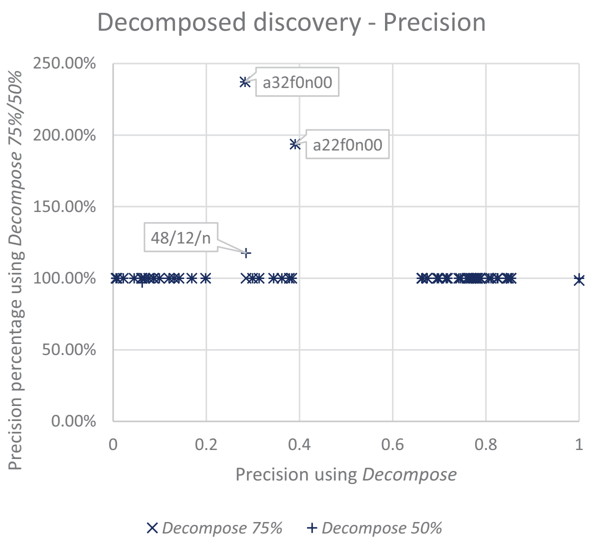


Fig. 42. Comparison of decomposed precision metrics.

pose 75% and *Decompose 50%*. As an example, the precision value for the *a32f0n00* case as obtained using *Decompose* is about 0.28, and the percentages for the two other configurations are about 237%, resulting in a precision value of about 0.67 (that is, the same value as obtained by *Do not decompose*). Apparently, both *Decompose 75%* and *Decompose 50%* were able to avoid the introduction of the additional source transitions for these cases. Still, for some other cases (*a42f0n00*, *48/12/-*, and *48/12/n*), these configurations did not improve precision to the same level as *Do not decompose*. Possibly, we need an even more coarse-grained decomposition (like 25%) to get the same precision.

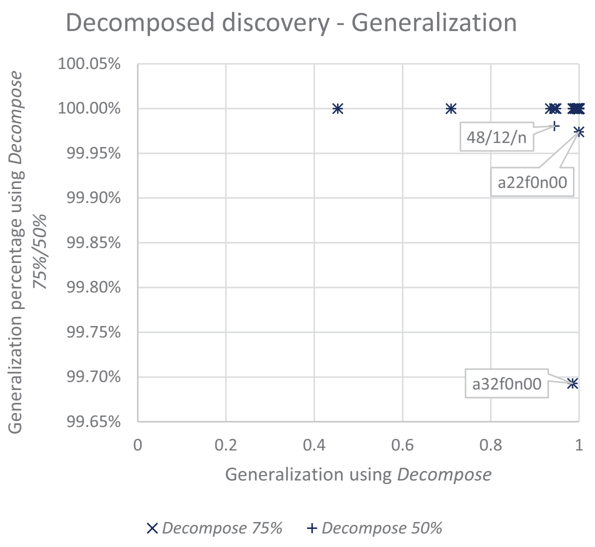


Fig. 43. Comparison of decomposed generalization metrics.

e) Generalization: Figure 43 shows the generalization values obtained using *Decompose*, and the percentages using *Decompose 75%* and *Decompose 50%*. As an example, the precision value for the *a32f0n00* case as obtained using *Decompose* is about 0.98, and the percentages for the both other configurations are about 99.7%, resulting in a precision value of about 0.98.

f) Conclusions: The non-maximal decomposition discovery algorithms (that is, *Decompose 75%* and *Decompose 50%*) can discover nets from the same set of logs that the maximal decomposition algorithm (*Decompose*) can. On average, the 50% decomposition algorithm takes a bit more time (104%) than the maximal decomposition algorithm, and the 75% decomposition algorithm takes also a bit more (105%). For the cases that take less than 100 seconds, the 50% decomposition algorithm is the fastest, but it is considerably slower for some cases that require more time, like the *BPIC2012* and the *BPIC2015_3* cases. Apparently, for these cases the 50% decomposition algorithm results in sublogs that are harder to handle for the ILP Miner than the sublogs for the other decomposition algorithms. The non-maximal decomposition algorithms result in equal or better precision values. Sometimes the precision values obtained match the ones obtained using the monolithic algorithm (which is perfect). The non-maximal decomposition algorithms result in equal or worse generalization values, but if worse the difference is only minor.

B. Replay

To evaluate the decomposed replay algorithm for a single case, that is, for a given combination of a log and a net, we perform the following steps:

1) We import the given event log. As before, we assume that the first classifier in that event log provides us with the activity log.
2) We import the accepting Petri net provided. We assume that a transition in this net is related to an activity in the log if and only if the label of the transition matches the activity perfectly.
3) We run the decomposed replay algorithm using the given configuration. Any computation time reported relates only to this step, and not to any of the other steps. In the end, this results in a (pseudo)-alignment with replay costs.
4) We output a text file containing the diagnostic results in condensed form.

