Divide and conquer: a tool framework for supporting decomposed discovery in process mining

Verbeek, H.M.W.; Munoz Gama, J.; van der Aalst, W.M.P.

Published in:
The Computer Journal

DOI:
10.1093/comjnl/bxx040

Published: 01/11/2017

© The British Computer Society 2017. All rights reserved.
For permissions, please e-mail: journals.permissions@oup.com
Advance Access publication on 8 May 2017 doi:10.1093/comjnl/bxx040

Divide and Conquer: A Tool Framework for Supporting Decomposed Discovery in Process Mining

H.M.W. Verbeek[1*], W.M.P. van der Aalst[1] and J. Munoz-Gama[2]

[1]*Architecture of Information Systems Group, Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands*
[2]*Information Systems Group, Department of Computer Science, Pontificia Universidad Católica de Chile, Santiago de Chile, Chile*
[*]*Corresponding author: h.m.w.verbeek@tue.nl*

Process mining has been around for more than a decade now, and, in that period, several discovery algorithms have been introduced that work fairly well on average-sized event logs, that is, event logs that contain ∼50 different activities. Nevertheless, these algorithms have problems dealing with big event logs, that is, event logs that contain 200 or more different activities. 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 or even smaller) 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 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 and that has been implemented in ProM 6. Using this novel framework, this paper demonstrates that significant speed-ups can be achieved for discovery. This paper also discusses the fact that decomposition may lead to different results, but that this may even turn out to have a positive effect.

Keywords: process mining; decomposition; discovery; ProM

Received 15 April 2016; revised 13 February 2017; editorial decision 6 April 2017
Handling editor: Rada Chirkova

1. 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 1 shows data related to a typical event recorded for some system, which can be interpreted as follows [2]:

On 1 October 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.

Typically, research done in the process mining area can be divided into three subfields: *process discovery*, *process conformance* and *process enhancement*. In this paper, we will only consider process discovery.

The field of *process discovery* [1] deals with discovering a process model from an event log. Example process discovery algorithms include the Alpha Miner [3], the ILP Miner [4] and the Inductive Miner [5]. The Alpha Miner was the first process discovery algorithm to discover concurrency adequately. The Integer Linear Problem (ILP) Miner basically converts the discovery problem into many ILP and solves the discovery problem by solving all these ILPs. The Inductive Miner is the most recent discovery algorithm of these three, which discovers block-structured [6] models in limited time by using a powerful divide-and-conquer approach.

As indicated, the ILP Miner may use many ILPs to solve the problem at hand. The size of these ILPs is mainly

TABLE 1. Event e_1.

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

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 [7]. This log contains 2000 traces, 57 activities, an average trace length of 52 and noise. For this event log, we have created nine increasingly 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 process mining framework ProM 6 [8]. 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 five 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 200 or more different activities, Ref. [9] has proposed a theoretical *decomposition* approach for process discovery. Instead of discovering a process model from the *overall* event log (the *monolithic* approach), this decomposed approach first decomposes the event log into a number of sublogs that each contains only a subset of the activities from the overall event log, second it employs the discovery algorithm on each of these sublogs resulting in as many process submodels, and third it merges all submodels into an overall process model. This decomposition approach may be significantly faster than the monolithic approach, provided that

(1) the event log can be decomposed reasonably fast over the sublogs,

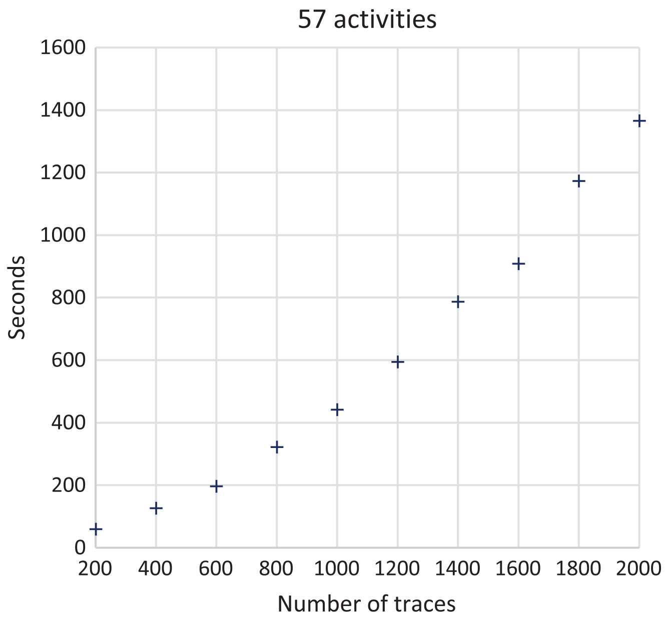


FIGURE 1. The effect of different numbers of traces using the *ILP miner*.

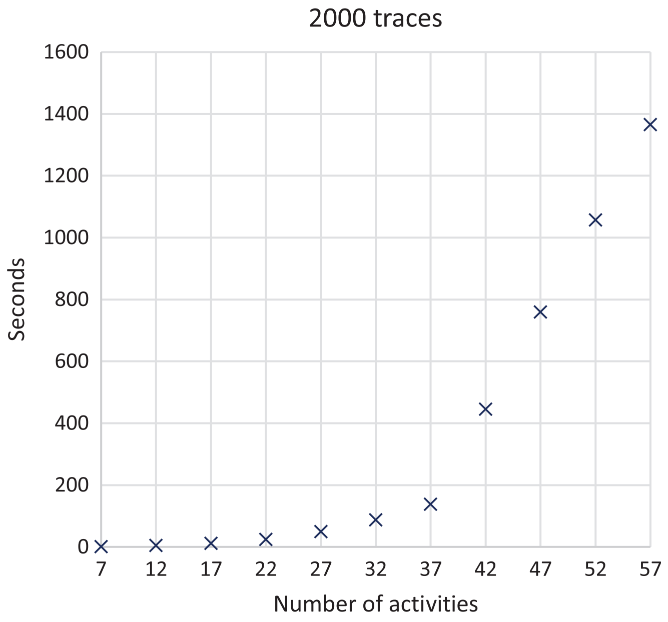


FIGURE 2. The effect of different numbers of activities using the *ILP miner*.

(2) the discovery algorithm is significantly faster on the sublogs, possibly by running it concurrently on these sublogs on different machines, and
(3) the discovered submodels can be merged in a reasonably fast way, even though it may require solving some ILP to remove redundant places [9].

[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.

This paper introduces the *Divide And Conquer* tool framework, which instantiates the theoretical decomposed discovery approach as introduced in [9], and which has been implemented in ProM 6. This framework offers an easy integration of existing discovery algorithms, that is, existing algorithms can be *decomposed* in an easy way. This paper also includes two evaluations of this framework: a first using a number of artificial data sets varying in size and complexity [10, 11, 7] and a second using real-life data sets varying in size and complexity [12, 13]. Finally, this paper discusses the fact that using decomposed discovery may lead to different results, but it also shows that this may have a positive effect as it was needed to win a contemporary process discovery contest [14]. Results show that (i) decomposition may provide results in cases where the monolithic approach fails, (ii) for larger cases decomposition provides results in less time, (iii) decomposition may actually result in a model that explains the log at hand at-least-as-good (that is, it may at-least-as-good classify whether or not a new trace originates from the same system that generated the log) and (iv) the decomposition overhead is insignificant when using a complex miner like the ILP Miner.

The remainder of this paper is organized as follows. Section 2 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 3 introduces the tool framework, which includes (i) different heuristics to split an overall event log into sublogs, (ii) information on how to add an existing discovery algorithm to the framework, (iii) an approach to merge many discovered subnets into an overall accepting Petri net and (iv) the implementation of the framework in ProM 6. Section 4 introduces the two evaluations conducted with the ILP Miner and the tool framework. The first evaluation uses artificial logs, whereas the second evaluation uses real-life logs. Section 5 discusses the fact that decomposed discovery may lead to different results but also that these differences may be positive. Furthermore, this section discusses the use of the other, non-ILP Miner, discovery algorithms within the framework. Section 6 concludes the paper.

2. PRELIMINARIES

This section presents the key concepts informally. See [9] for formalizations of these concepts.

2.1. 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 2 shows the example activity log L_1, which contains information on 20 cases. For example, four cases followed the trace $\langle a_1, a_2, a_4, a_5, a_8\rangle$. In total, the log contains eight activities ($\{a_1, \ldots, a_8\}$) and $13 + 17 + 9 + 2 \times 9 + 9 + 4 \times 5 + 9 + 9 + 5 + 5 + 17 + 3 \times 5 + 5 + 5 = 156$ activity occurrences.

TABLE 2. 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

An *event log* is a collection of traces, where every trace is a sequence of *events*. Table 1 shows a typical event from an *event* log, containing the following attributes [2]:

- **concept:name** The activity name of the event, in this case the events refer 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 [2]).
- **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 1 October 2015.

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* [2], 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 [2], this will be the value of the `concept:name` attribute (see, for example, Table 2), 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.

2.2. 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 $\left[p_1^2, p_2^3\right]$. This latter marking would be visualized by putting two dots in place p_1 and three dots in place p_2.

As usual [15], 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 $M\left[t\right\rangle M'$, 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 $\left\langle [p_1], t_1, [p_2], t_2, [p_3, p_4], t_3, [p_4, p_5]\right\rangle$ 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 a transition sequence.

As usual in process mining [1], we extend Petri nets with labels, an initial marking and a set of final markings, yielding an *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. Invisible transitions are visualized using a black square.

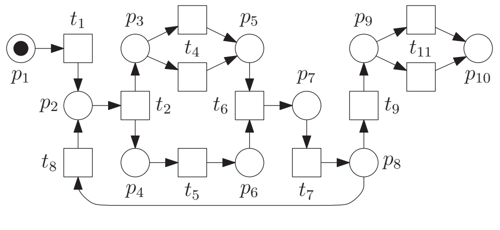


FIGURE 3. A Petri net.

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.

2.3. Discovery algorithms

A *discovery algorithm* (see Fig. 5) is an algorithm that takes as an 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 [3],

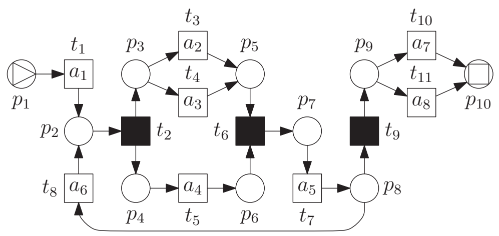


FIGURE 4. An accepting Petri net N_1. Note that transitions are labeled and there is a well-defined start and end.

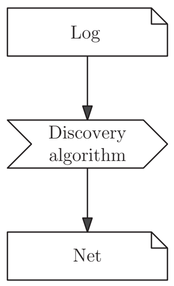


FIGURE 5. Conceptual view on a discovery algorithm.

the Heuristics Miner [16], the Hybrid ILP Miner [17], the ILP Miner [4] and the Inductive Miner [5]. Example discovery algorithms that may result in a non-injective labeling function include the Evolutionary Tree Miner [18].

3. TOOL 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 (i) the complexity of the discovery algorithm is significantly worse than linear (in the number of different activities) and (ii) the additional overhead of having to decompose the log beforehand and merge the submodels afterwards do not spoil the benefits, 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 6 then shows a conceptual view on a decomposed discovery algorithm. First, the algorithm uses different heuristics to construct a collection of possible activity cluster sets, and selects the best activity cluster set from that collection. Second, for every activity cluster in the selected set, 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.

3.1. 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 7 shows the approach the decomposed discovery algorithm uses to achieve this. First, a matrix is discovered 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

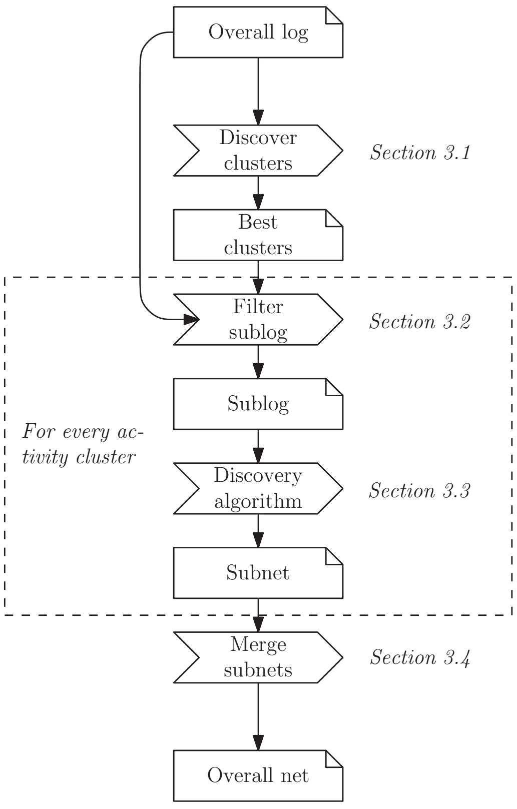


FIGURE 6. Conceptual view on a decomposed discovery algorithm.

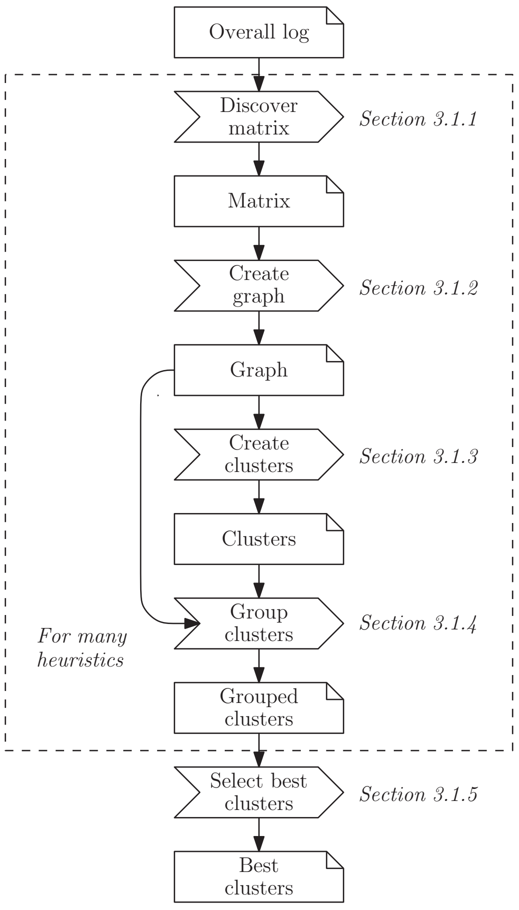


FIGURE 7. Conceptual view on discovering activity clusters.

TABLE 3. 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

from the initial set of clusters by grouping very small or very coherent clusters together.

These four steps are executed using a collection of different settings (different heuristics), leading to a collection of as many cluster sets. Fifth and last, the best set from the collection is selected and returned as a result. In the remainder of this section, we provide the necessary details for these five steps.

3.1.1. *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 3 shows an example causal activity matrix M_1 for log L_1.

The strengths in a causal activity matrix range from -1.0 (weakest) to 1.0(strongest), 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.

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 4 shows an overview of the heuristics currently implemented in the tool framework for discovering a causal activity matrix from a log.

3.1.2. *Create graph*

In this step, we create a graph containing the stronger direct causal relations. This is done by removing the relations from the casual activity matrix that are not strong enough to satisfy a preset threshold. We will refer to such graph as a *causal activity graph*. Figure 8 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

TABLE 4. 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

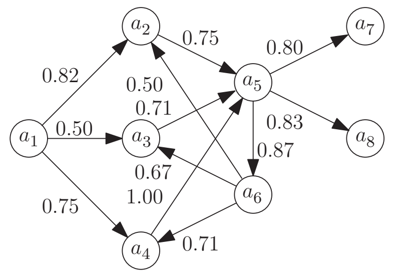


FIGURE 8. Example causality graph G_1 for matrix M_1.

confident we are that 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 $M\perp_z$ using the following rule:

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

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 $M_1\perp_{0.5}$.

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:

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

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 [1]. In case of concurrent activities, direct causal relations are not wanted.

3.1.3. 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. For this equivalence class, two arcs are directly equivalent if one of the following conditions hold:

- *Input arcs*: Both arcs share the same source node. As an example, the arcs (a_1, a_2), (a_1, a_3) and (a_1, a_4) in Fig. 8 belong to the same equivalence class, as they all have a_1 as source node.
- *Output arcs*: Output arcs of the same target node. As an example, the arcs (a_1, a_2) and (a_6, a_2) in Fig. 8 belong to the same equivalence class, as they both have a_2 as target node.

As now (a_1, a_2) is equivalent to both (a_1, a_3) and (a_6, a_2), (a_1, a_3) and (a_6, a_2) are equivalent as well (if x is equivalent to y and y is equivalent to z, then x is equivalent to z). In a similar way, the arcs (a_1, a_4), (a_6, a_4) and (a_6, a_3) are also equivalent to (a_1, a_2). However, the arc (a_3, a_5) is not equivalent to (a_1, a_2), as there are no nodes equivalent to (a_1, a_2) that have either a_3 as source node or a_5 as target node. Note that although there are equivalent arcs that have a_3 as *target* node, there are no arcs that have a_3 as *source* node. Second, a single cluster is created for every equivalence class. This cluster contains all arcs in that equivalence class, and all nodes connected to these arcs. As an example, the arcs (a_1, a_2), (a_1, a_3), (a_1, a_4), (a_6, a_2), (a_6, a_3) and (a_6, a_4) form a cluster together with the nodes a_1, a_2, a_3, a_4 and a_6. Likewise, the arcs that share node a_5 as target node form a cluster, as do the arcs that share a_5 as source node. Figure 9 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 cluster, etc. This ordering allows us to keep track of which subnet was discovered from which sublog.

3.1.4. Group clusters

This step changes the initial set of activity clusters by grouping clusters that are strongly related to each other. As an example, the two leftmost clusters as shown in Fig. 9 have three activities in common(a_2, a_3 and a_4), whereas the other pairs of clusters only have a single activity in common (either a_5 or a_6). When having to group clusters, it is, therefore, better to group the two leftmost clusters. Figure 10 shows the resulting set of activity clusters.

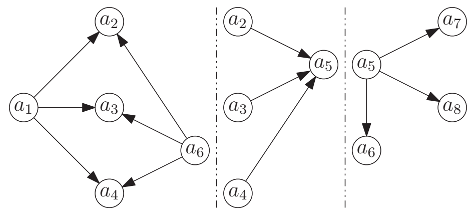


FIGURE 9. Example activity clusters C_1 for graph G_1.

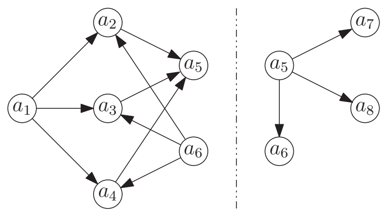


FIGURE 10. Example grouped activity clusters C_2.

This grouping of clusters, along with the reason for doing this, has been described in detail in [19]. 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 5. 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.

TABLE 5. Quality metrics for activity clusters. Although other metrics are possible as well, we limit ourselves here to those defined in [19].

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 6. 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	$\{-0.5, 0.0, 0.5, 0.9\}$	$\{0.005\}$

3.1.5. 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 6 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.

The reason for selecting these heuristics is that experiments have shown 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 these parameters, that is, $(-1.0, 1.0)$. The reason for adding the values -0.6 and 0.9 is that we have seen empirically that for these values these heuristics often provide good results.