These steps have been implemented as the *Evaluate Decomposed Replay* plug-in.

As the *BPI 2012* and *BPI 2015* data sets did not include process models, these data sets could not be used in this evaluation.

First, we compare the monolithic replay algorithm (as implemented by the *Do not decompose* configuration, see Section IV-G) with the decomposed replay algorithm (as implemented by the *Decompose* configuration). Second, we compare the monolithic replay algorithm with the alternative replay algorithms, that is, we compare the *Do not decompose* configuration with the *Hide* configuration and the *hide and reduce* configuration.

1) Monolithic vs. Decomposition: First, we show for which logs and nets in the data sets both replay configurations are feasible. Second, we show the computation times needed by both configurations, and compare them where possible.

Next, we show the replay cost values as obtained by both configurations. Finally, we summarize our findings.

a) Feasibility: The *Do not decompose* configuration (simply called *Do not decompose* henceforth) is feasible for all cases in the *DMKD 2006* data set, all cases in the *IS 2014* data set, and the *prAm6* and *prBm6* cases from the *BPM 2013* data set. *Do not decompose* runs out of time (that is, it does not finish in 10 minutes) for all remaining cases from the *BPM 2013* data set.

The *Decompose* configuration (simply called *Decompose* henceforth) is feasible for all cases in all data sets.

As a result, we can only compare the computation times and costs for those cases that were feasible for *Do not decompose*.

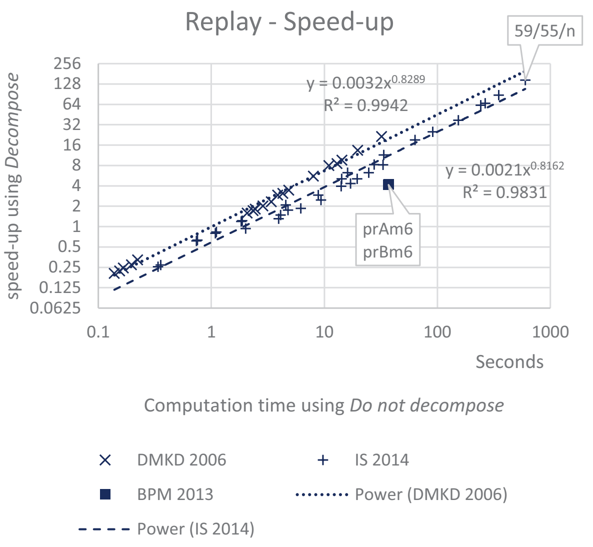


Fig. 44. Comparison of feasible computation times.

b) Computation times: Figure 44 shows the feasible computation times using *Do not decompose*, and the speed-ups obtained by using *Decompose*. For example, this figure shows that *Do not decompose* took about 599 seconds (almost 10 minutes) to replay the *59/55/n* log on its net, and it also shows that *Decompose* is 146 times as fast, requiring only 4 seconds. For all logs considered, *Decompose* outperforms *Do not decompose* for those cases where the latter takes more than 3 seconds.

Figure 44 clearly shows that the speed-up obtained by *Decompose* depends on the computation time of *Do not decompose*. This is especially clear for the *DMKD 2006* and *IS 2014* data sets: The higher the computation time needed by *Do not decompose*, the higher the speed-up of *Decompose*. For the *BPM 2013* this relation is not obvious, as it contains only two points which are almost on top of each other. The figure finally shows that, like with discovery, the speed-up also depends on the data set the case originates from. For example, the speed-up of a case from the *DMKD 2006* data set is typically higher than the speed-up for a case from the *IS 2014* data set. Again, we find this surprising.

Figure 45 shows, for the 10 most time-consuming cases, the computation times for both configurations, and also where time was spend. First, time can be spend on the replay of the