3.2. 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, the second sublog to the second cluster, etc. For a given cluster, a sublog is obtained from the log by filtering in those activities that correspond to nodes in that cluster. As an example, Table 7 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 an example of this, Fig. 11 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

TABLE 7. Filtered traces for activity log L_1 and activity clusters C_1 in Fig. 9 in tabular form.

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

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 $\langle a_1, a_4, a_2\rangle$) [9]. 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.

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 8 shows the result of adding these two artificial activities to the first sublog. Figure 12 shows the result of running the Alpha miner on this sublog. Clearly, the resulting net now handles activity a_4 in a proper way.

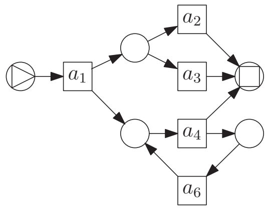


FIGURE 11. Result of the Alpha Miner on the sublog obtained for cluster $\{a_1, a_2, a_3, a_4, a_6\}$.

TABLE 8. Filtered traces with artificial start and end activities added for the first cluster in Table 7.

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

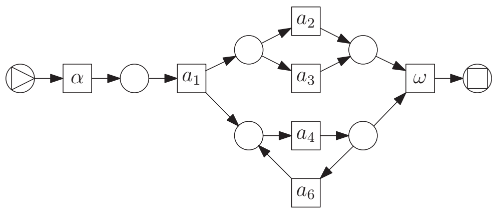


FIGURE 12. Result of the Alpha Miner on the first sublog with artificial start and end activities.

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.

3.3. Discovery algorithm

The goal of this step is to discover a subnet from every sublog by using the provided discovery algorithm. Table 9 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 toward 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 with an initial marking but without indicating explicit final markings. Recall that the replay needs to check which traces are *accepted* by the model. As a result, not only the initial marking but also the final markings are important. Therefore, we assume that a Petri net has both an explicit initial marking and an explicit collection of final markings. Such a Petri net we call an *accepting* Petri net.

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

TABLE 9. Discovery algorithms in ProM 6[8] 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 [3]	'Alpha Miner'	
Heuristics Miner [16]	'Mine for a Heuristics Net using Heuristics Miner'	'Convert Heuristics net into Petri net'
Hybrid ILP Miner [17]	'ILP-Based Process Discovery'	
ILP Miner [4]	'ILP Miner'	
Inductive Miner [5]	'Mine Petri net with Inductive Miner, with parameters'	
Evolutionary Tree Miner [18]	'Mine a Process Tree with ETMd using parameters and classifier'	'Convert Process Tree to Petri Net'

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 9 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.

As an example, Fig. 13 shows the resulting subnets that the Hybrid ILP Miner discovered from the sublogs that are shown in Table 7.

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.

3.4. 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.

3.4.1. *Join subnets*

This step joins a collection of subnets into one overall net using the following rules (cf. [9]):

- *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.

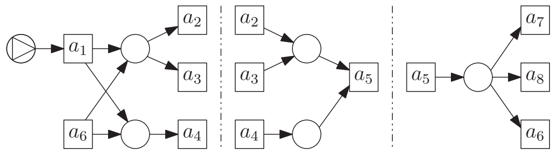


FIGURE 13. Result of the Hybrid ILP Miner on all sublogs from Table 7.

- *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.

Assume, for the sake of argument, that some discovery algorithm has discovered the two subnets as shown in Fig. 14. Joining these two subnets results in the overall net shown in Fig. 15.

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 Fig. 16. 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 17 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 and t_6 in the subnet is now replaced by the transition

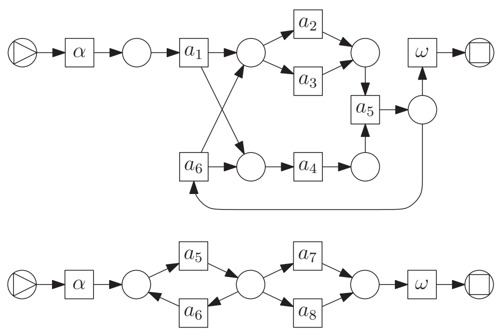
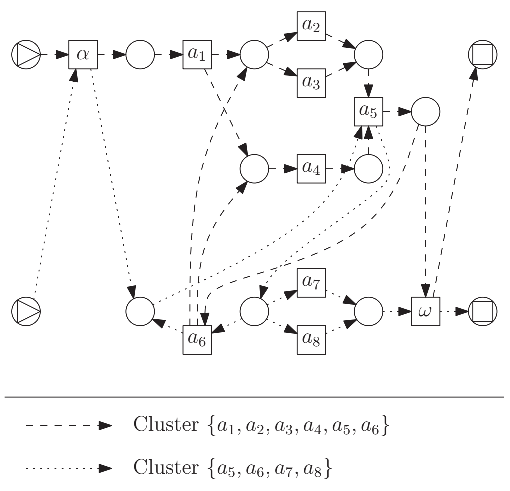


FIGURE 14. Possible nets resulting from discovery algorithm.



α
a_1
a_2
a_3
a_4
a_5
a_6
a_7
a_8
ω
Cluster $\{a_1, a_2, a_3, a_4, a_5, a_6\}$
Cluster $\{a_5, a_6, a_7, a_8\}$

FIGURE 15. Net that result from joining the subnets as shown in Fig. 14.

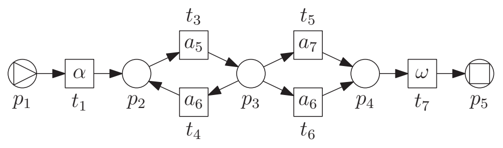


FIGURE 16. Possible discovered net that contains two duplicate transitions labeled a_6.

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 in a single subnet are removed first by adapting every subnet.

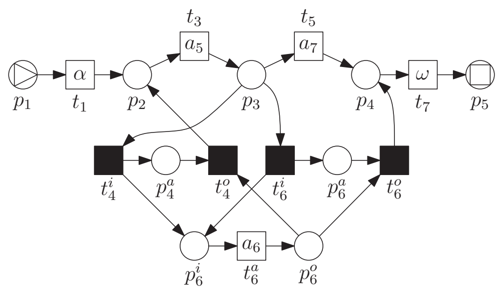


FIGURE 17. Similar net that contains only one visible transition labeled a_6.

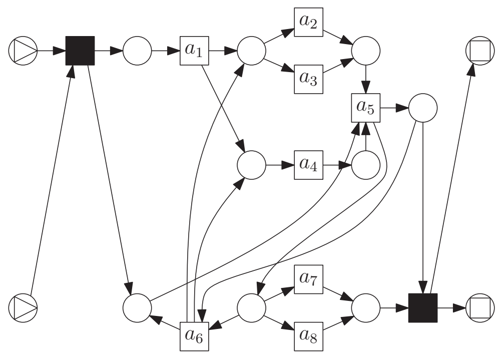


FIGURE 18. Net from Fig. 15 with artificial labels made invisible.

3.4.2. *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, Fig. 18 shows the result of performing this step on the net as shown in Fig. 15.

3.4.3. *Reduce net*

This step reduces the size of the overall Petri net by applying variants on classical behavior-preserving reduction rules [20] and by removing places that are structurally redundant [21]. 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 Fig. 18 that corresponds to the transition labeled α in Fig. 15. 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 Fig. 15. This transition has the same

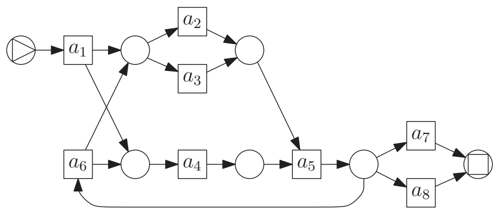


FIGURE 19. Net from Fig. 18 after reductions.

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 [20] 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.

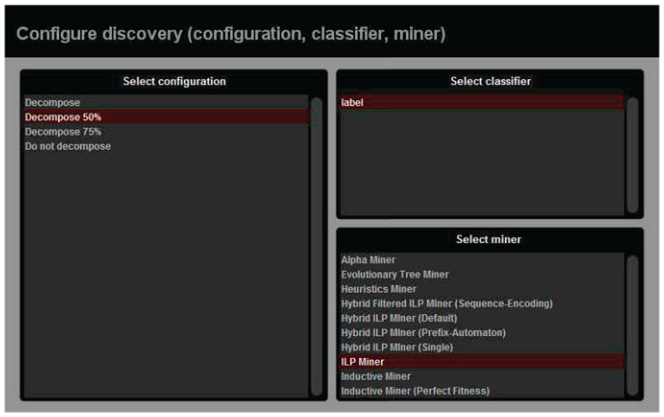
As an example, Fig. 19 shows the result of performing this step on the net as shown in Fig. 18. In this example, the reduction was able to remove all invisible transitions. However, in general, this might not be the case (e.g. skip transitions).

3.5. Implementation

The decomposed discovery algorithm has been implemented as the *Discover using Decomposition* action in the publicly available *DecomposedMiner* package of ProM 6. Figure 20 shows the dialog for this action, which allows the user to select the *configuration*, the classifier (see Section 2.1), and the discovery algorithm (see Section 3.3).

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 3.1.1), the aforementioned 12 configurations (see Table 6) are used. For creating a graph (see Section 3.1.2), the zero value is set to 0.0 and the concurrency threshold to 0.005. For creating initial clusters (see Section 3.1.3), no parameters are required. For filtering the overall log (see Section 3.2), 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 3.3), the selected discovery algorithm is used. Merging the subnets into an overall net (see Section 3.4) 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 3.1.5). As an example, if the best initial


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)

FIGURE 20. Dialog for *Discover using Decomposition* action in ProM 6.

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.

4. EVALUATIONS

This section evaluates the implemented tool framework on existing artificial and real-life data sets. For both evaluations, we use the *ILP Miner*, as this discovery algorithm is known to have an exponential complexity [4] in the number of different activities in the log. Although other discovery algorithms are supported by the framework (see also Table 9), in this evaluation, we focus only on the ILP miner.

To assess the quality of the discovered models, we use the state-of-the-art alignment-based conformance metrics *precision* and *generalization*. In the area of process mining, four such quality metrics are generally accepted [1]:

- *Fitness*: The extent to which the model allows for the behavior as seen in the event log.
- *Precision*: The extent to which the model does not allow for behavior completely unrelated to behavior as seen in the event log.
- *Generalization*: The extent to which the model generalizes the behavior as seen in the event log.
- *Simplicity*: The extent to which the model is the simplest model that explains the behavior as seen in the event log.

As the ILP Miner guarantees a perfect fitness (the discovered model always allows for all behavior as seen in the event log), there is no reason to assess fitness. Furthermore, the metrics for simplicity are rather subjective and do not relate to the model's behavior. Therefore, we restrict this quality assessment to the precision and generalization metrics.

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) Next, we measure the quality of the resulting net with respect to the log using the precision and generalization metrics. For this, we replay [22] the given event log on the discovered net. This provides us with the log alignment needed for the next step.
(4) We calculate 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 are 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 3.5) 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).

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 Fig. 6), 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.

All plug-ins used for doing the evaluations are available through the *Divide And Conquer Test* package in ProM 6. This package can be downloaded from `https://svn.win.tue.nl/repos/prom/Packages/DivideAndConquerTest/Trunk`, which is a folder in our subversion repository.

The evaluations 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 is allocated to the Java VM. Note that the approach can be distributed over multiple computers, but we only use one computing node.

4.1. Artificial data sets

Table 10 shows the list of three *artificial* data sets containing 59 event logs (with their characteristics) that are used for this evaluation.

TABLE 10. Artificial data sets used in the evaluation.

Artificial data set	Description
DMKD 2006 [10]	20 synthetic events logs generated from four Petri nets, containing 12, 22, 32 and 42 activities, 1000 traces, and different noise levels. This data set uses 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
IS 2014 [7]	32 synthetic event logs generated from four highly structured Petri nets, containing 59, 48, 32 and 57 activities, 2000 traces, four different average trace lengths (~15–55), with and without noise. This data set uses case labels like *59/55/n*, where *59* indicates the reported number of activities in [7], *55* indicates the average trace length, and *n* indicates that this log contains noise
BPM 2013 [11]	7 synthetic event logs (*A*–*G*) generated from seven 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%. This data set uses case labels like *prAm6*, which directly relates to the case from the data set

TABLE 11. Feasible artificial logs for all configurations. 'Yes' indicates that both discovery and replay are feasible, 'Yes/No' that discovery is feasible but replay is not, and 'No' that discovery is not feasible.

Data set	Event log	*Do not decompose*	*Decompose 50%*	*Decompose 75%*	*Decompose*
DMKD 2006	*All*	Yes	Yes	Yes	Yes
IS 2014	*All*	Yes	Yes	Yes	Yes
BPM 2013	prAm6	Yes	Yes/No	Yes/No	Yes/No
	prBm6	Yes	Yes/No	Yes/No	Yes/No
	prCm6	Yes/No	Yes	Yes	Yes
	prDm6	No	No	No	No
	prEm6	Yes/No	Yes/No	Yes/No	Yes/No
	prFm6	No	No	No	No
	prGm6	No	Yes/No	Yes/No	Yes/No

4.1.1. *Monolithic vs. maximal-decomposition*

First, we show for which logs in the artificial 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 artificial log on the discovered net [22]. Therefore, third, we show for which artificial logs this replay is 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.

Feasibility. Table 11 shows for every log from the artificial data sets whether they are feasible using the *Do not decompose* and *Decompose* configurations.Both configurations run out of memory for the *prDm6* and *prFm6* logs from the *BPM 2013* data set, while *Do not decompose* runs out of time (that is, it needs to be stopped after a week) for the *prGm6* log from the same data set. This table shows that one more log (the *prGm6* log) is feasible with the *Decompose* configuration, and that hence we can only compare computation times for those logs that are feasible with *Do not decompose*.

Computation times. Figure 21 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* takes almost 75.000 seconds (>20 hours) to discover a net from the *prCm6* log, and it also shows that *Decompose* is ~150 times as fast, needing only ~500 seconds (<10 minutes). *Decompose* outperforms *Do not decompose* for all logs where the latter takes >10 seconds.

Figure 21 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 artificial 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 assumed both data sets to be of similar complexity.

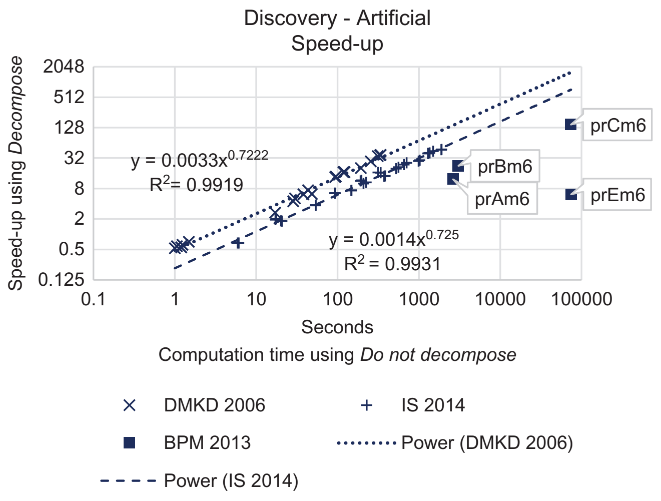


FIGURE 21. Comparison of feasible computation times for the artificial logs. The speed-up by decomposition tends to be high when computation times are long.

Figure 22 shows, for the 10 most time-consuming artificial logs, the feasible computation times for both configurations, and also where time is spent. First, time is spent on the discovery of the subnets, that is, on running the discovery algorithm on the sublogs. Figure 22 shows the percentage of time spent on this (see the bottom-most bars, labeled *Discovery*). Second, time is spent on the reduction of the discovered overall net, which is shown using the middle bars, labeled *Reduction*. Third, time is spent on, for example, computing the best activity cluster, splitting the log or merging the subnets into an overall net, which is accumulated in the top-most bars, labeled *Other*. Clearly, *Decompose* spends the majority of its time (at least 86% for the logs shown in Fig. 22, at least 83% for all feasible logs) in the decomposed discovery, only a fraction is spent on the overhead of the decomposition approach. This shows that when using the decomposed ILP Miner, 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 22 also shows the computation times using *Decompose* for the *prGm6* log, for which *Do not decompose* is infeasible. It takes *Decompose* ~62 000 seconds (~17 hours) to discover a net from the *prGm6* log. Given the fact that *Do not decompose* for this log needs to be stopped after a week, the speed-up of *Decompose* for this log is at least 10.

Feasibility of replay. Table 11 also shows for which of the feasible artificial logs the replay is feasible. As mentioned earlier, this replay is required to compute the precision and generalization metrics. For the artificial logs for which the replay is not feasible (that is, for all logs from the *BPM 2013* data set except the *prCm6* log), the evaluation runs out of time. As a result, we can only compare precision and generalization for all logs in the *DMKD 2006* data set and all logs in the *IS 2014* data set.

Precision. Figure 23 shows the precision values obtained using *Do not decompose*, and the precision gain/loss as obtained using *Decompose*. As an example, the precision obtained with *Do not decompose* on the *48/12/n* log is ~0.86, and *Decompose* results in a precision loss of ~0.33, which results in a precision of $0.33 \times 0.86 \approx 0.28$). 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 (gain of 1.0123) and the *32/18/-* log (gain of 1.0042).

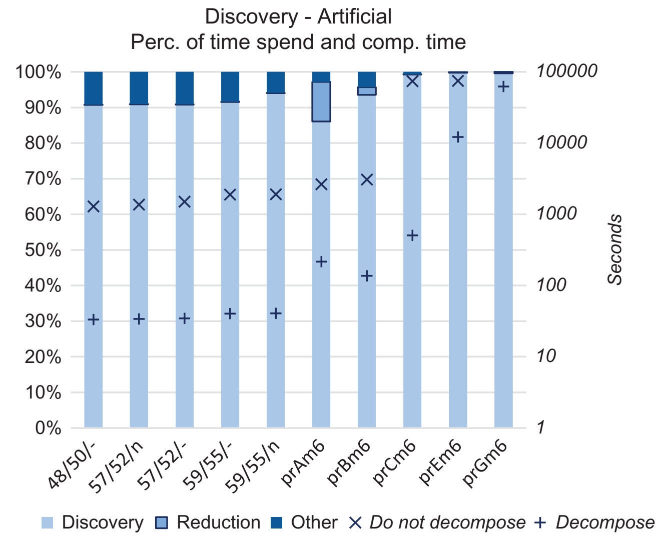


FIGURE 22. Categorized percentages of computation times for the most-hard feasible artificial logs together with their computation times.

Figure 24 shows why precision can be lower when using *Decompose*. The net that is discovered with *Decompose* contains three source transitions (transitions without incoming

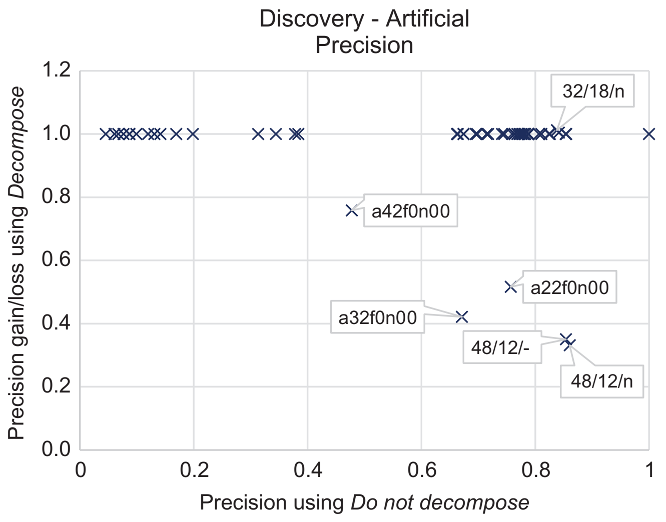


FIGURE 23. Comparison of precision metrics on all feasible artificial logs.