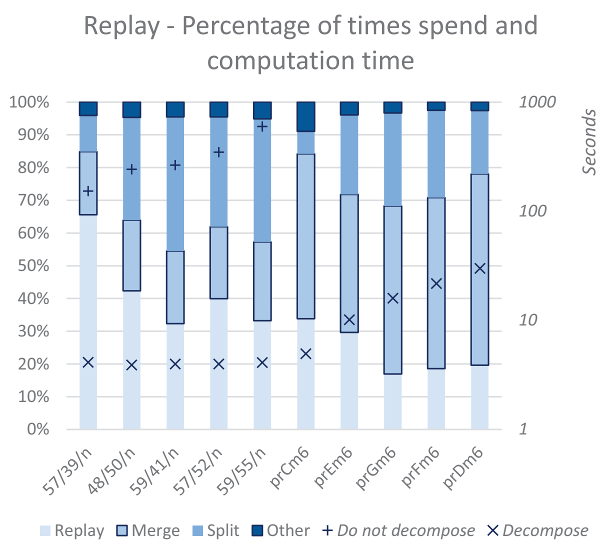


Fig. 45. Categorized percentages of computation times for the most-hard feasible cases together with their computation times.

sublogs on the subnets, that is, on running the replay algorithm on the sublogs and subnets. Figure 45 shows the percentage of time spend on this using the bottom-most bars, labeled *Replay*. Second, time can be spend in splitting the log (the bars labeled *Split log*) and merging the alignments (the bars labeled *Merge*). The percentage of merge time needed for the decomposition approach in these cases ranges from about 18% (*prGm6*) to about 65% (*57/39/n*). On average, on these cases *Decompose* spends 47% of its time on merging the alignments, 25% on the actual replay, and 24% on splitting the logs. This indicates that by improving the alignment merge and/or the log split, we would improve the decomposition approach considerably.

Figure 45 also shows the computation times using *Decompose* for those cases for which were infeasible with *Do not decompose*. For example, it took *Decompose* 5 seconds to replay the *prCm6* log on its net. Given the fact that *Do not decompose* for this case did not finish in 10 minutes, the speed-up of *Decompose* for this case is at least 120.

c) Replay costs: For many of the cases in the data sets, the replays costs are 0 as the log and net are perfectly fitting. This is the case for all cases from the *DMKD 2006* data set that end with *00* (like *a32f0n00*), for all the cases from the *IS 2014* data set that end with - (like *32/34/-*), with all the cases from the same data set that end with *n* but have the lowest average trace length for the given set of activities (like *59/17/n*), and for the *prBm6* case from the *BPM 2013* data set.

Recall that the decomposition replay approach *guarantees* that it yields costs 0 if and only if the monolithic replay approach yields costs 0, and that the decomposition approach never yields higher costs than the monolithic approach. As a result of the former, we will only compare the cases with non-zero costs.

Figure 46 shows the non-zero costs values obtained using both *Do not decompose* and *Decompose*. For example, for the *57/52/n* case, the costs value obtained using *Do not decompose*

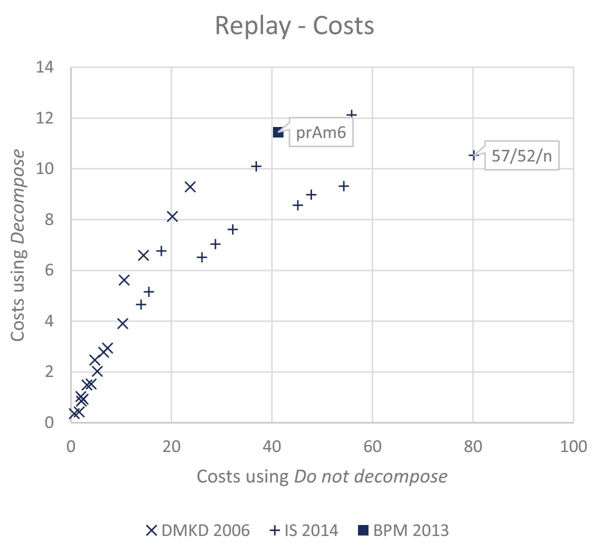


Fig. 46. Comparison of replay costs.

is about 80, while the costs obtained by *Decompose* is just above 10.

Figure 47 explains this difference. The top part in this figure shows the part of the *57/52/n* net where *Do not decompose* detects mismatches, whereas the bottom part shows the two subnets of this net where *Decompose* detects mismatches. Clearly, each of the two subnets individually allows for more behavior than the part of the net, which explains why *Decompose* can report less costs. However, note that *Decompose* clearly indicates the same transitions to be involved in the mismatches: the transitions *AR*, *AU*, *AV*, and *AZ*. Even better, using *Decompose*, it is straightforward to conclude that the noise as contained in this log was introduced by (1) swapping the occurrences of *AR* and *AU* and (2) swapping the occurrences of *AV* and *AZ*, a fact that is not so obvious when using *Do not decompose*. Therefore, *Decompose* can very well be used to diagnose the same mismatches, and can do so in less time.

d) Conclusions: If the monolithic replay algorithm (that is, the *do not decompose* configuration) can replay a log on its net, then the decomposition replay algorithm (that is, the *Decompose* configuration) can also replay this log on its net. However, the decomposition replay algorithm can also replay logs on nets on for which the monolithic replay algorithm fails. In fact, the decomposition replay algorithm was able to replay all logs on their nets from all data sets. As such, this algorithm can be applied on more cases than the monolithic replay algorithm.