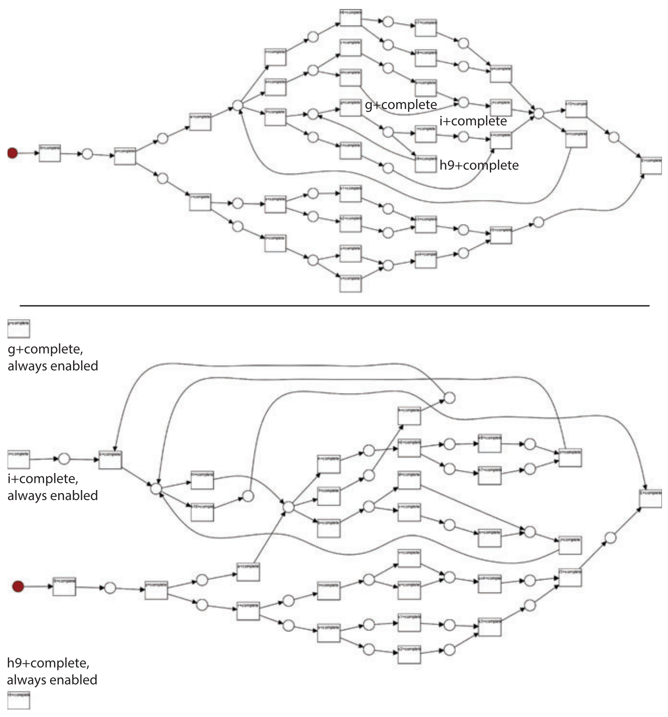

g+complete
i+complete
h9+complete
g+complete, always enabled
i+complete, always enabled
h9+complete, always enabled

FIGURE 24. 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*.

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 (0.28 instead of 0.67).

In contrast, Fig. 25 shows that precision can also be (slightly) 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 slightly more precise (0.85 instead of 0.84).

Generalization. Figure 26 shows the generalization values obtained using *Do not decompose*, and the generalization gain/loss 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.

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 larger and more complex logs than the monolithic algorithm.

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

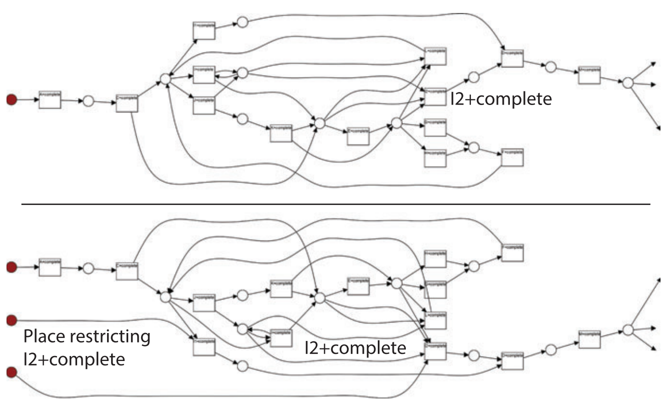


FIGURE 25. Example *32/18/n* explaining why precision can be higher when using *Decompose*. The top net is the result from *Do not decompose*, the bottom net from *Decompose*.

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

4.1.2. Maximal-decomposition vs. non-maximal-decomposition

First, we show for which artificial logs all three configurations (*Decompose*, *Decompose 75%* and *Decompose 50%*) 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 artificial logs (and discovered nets) the replay [22] is 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.

Feasibility. Table 11 shows for which artificial logs the *Decompose 50%* configuration (simply called *Decompose 50%* henceforth) and the *Decompose 75%* configuration (simply called *Decompose 75%* henceforth) are feasible.Both *Decompose 75%* and *Decompose 50%* only fail for the *prDm6* and *prFm6* logs from the *BPM 2013* data set (by also running out of memory). As a result, we can compare computation times for all logs that are feasible with *Decompose*.

Computation times. Figure 27 shows the computation times required by *Decompose*, and the speed-ups obtained using *Decompose 75%* and *Decompose 50%*. As an example, it takes *Decompose* ~210 000 seconds (~58 hours) to discover a net from the *prGm6* log, and the speed-up of *Decompose*

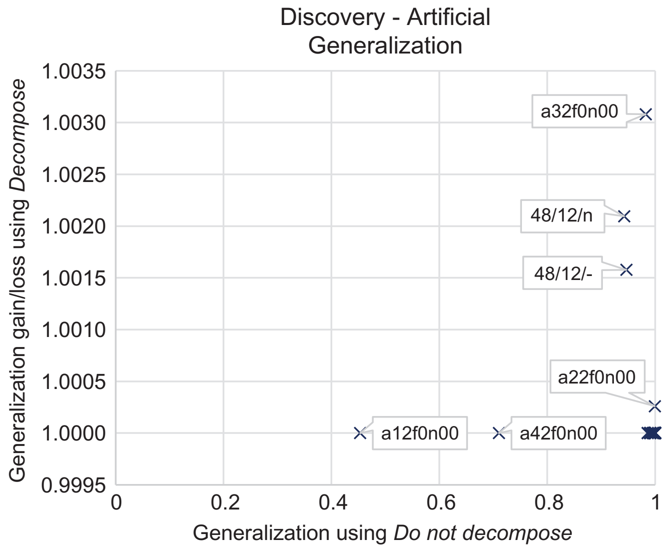


FIGURE 26. Comparison of generalization metrics on all feasible artificial logs. *Decompose* results in very similar generalization values.

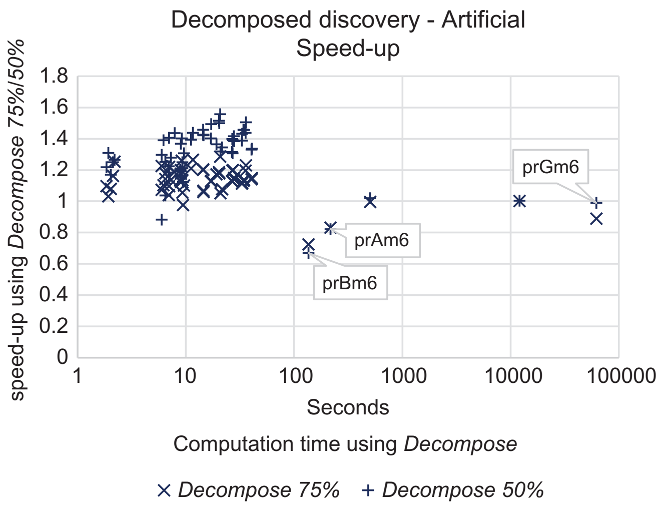


FIGURE 27. Comparison of decomposed computation times for artificial logs.

50% is ~1.09, resulting in a required computation time of ~193 000 seconds (~54 hours). For the easier logs, *Decompose 50%* outperforms *Decompose*, but for the harder logs, there seems to be no improvement.

Feasibility of replay. Table 11 also shows for which of the feasible artificial logs the replay is feasible. The replay on the nets discovered using *Decompose 50%* is feasible for exactly the same set of logs as for which *Decompose 75%* is feasible. Like with the *Decompose* configuration, the replay for *Decompose 75%* and *Decompose 50%* runs out of time for all logs from the *BPM 2013* data set but the *prCm6* log. As a result, we can compare precision and generalization for all logs that are feasible with *Decompose*.

Precision. Figure 28 shows the precision values obtained using *Decompose*, and the gains/losses using *Decompose 75%* and *Decompose 50%*. As an example, the precision value for the *a32f0n00* log as obtained using *Decompose* is ~0.28, and the gains for the two other configurations are ~2.37, resulting in a precision value of ~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 logs. Still, for some other logs (*a42f0n00*, *48/12/-* and *48/12/n*), these configurations do 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.

Generalization. Figure 29 shows the generalization values obtained using *Decompose*, and the gains/losses using *Decompose 75%* and *Decompose 50%*. As an example, the precision value for the *a32f0n00* log as obtained using

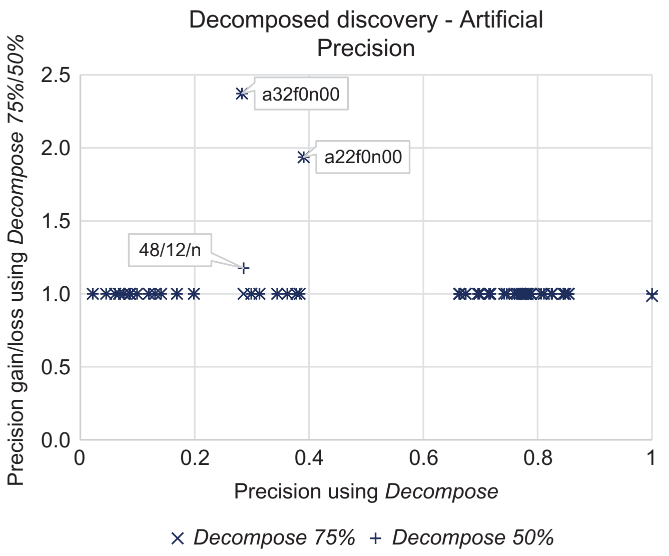


FIGURE 28. Comparison of decomposed precision metrics for artificial logs.

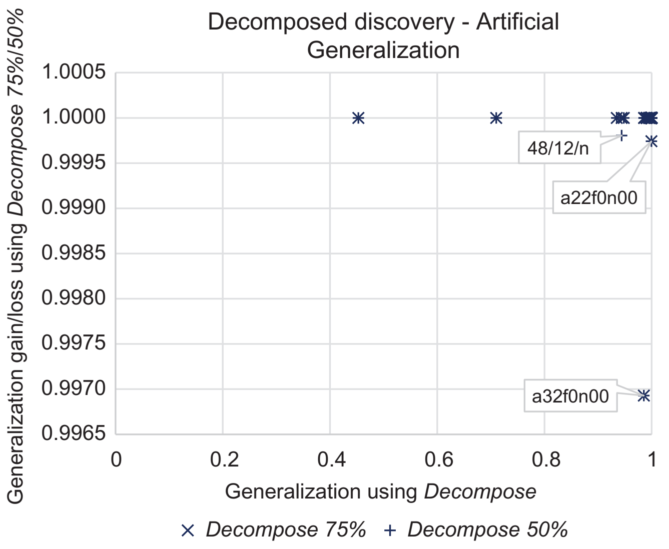


FIGURE 29. Comparison of decomposed generalization metrics for artificial logs.

Decompose is ~0.98, and the losses for both other configurations are ~0.997, resulting in a precision value of ~0.98.

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 an 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 logs that take <100 seconds, the 50% decomposition algorithm is the fastest, but it is considerably slower for some logs that require more time, like the *prAm6* and *prBm6* logs. Apparently, for these logs, 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.

4.2. Real-life data sets

Table 12 shows the list of *real-life* data sets (with their characteristics) that are used for this evaluation. As an example, Fig. 30 shows a graphical overview of the *BPIC 2012* event log, which nicely shows that in the underlying process the vast majority of the work is done on working days (the vertical gaps in the overview correspond to the weekends).

4.2.1. *Monolithic vs. maximal-decomposition*

First, we show for which logs in the real-life data sets both configurations are feasible. Second, we show the computation times required by both configurations, and compare them where possible. Third, we show for which logs the replay [22] required for precision and generalization is feasible. Fourth, we show the feasible precision and generalization values obtained by both configurations, and compare them where possible. Finally, we summarize our findings.

Feasibility. Table 13 shows for every real-life data set and both configurations, the set of logs that are feasible. The *Do not decompose* configuration runs out of time (that is, it needs to be stopped stopped after a week) for all logs from the *BPIC 2015* data set. This table clearly shows that more real-life logs are feasible with the *Decompose* configuration, and that hence we can only compare computation times for those logs that are feasible with *Do not decompose*.

Computation times. Figure 31 shows, for all real-life logs, the feasible computation times for both configurations (where possible), and also where time is spent. Clearly, *Decompose* spends the majority of its time (about than 90% for the *BPIC 2012* log and more than 99% for the *BPIC 2015* logs) in the decomposed discovery, only a fraction is spent on the overhead of the decomposition approach. Like with the artificial logs, this shows that when using the ILP miner, there is no urgent need to improve on, for example, the reduction of the net, as the entire approach would hardly benefit from this.

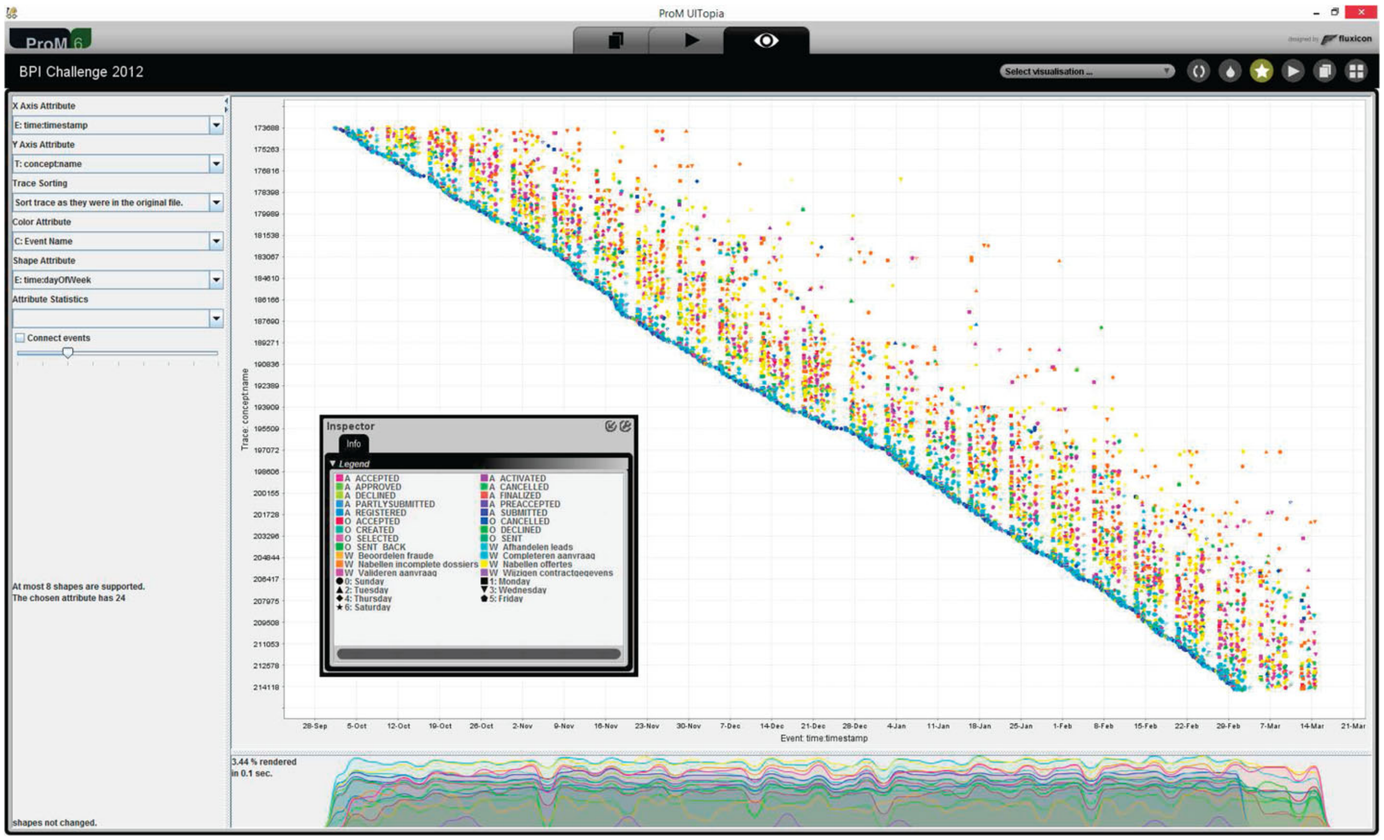
Figure 31 also shows that on the *BPIC 2012* log, *Decompose* is ~25 times as fast as *Do not decompose*. Furthermore, it shows the computation times using *Decompose* for the *BPIC 2015* logs, for which *Do not decompose* is infeasible. For example, it takes *Decompose* 2167 seconds (~36 minutes) to discover a net from the *BPIC2015_5* log. As *Do not decompose* for this log needs to be stopped after a week, the speed-up of *Decompose* for this log is at least 280.

Feasibility of replay. Table 13 also shows for which of the real-life logs the replay is feasible.As the discovery using *Do not decompose* is not feasible for any of the *BPIC 2015* logs, we can only compare precision and generalization for the *BPIC 2012* log.

Precision and generalization. The precision and generalization values obtained using *Decompose* are exactly the same as the values using *Do not decompose*. This is not surprising, as both discover the same net. Figure 32 shows the few

TABLE 12. Real-life data sets used in the evaluation.

Real-life Data set	Description
BPIC 2012 [12]	A real-life log taken from a Dutch Financial Institute. This log contains 13 087 cases, ~262 200 events, and 36 event classes. Apart from some anonymization, the log contains all data as it came from the financial institute. The process represented in the event log is an application process for a personal loan or overdraft within a global financing organization. The event log is a merger of three intertwined sub processes
BPIC 2015 [13]	Five real-life event logs, provided by five Dutch municipalities. The data contains all building permit applications over a period of ~4 years. There are many different event classes present. The cases in the log contain information on the main application as well as objection procedures in various stages. Furthermore, information is available about the resource that carried out the task and on the cost of the application. Some statistics on the logs: • **Log 1**: 1199 cases, 52 217 events, 398 event classes • **Log 2**: 832 cases, 44 354 events, 410 event classes • **Log 3**: 1409 cases, 59 681 events, 383 event classes • **Log 4**: 1053 cases, 47 293 events, 356 event classes • **Log 5**: 1156 cases, 59 083 events, 398 event classes



X Axis Attribute: E: time:timestamp
Y Axis Attribute: T: concept:name
Trace Sorting: Sort trace as they were in the original file.
Color Attribute: C: Event Name
Shape Attribute: E: time:dayOfWeek
Attribute Statistics:
Connect events

At most 8 shapes are supported.
The chosen attribute has 24

shapes not changed.

FIGURE 30. A dotted chart of the real-life *BPIC 2012* event log. Cases are plotted on the *Y*-axis, time on the *X*-axis, the color of a dot denotes the event class, and the shape of the dot denotes the day of the week.

TABLE 13. Feasible real-life logs for all configurations. 'Yes' indicates that both discovery and replay are feasible, 'Yes/No' that discovery is feasible but replay is as not, and 'No' that discovery is not feasible.

Data set	Event log	*Do not decompose*	*Decompose 50%*	*Decompose 75%*	*Decompose*
BPIC 2012	BPIC2012	Yes	Yes	Yes	Yes
BPIC 2015	BPIC2015_1	No	Yes	Yes	Yes/No
	BPIC2015_2	No	Yes/No	Yes/No	Yes/No
	BPIC2015_3	No	Yes	Yes	Yes
	BPIC2015_4	No	Yes	Yes	Yes
	BPIC2015_5	No	Yes/No	Yes/No	Yes/No

connected parts of this net. The remainder of the net contains only disconnected transitions, indicating that the ILP Miner has had its problems with this real-life log. But where *Do not decompose* takes 1420 seconds (~24 minutes) to discover this (disappointing) result, *Decompose* takes only 56 seconds (less than a minute). As such, *Decompose* is obviously an improvement over *Do not decompose* for this log.