The decomposition replay algorithm is typically faster than the monolithic replay algorithm. If the monolithic algorithm takes more than 10 seconds, then the speed-up is at least 4, but can be more than 100. A point of attention for the decomposition replay algorithm is the fact that the required merge of the alignments and the required split of the log may take significant time. In the data sets, there were examples where the decomposition algorithm took 30 seconds, of which only 6 seconds were used for the actual replay, and more than 15 seconds were used on merging the alignments. As a result, we aim to improve on the merging of alignments and the splitting of logs.

The decomposition algorithm on a log and net that are not perfectly fitting results in costs that are quite a bit lower than the costs as obtained using the monolithic algorithm. However, although not the same mismatches are detected by the decomposition algorithm, it will detect similar mismatches, which can be used to diagnose why a log and a net do not match perfectly.

2) Monolithic vs. Alternative: Next, we compare the *Do not decompose* configuration to the two alternative replay configurations: the *Hide* configuration and and the *Hide and reduce* configuration. Recall that the difference between these two configurations and the *Decompose* configuration lies in the way they project the overall net into subnets. The *Decompose* configuration uses the *Decompose* projection approach (see Section IV-C1), the *Hide* configuration uses the *Hide* projection approach (see Section IV-C2), and the *Hide and reduce* configuration uses the *Hide and reduce* projection approach. First, we show for which cases from the data sets both alternative replay configurations are feasible. Second, we show the computation times needed by both alternative configurations, and compare them to the computation times as needed by the *Do not decompose* configuration where possible. Third, we show the replay cost values as obtained by the alternative configurations, and compare them to the costs as obtained by the *Do not decompose* configuration where possible. Finally, we summarize our findings.

a) Feasibility: The *Hide* configuration (simply called *Hide* henceforth) is feasible for all logs and nets in the *DMKD 2006* data set, all logs and nets in the *IS 2014* data set, and the *prAm6*, *prBm6*, and *prCm6* logs and nets in the *BPM 2013* data set. *Hide* runs out of memory for the *prDm6* case, while it runs out of time (10 minutes for a single replay) for all remaining cases from the *BPM 2013* data set.

The *Hide and reduce* configuration (simply called *Hide and reduce* henceforth) is feasible for all cases from all data sets.

As a result, we can only compare the computation times and costs for those cases that are feasible for *Do not decompose*.

b) Computation times: Figure 48 shows the computation times required by *Do not decompose*, and the speed-ups obtained by using *Hide* and *Hide and reduce*. For example, the figure shows that it took *Do not decompose* about 599 seconds (almost 10 minutes) to replay the *59/55/n* log on its net, and it also shows that *Hide and reduce* is about 96 times as fast. In contrast, for this case, *Hide* is about 6 times as slow as *Do not decompose*. When we compare Figure 48 with Figure 44, then we observe that *Hide* and *Hide and reduce* do not offer any advantages over *Decompose* when it comes to computation times, as they are always slower than that configuration. Typically, *Hide* is a lot slower, and even slower than *Do not decompose*, while *Hide and reduce* is only a bit slower. However, like *Decompose*, *Hide and reduce* is faster than *Do not decompose*.

Figure 48 also shows that, in general, the speed-up as obtained by *Hide and reduce* depends on the computation time of *Do not decompose*: The higher the computation time, the