Conclusions. Do not decompose can only discover a net for the *BPIC 2012* log, whereas *Decompose* can discover a net for every log in the real-life data sets. Furthermore, for the only log that *Do not decompose* can handle successfully, *Decompose* is ~25 times as fast. As a result, *Decompose* is clearly always better than *Do not decompose* on the real-life data sets.

4.2.2. *Maximal-decomposition vs. non-maximal-decomposition*

First, we show for which logs all three configurations (*Decompose*, *Decompose 75%* and *Decompose 50%*) are

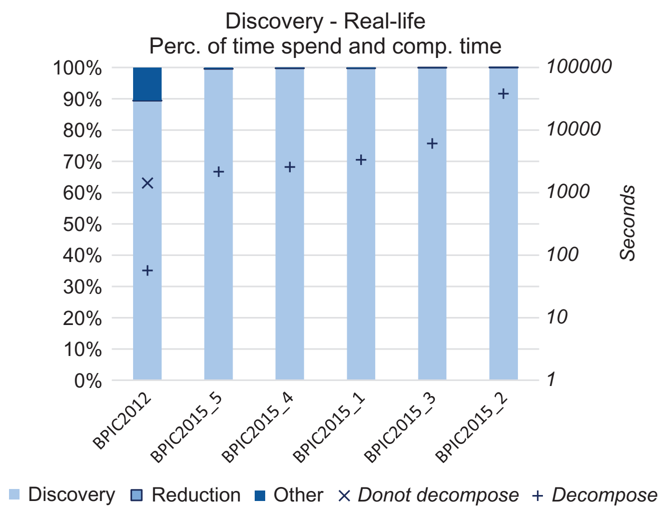


FIGURE 31. Categorized percentages of computation times for the real-life logs together with their computation times.

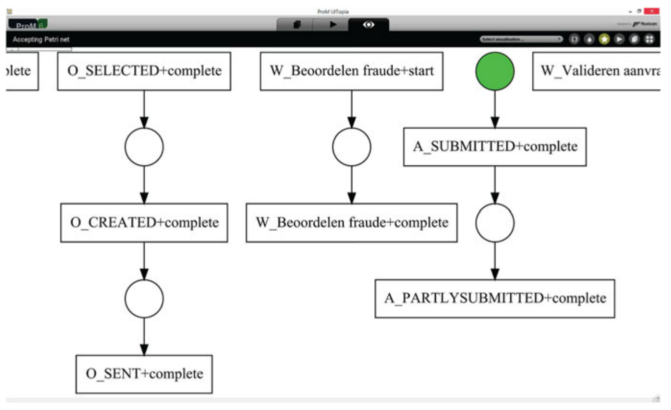


FIGURE 32. Connected parts of the accepting Petri net discovered using either *Decompose* or *Do not decompose*.

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 replay [22] is feasible, as again this is needed to compute the precision and generalization. Fourth, we show the precision and generalization values obtained using *Decompose* and the percentages obtained by the other two configurations. Finally, we summarize our findings.

Feasibility. Table 13 shows for which real-life logs *Decompose 50%* and *Decompose 75%* are feasible.As Table 13 shows, we can compare computation times for all real-life logs.

Computation times. Figure 33 shows the computation times required by *Decompose* on the real-life logs, and the speed-ups obtained using *Decompose 75%* and *Decompose 50%*.

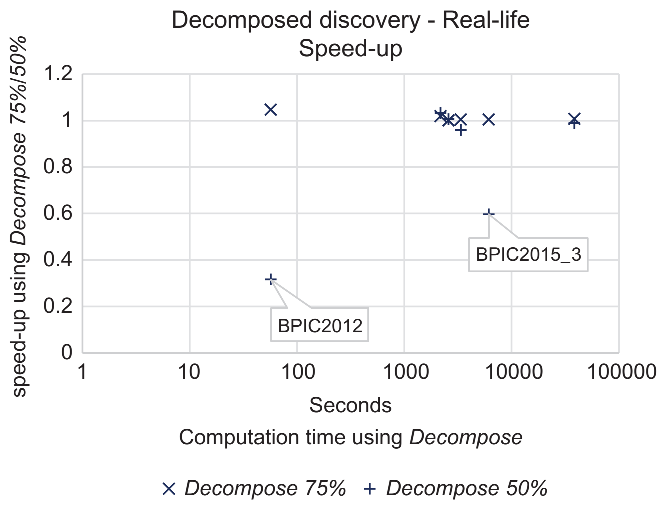


FIGURE 33. Comparison of decomposed computation times for real-life logs.

Apparently, there seems to be no real improvement in using either *Decompose 75%* or *Decompose 50%* over *Decompose*. In fact, there are two logs (*BPIC2012* and *BPIC2015_3*) for which *Decompose 50%* needs significantly more time than the other two configurations.

Feasibility of replay. Table 13 also shows for which of the feasible logs the replay is feasible.The replay for *Decompose 75%* and *Decompose 50%* runs out of memory for the *BPIC2015_2* log and out of time for the *BPIC2015_5* log. As a result, we can compare precision and generalization for all logs that are feasible with *Decompose*.

Precision and generalization. Both *Decompose 75%* as *Decompose 50%* typically return the same values for precision and generalization. The only exception to this is the precision of the net discovered from the *BPIC 2012* log using *Decompose 50%*, which drops to 97%. The latter is caused by the fact that *Decompose 50%* does not discover the place (see Fig. 32) between `O_SELECTED+complete` and `O_CREATED+complete`.

Conclusions. Both *Decompose 75%* and *Decompose 50%* can discover nets from the same real-life logs that *Decompose* can. Where *Decompose 75%* requires about the same time as *Decompose* requires, *Decompose 50%* may require significantly more. Apparently, some of the clusters become to big to be handled comfortably by the ILP Miner. Typically, both *Decompose 75%* and *Decompose 50%* preserve precision and generalization, although in one case the precision dropped to 97%.

4.3. Wrapping up

The three decomposed ILP Miners outperform the monolithic ILP Miner in two ways. First, the decomposed ILP miners can discover nets from logs on which the monolithic ILP Miner simply fails. As an example, the monolithic ILP Miner fails on five out of the six real-life logs, whereas the three decomposed ILP Miners succeed on all of them. Second, if the monolithic ILP Miner is able to discover a net, then the three decomposed ILP Miners can discover a net much faster (up to a factor 280). The net as discovered by a decomposed ILP Miner may be different (we will discuss this in the next section), but typically it will result in a net with equals or better precision, and equal or slightly worse generalization.

Using a non-maximal-decomposed ILP Miner has a positive effect for the smaller logs, but no effect or a negative effect for the larger logs. For some larger real-life logs, the negative effect may even be called considerable (about three times as slow). As such, starting with the maximal-decomposed ILP miner seems to be a good idea.

5. DISCUSSIONS

In the previous section, we have seen that sometimes decomposed discovery may result in a net that is different from the net as discovered without (or with less) decomposition. In this section, we first discuss the possible differences in the discovered nets. Second, we show that the effect may be positive. To demonstrate this, we discuss the *DrFurby Classifier* contribution to the 'Process Discovery Contest @ BPM 2016' [14]. This contest was won by the first author using the decomposition approach described here. From all submissions, most traces (193 out of 200) were classified correctly.

Furthermore, we have seen that the ILP Miner can be successfully decomposed using the tool framework. However, this may not hold for the other discovery algorithms, as these algorithms may have different characteristics. Therefore, we discuss the use of the tool framework with the other discovery algorithms as implemented in the tool framework. Finally, we conclude this section by summarizing our findings.

5.1. Differences

The decomposition approach as presented in [9] offers the following formal guarantees:

- *Perfect fitness is preserved*: The entire log is perfectly fitting the resulting merged net if *and only if* every sublog is perfectly fitting the subnet that was discovered from it.
- *Upper bound for misalignment costs*: The misalignment costs of replaying every sublog on the net that was discovered from it is an upper bound for the misalignment costs of replaying the entire log on the merged net. These misalignment costs are closely related to the fitness metric, as a trace is perfectly fitting if and only if these costs are 0.
- *Same fraction of perfectly fitting traces*: The fraction of traces from the entire log perfectly fitting the merged net equals the fraction of traces perfectly fitting every subnet.

The above guarantees prove that discovery of a perfectly fitting net can be successfully decomposed. Nevertheless, the approach does not guarantee that the result of the decomposed discovery will be the same as the result from the monolithic discovery, *even when* all discovered nets are perfectly fitting. Recall that the decomposition splits the entire log into a collection of sublogs, and that it calls on the discovery algorithm for every sublog. Clearly, the discovery algorithm can only use the information that is contained in the provided sublog, as the information contained in the other sublogs is withheld from it. As such, it might take different decisions than it would have if all information would be available. Examples of such situations are shown in Figs 24 and 25. Nevertheless, as mentioned earlier and as we will show with the following process discovery contest, the differences may just be for the better.

5.2. Process discovery contest

The aim of the 'Process Discovery Contest @ BPM 2016' [14] was to evaluate the state-of-the-art in process discovery. To this end, the organizers of the contest created 10 process models, say $M_1,\ldots,M_{10}$, which were not disclosed to the contestants. For every created process model M_i (with $i \in \{1,\ldots,10\}$), the organizers also created two event logs: a $March_i$ training log containing 1000 traces, which was disclosed in March 2016, and a $June_i$ test log containing 20 traces, which was disclosed in June 2016. The test log also contained negative cases, that is, traces impossible according to the original model.