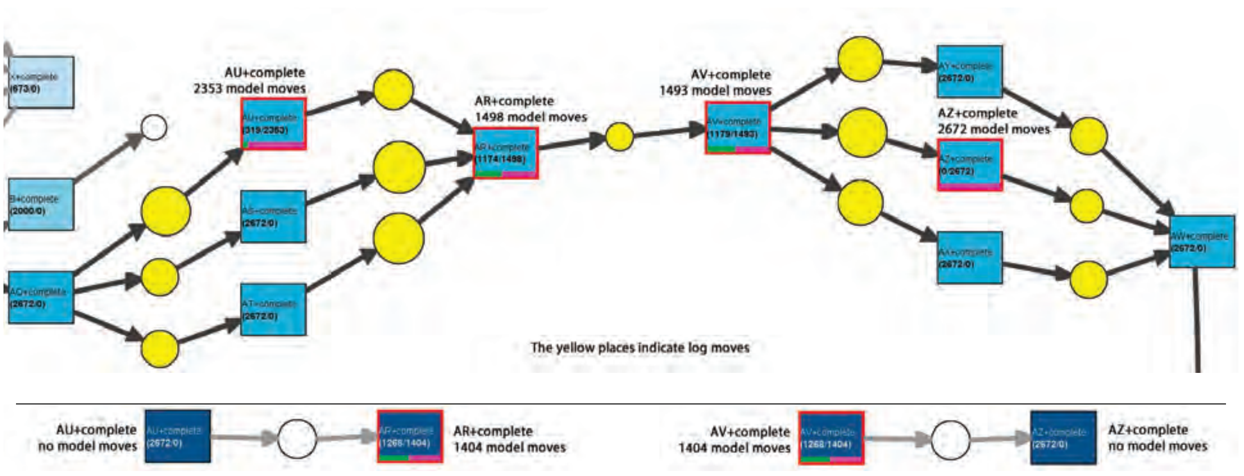


Fig. 47. Example *57/52/n* showing that the decomposition approach diagnoses the same mismatches.

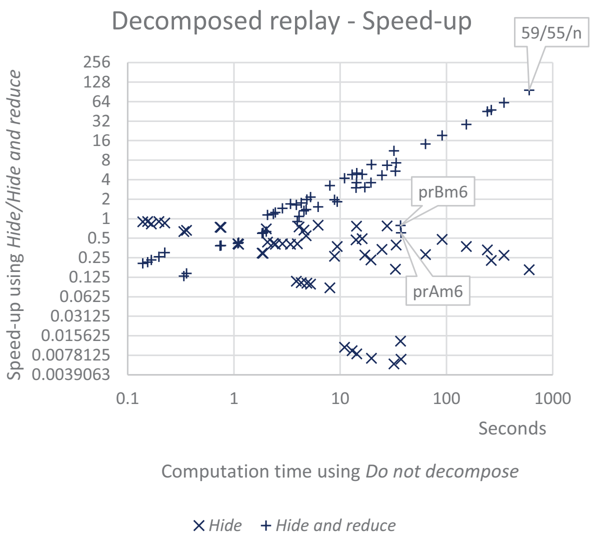


Fig. 48. Comparison of computation times.

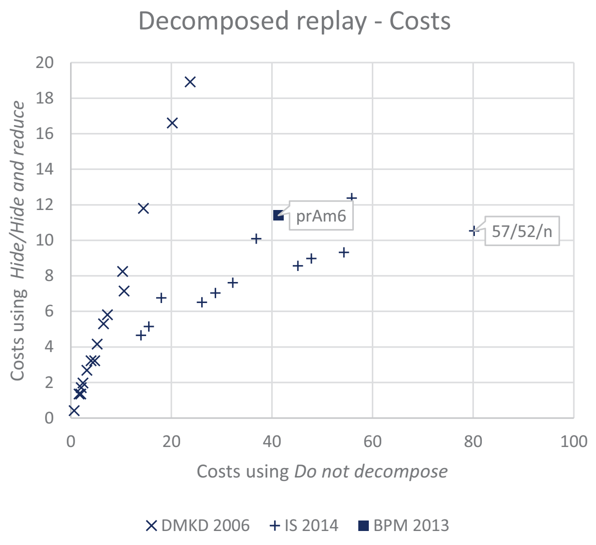


Fig. 49. Comparison of replay costs.

higher the speed-up. In contrast, the speed-up of *Hide* gets worse when computation time of *Do not decompose* goes up.

c) Replay costs: Figure 49 shows the non-zero costs values as obtained by *Do not decompose* and both *Hide* and *Hide and reduce* (as *Hide* and *Hide and reduce* always return the same costs on these data sets, we only show one). Note the resemblance of this figure with Figure 46: Basically, the only significant change involves the data series for the *DMKD 2006* data set. For this data set, the costs as obtained by *Hide* and *Hide and reduce* are higher than with *Decompose*, which is good (the higher, the better).

d) Conclusions: In this section, we have compared the monolithic replay algorithm (that is, *Do not decompose*) with two alternative replay algorithms: A first that uses the *Hide* projection (that is, *Hide*) and a second that uses the *Hide and reduce* projection (that is, *Hide and reduce*). Like the decomposition algorithm (that is, *Decompose*), both alternative algorithms can replay all cases from all data sets. The first alternative algorithm can only replay those cases that the monolithic algorithm can replay, and one more: the *prCm6* case.

The first alternative algorithm is typically much slower than the monolithic algorithm. However, the second alternative algorithm is typically faster. If the monolithic algorithm takes more than 10 seconds, then the speed-up of using the second alternative algorithm is at least 3, but it can be about 100. When compared to the decomposition algorithm, the second alternative algorithm is on average twice as slow.

Using the alternative algorithms on a log and net that are not fitting results in costs that are also quite a bit lower then costs as obtained using monolithic algorithm. For the *IS 2014* and *BPM 2013* data sets, these costs are similar to the costs as obtained by the decomposition algorithm. However, for the *DMKD 2006* data set, the costs are higher than the costs obtained by the decomposition algorithm. In general, the alternative algorithms result in costs that are somewhere in-between the costs obtained by the decomposition algorithm and the costs obtained by monolithic algorithm.

VI. Conclusions

This paper presents the *Divide and Conquer* framework. This framework fully supports the decomposed process mining as introduced in [2], and has been implemented in ProM 6. As such, the framework allows for (1) easy decomposed discovery, using existing discovery algorithms, and (2) easy decomposed replay, using the cutting-edge cost-based replayer. The current framework supports six discovery algorithms, but can easily support more.

For the decomposed discovery, the framework allows the end user to select the classifier to use (which maps the event log at hand to an activity log), the miner (or discovery algorithm) to use, and a *configuration* to use. Available configurations include *Do not decompose* (monolithic discovery algorithm), *Decompose* (maximal decomposition discovery algorithm), *Decompose 75%* (75% decomposition discovery algorithm), and *Decompose 50%* (50% decomposition discovery algorithm). The selected level of decomposition (maximal, 75%, or 50%) determines the number of sublogs to overall log will be split into. For the maximal decomposition, this number will be maximal. Whatever classifier, miner, and configuration the user selects, the end result will be an overall net discovered for the log at hand.

For the decomposed replay, the framework allows the user to select the classifier to use, and a *configuration* to use (as there is only one replayer supported at the moment, there is no need to select a replayer). Available configurations include *Do not decompose* (monolithic replay algorithm), *Decompose* (decomposition replay algorithm), *Hide* (replay algorithm that uses only hiding), and *Hide and reduce* (replay algorithm that uses hiding and reduction). The selected configuration determines the level of decomposition (maximal or not) and the projection used for the subnets (*Decompose*, *Hide*, or *Hide and reduce*). Whatever classifier and configuration the user selects, the end result will be an overall *pseudo*-alignment for the log and net at hand, where a pseudo-alignment is an alignment except for the fact that the transition sequence may not be executable in the net. If possible, this pseudo-alignment is an alignment, but this is not always possible.

Adding a new miner to the framework is easy, provided that the miner either results in (1) a net with an explicit initial marking and an explicit set of final markings, or (2) a net with an implicit initial marking (one token in every source place) and an implicit set of final markings (a token in one sink place). However, if a new miner emerges that does not satisfy these requirements, then it can still be added, but a wrapper needs to be created that assigns an initial marking and a set of final markings to the discovered net.

The decomposed ILP Miner can discover a net from a log in about half an hour where the ILP Miner takes more than a week (and might not even succeed). The decomposed replayer can replay a case (a log on a corresponding net) in seconds where the replayer itself would take 10 minutes. This shows that decomposition indeed can speed up both the discovery and the replay. Because of the formal guarantees as provided by the decomposition, the results of both the decomposed discovery and replay provide a valuable and reliable alternative. For discovery, precision may drop, while generalization may increase. The possible drop in precision may be mitigated by using a non-maximal decomposition algorithm, like the 50% decomposition algorithm. For replay, costs may drop. However, this drop is not a downside, as the decomposition replay algorithm still points out valid mismatches between a log and a net, and the decomposition algorithm only reports mismatches if the monolithic replay does. As a result, when all mismatches reported by decomposition replay algorithm have been solved, then they will also have been solved for the monolithic replay algorithm.

Future work on the framework includes additional non-maximal decomposition algorithms and improvements of the overhead. Our evaluation shows that in discovery we can go from maximal decomposition to 50% decomposition while maintaining high speed-ups. Discovery may take more time, but on average the computation times are still reasonable, and the results get only better. Therefore, for discovery, we aim to check whether, for example, a 25% decomposition algorithm is even better, both in computation times and in results. In replay, we did not try a 50% decomposition algorithm yet, but this may for sure be very beneficial. Computation times may become better, as we have less sublogs to replay, and replay costs also get better when decomposing less.

Our evaluation shows that in discovery the overhead takes only a minor fraction of the overall time, while in replay the overhead may be considerable. As such, it makes more sense to investigate whether we can improve on the alignment merge than to try to improve the reduction of the discovered overall net.

References

[1] Aalst, W.M.P.v.d.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer-Verlag, 1st edn. (2011)

[2] Aalst, W.M.P.v.d.: Decomposing Petri nets for process mining: A generic approach. Distrib. Parallel Dat. 31(4), 471–507 (2013)

[3] Aalst, W.M.P.v.d., Adriansyah, A., Dongen, B.F.v.: Replaying history on process models for conformance checking and performance analysis. Data Min. Knowl. Disc. 2(2), 182–192 (2012)

[4] Aalst, W.M.P.v.d., Rubin, V., Verbeek, H.M.W., Dongen, B.F.v., Kindler, E., Günther, C.W.: Process mining: a two-step approach to balance between underfitting and overfitting. Softw. Syst. Model. 9(1), 87–111 (2008)

[5] Aalst, W.M.P.v.d., Weijters, A.J.M.M., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE T. Knowl. Data En. 16(9), 1128–1142 (2004)

[6] Agrawal, R., Gunopulos, D., Leymann, F.: Mining process models from workflow logs. In: Schek, H.J., Alonso, G., Saltor, F., Ramos, I. (eds.) Proceedings of the 6th International Conference on Extending Database Technology: Advances in Database Technology. Lect. Notes Comput. Sc., vol. 1377, pp. 467–483 (1998)

[7] Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process mining based on regions of languages. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) Business Process Management: 5th International Conference, BPM 2007, Brisbane, Australia, September 24-28, 2007. Proceedings. Lect. Notes Comput. Sc., vol. 4714, pp. 375–383 (2007)

[8] Berthelot, G.: Transformations and decompositions of nets. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) Advances in Petri Nets 1986 Part I: Petri Nets, central models and their properties. Lect. Notes Comput. Sc., vol. 254, pp. 360–376 (1987)

[9] Broucke, S.K.L.M.v., De Weerdt, J., Vanthienen, J., Baesens, B.: Determining process model precision and generalization with weighted artificial negative events. IEEE T. Knowl. Data En. 26(8), 1877–1889 (2014)

[10] Buijs, J.C.A.M.: Flexible Evolutionary Algorithms for Mining Structured Process Models. Ph.D. thesis, Eindhoven University of Technology (2014)

[11] Calders, T., Günther, C.W., Pechenizkiy, M., Rozinat, A.: Using minimum description length for process mining. In: Proceedings of the 2009 ACM Symposium on Applied Computing. pp. 1451–1455. ACM (2009)

[12] Carmona, J., Cortadella, J.: Process mining meets abstract interpretation. In: Balcázar, J.L., Bonchi, F., Gionis, A., Sebag, M. (eds.) Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2010, Barcelona, Spain, September 20-24, 2010, Proceedings, Part I. Lect. Notes Comput. Sc., vol. 6321, pp. 184–199 (2010)

[13] Carmona, J., Cortadella, J., Kishinevsky, M.: A region-based algorithm for discovering Petri nets from event logs. In: Dumas, M., Reichert, M., Shan, M.C. (eds.) Business Process Management: 6th International Conference, BPM 2008, Milan, Italy, September 2-4, 2008. Proceedings. Lect. Notes Comput. Sc., vol. 5240, pp. 358–373 (2008)

[14] Cook, J.E., Wolf, A.L.: Discovering models of software processes from event-based data. ACM Trans. Softw. Eng. Methodol. 7(3), 215–249 (1998)

[15] Cook, J.E., Wolf, A.L.: Software process validation: Quantitatively measuring the correspondence of a process to a model. ACM Trans. Softw. Eng. Methodol. 8(2), 147–176 (1999)

[16] De Weerdt, J., De Backer, M., Vanthienen, J., Baesens, B.: A robust F-measure for evaluating discovered process models. In: Computational Intelligence and Data Mining (CIDM), 2011 IEEE Symposium on. pp. 148–155. IEEE (2011)

[17] Dongen, B.F.v.: BPI challenge 2012 data set (2012), doi: 10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f

[18] Dongen, B.F.v.: BPI challenge 2015 data set (2015), doi: 10.4121/uuid:31a308ef-c844-48da-948c-305d167a0ec1

[19] Gaaloul, W., Gaaloul, K., Bhiri, S., Haller, A., Hauswirth, M.: Log-based transactional workflow mining. Distrib. Parallel Dat. 25(3), 193–240 (2009)

[20] Goedertier, S., Martens, D., Vanthienen, J., Baesens, B.: Robust process discovery with artificial negative events. J. Mach. Learn. Res. 10, 1305–1340 (2009)

[21] Günther, C.W., Verbeek, H.M.W.: XES standard definition. Tech. Rep. BPM-14-09 (2014)

[22] Hompes, B.F.A., Verbeek, H.M.W., Aalst, W.M.P.v.d.: Finding suitable activity clusters for decomposed process discovery. In: Ceravolo, P., Russo, B., Accorsi, R. (eds.) Data-Driven Process Discovery and Analysis: 4th International Symposium, SIMPDA 2014, Milan, Italy, November 19-21, 2014, Revised Selected Papers. Lect. Notes Bus. Inf., vol. 237, pp. 32–57 (2015)

[23] Leemans, S.J.J., Fahland, D., Aalst, W.M.P.v.d.: Discovering block-structured process models from event logs - a constructive approach. In: Colom, J.M., Desel, J. (eds.) Application and Theory of Petri Nets and Concurrency. Lect. Notes Comput. Sc., vol. 7927, pp. 311–329 (2013)

[24] Martin, N., Depaire, B., Caris, A.: The use of process mining in business process simulation model construction. Wirtschaftsinf. 58(1), 73–87 (2015), journal is also known as Business & Information Systems Engineering.

[25] Maruster, L., Weijters, A.J.M.M., Aalst, W.M.P.v.d., Bosch, A.v.d.: A rule-based approach for process discovery: Dealing with noise and imbalance in process logs. Data Min. Knowl. Disc. 13(1), 67–87 (2006)

[26] Munoz-Gama, J., Carmona, J.: A fresh look at precision in process conformance. In: Hull, R., Mendling, J., Tai, S. (eds.) Business Process Management. Lect. Notes Comput. Sc., vol. 6336, pp. 211–226 (2010)

[27] Munoz-Gama, J., Carmona, J., Aalst, W.M.P.v.d.: Conformance checking in the large: Partitioning and topology. In: Daniel, F., Wang, J., Weber, B. (eds.) Business Process Management: 11th International Conference, BPM 2013, Beijing, China, August 26-30, 2013. Proceedings. Lect. Notes Comput. Sc., vol. 8094, pp. 130–145 (2013)

[28] Munoz-Gama, J., Carmona, J., Aalst, W.M.P.v.d.: Single-entry single-exit decomposed conformance checking. Inf. Syst. 46, 102–122 (2014)

[29] Murata, T.: Petri nets: Properties, analysis and applications. P. IEEE 77(4), 541–580 (1989)

[30] Rozinat, A., Aalst, W.M.P.v.d.: Decision mining in ProM. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) Business Process Management: 4th International Conference, BPM 2006, Vienna, Austria, September 5-7, 2006. Proceedings. Lect. Notes Comput. Sc., vol. 4102, pp. 420–425 (2006)

[31] Rozinat, A., Aalst, W.M.P.v.d.: Conformance checking of processes based on monitoring real behavior. Inf. Syst. 33(1), 64–95 (2008)

[32] Solé, M., Carmona, J.: Process mining from a basis of state regions. In: Lilius, J., Penczek, W. (eds.) Applications and Theory of Petri Nets: 31st International Conference, PETRI NETS 2010, Braga, Portugal, June 21-25, 2010. Proceedings. Lect. Notes Comput. Sc., vol. 6128, pp. 226–245 (2010)

[33] Verbeek, H.M.W., Buijs, J.C.A.M., Dongen, B.F.v., Aalst, W.M.P.v.d.: ProM 6: The process mining toolkit. In: Proc. of BPM Demonstration Track 2010. vol. 615, pp. 34–39. CEUR-WS.org (2010)

[34] Weijters, A.J.M.M., Aalst, W.M.P.v.d.: Rediscovering workflow models from event-based data using Little Thumb. Integr. Comput.-Aided Eng. 10(2), 151–162 (2003)

[35] Werf, J.M.E.M.v.d., Dongen, B.F.v., Hurkens, C.A.J., Serebrenik, A.: Process discovery using integer linear programming. Fundam. Inf. 94(3-4), 387–412 (2009)

[36] Zelst, S.J.v., Dongen, B.F.v., Aalst, W.M.P.v.d.: ILP-based process discovery using hybrid regions. In: Proceedings of the International Workshop on Algorithms & Theories for the Analysis of Event Data, ATAED 2015, Satellite event of the conferences: 36th International Conference on Application and Theory of Petri Nets and Concurrency Petri Nets 2015 and 15th International Conference on Application of Concurrency to System Design ACSD 2015, Brussels, Belgium, June 22-23, 2015. vol. 1371, pp. 47–61. CEUR-WS.org (2015)