A submission should include a discovery algorithm D and a way to classify the traces of every $June_i$ log as positive (perfectly fit M_i) or negative (do not perfectly fit M_i), using the model as discovered by algorithm D from the $March_i$ log. Using the undisclosed process model M_i, the organizers then determine how many traces from every $June_i$ log are correctly classified by the submission. In the end, the submission which classifies the most traces over all $June_i$ logs correctly wins the contest. In case of a tie, the time required for D to discover the models tip the balance. Clearly, the better the 10 discovered process models match the 10 undisclosed process models, the better the classification.

To allow the contestants to test their submission prior to submitting it, the organizers also created for every undisclosed process model M_i an $April_i$ test log (disclosed in April 2016) and a May_i test log (disclosed in May 2016). Both the $April_i$ and the May_i test log are known to contain 10 positive traces and 10 negative traces. By using this information, the contestants can test and calibrate their submission.

5.2.1. *The* DrFurby Classifier

The first author participated in this contest using the decomposition approach presented in this paper. The basic idea behind the *DrFurby Classifier* [23] is (i) to minimize the number of false negatives (i.e. positive traces that are classified as negative traces) by only including algorithms that guarantee perfect fitness and (ii) to minimize the number of false positives (i.e. negative traces that are classified as positive traces) by including many of these algorithms. However, to minimize the time required by it, the *DrFurby Classifier* includes only a minimal set of these algorithms that provide the maximal result.

Examples of relevant algorithms that guarantee perfect fitness include:

- *Inductive Miner*: The 'Inductive Miner Infrequent' with noise threshold set to 0.0 [5].
- *ILP Miner*: The 'ILP Miner' [4].
- *Hybrid ILP Miner (Default)*: The 'Hybrid ILP Miner' with default settings [17].
- *Hybrid ILP Miner (Single)*: The 'Hybrid ILP Miner' that uses only a single variable per causal relation [17].

Next to these discovery algorithms, the *DrFurby Classifier* also exploits the tool framework presented in this paper, as it also *preserves* (perfect) fitness [9].

The *DrFurby Classifier* takes a $March_i$ log and a $June_i$ log as input, and creates a classified $June_i^+$ log as output. To create the output from the inputs, it iteratively uses a number of perfect-fitness-guaranteeing discovery algorithms with different decomposition settings. For each combination of a discovery algorithm D and a setting S, it first discovers an accepting Petri net $D_S(March_i)$ using the framework. Second, it checks which traces of the $June_i$ log can be perfectly replayed on $D_S(March_i)$ [22]. If a trace is perfectly replayed on all such combinations, it is classified as positive by the *DrFurby Classifier*, otherwise, it is classified as negative.

Configuration. The $April_i$ and May_i logs were used to determine a minimal set of (possibly decomposed) discovery algorithms that provides maximal result. For this sake, all discovery algorithms mentioned earlier were tested, and various decomposition settings (Do not decompose, maximal decomposition, Decompose by 80%, by 60%, by 40%, ...). In the end, two decomposed discovery algorithms were found that provide the desired maximal result:

- The Inductive Miner with maximal decomposition (called D_{100}^{IM} henceforth).
- The Hybrid ILP Miner without decomposition (called D_0^{HIM} henceforth).

As a result, the *DrFurby Classifier* is configured with only these two decomposed discovery algorithms, leading to a discovery algorithm D^{Furby}.

Implementation. The *DrFurby Classifier* is implemented as the *DrFurby Classifier* plug-in in ProM 6.6, where it can be found in the *Divide And Conquer* package.

To enrich the classified test log with the necessary classification attributes, a *DrFurby Extension* is implemented as well (see [2] for details on log extensions), which uses the prefix `drfurby`. This extension defines attributes as listed in Table 14.

As a result, by inspecting the output log, the user can see how many traces are classified positive (negative), which traces are classified positive (negative), etc. Furthermore, to get a quick overview of which traces are classified positive, the *DrFurby Classifier* appends a plug sign (+) to the name of every trace that is classified positive.

Results. Table 15 shows the number of traces classified correctly by the *DrFurby Classifier* for all $April_i$, May_i and $June_i$ logs. As a result, for the $June_i$ logs, the *DrFurby Classifier* classifies 193 out of 200 traces correctly. In addition, Fig. 34 shows two views on the resulting classified log $June_3^+$. These views show that 13 out of 20 traces are classified as positive, and that only seven are classified as negative.

5.2.2. *Maximal decomposition vs. no decomposition*

Instead of using the Inductive Miner with maximal decomposition, the *DrFurby Classifier* could also been configured

TABLE 14. The attributes of the *DrFurby Extension* which are added to the output log to allow the user to inspect the classification results.

Attribute	Level	Type	Description
classification	trace	String	Classification of the trace ('positive' or 'negative')
him0Costs	trace	Continuous	The costs of replaying this trace on the accepting Petri net as discovered by the D_0^{HIM} discoverer
im100Costs	trace	Continuous	The costs of replaying this trace on the accepting Petri net as discovered by the D_{100}^{IM} discoverer
millis	log	Discrete	The number of milliseconds it takes to classify the test log
name	log	Literal	The name of the test log
negative	log	Discrete	The number of traces in the test log classified negative
positive	log	Discrete	The number of traces in the test log classified positive
totalCosts	trace	Continuous	The accumulated costs of replaying this trace on all discovered accepting Petri nets

TABLE 15. Numbers of traces classified correctly by the *DrFurby Classifier*. Note that the optimal answer is 20 in all cases, hence it perfectly classifies 22 out of 30 logs.

i	1	2	3	4	5	6	7	8	9	10
$April_i$	20	20	20	20	20	20	20	19	20	20
May_i	20	20	18	20	19	20	20	18	20	20
$June_i$	20	20	17	20	20	19	19	18	20	20

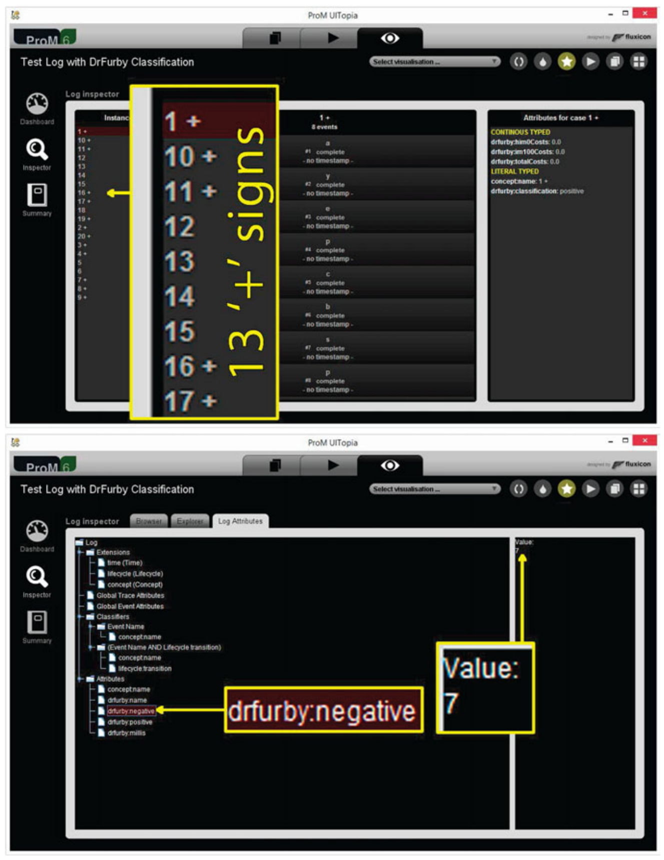


FIGURE 34. Two views on the results of the *DrFurby Classifier* for the $June_3$ log. The top view shows that the traces 1–4, 7–11, 16, 17, 19 and 20 are classified as positive. The bottom view shows that 7 traces are classified as negative.

with the Inductive Miner *without decomposition* (called D_0^{IM} henceforth). For 9 out of the 10 $March_i$ logs, this would not make a difference, as the results would be the same ($D_0^{IM}(March_i) = D_{100}^{IM}(March_i)$). However, for $March_3$, it would make a difference.

Table 16 shows the classification results on the May_3 log. This table shows that the *DrFurby Classifier* classifies eight traces as negative. Furthermore, it shows that if we would have used D_0^{IM} instead of D_{100}^{IM}, it would only classify five traces as negative. As a result, using the maximal-decomposed Inductive Miner (D_{100}^{IM}) instead of the monolithic Inductive Miner (D_0^{IM}), the *DrFurby Classifier* correctly classifies three more traces as negative. For the $June_3$ log, two more traces (7 instead of 5) are correctly classified as negative.

As a result, the *DrFurby Classifier* using a maximal-decomposed Inductive Miner works better (7 false positives on all $June_i$ logs, no false negatives) than it would using the monolithic Inductive Miner (9 false positives on all $June_i$ logs, no false negatives). That is, with decomposition, the *DrFurby Classifier* classifies 193 out of 200 traces correctly, whereas without decomposition it would only have classified 191 correctly.

In the end, this improvement from 191 to 193 made the *DrFurby Classifier* win [14] the contest, as the two runner-ups in the contest classify both 192 traces correctly. One runner-up did not use any decomposition techniques, but the other runner-up actually used the ILP Miner with maximal decomposition as supported by the tool framework for some logs (like the $March_3$ log). For the other logs (like the $March_1$ log), the second runner-up used the Inductive Miner without decomposition. As a result, in the top three of the contest, two submissions actually use the tool framework presented in this paper, and one of them won. Although the main goal of the paper is the feasibility and speed-ups of the decomposed approach, this result shows the competitive value of the approach on the quality perspective.

5.3. Other discovery algorithms

As shown earlier in Table 9, six different discovery algorithms are implemented in the tool framework. The previous

TABLE 16. Classification (+ = positive, − = negative) of the May_3 log using the *DrFurby Classifier*. The third row (D^{Furby}) can be viewed as the conjunction of the first two rows. Clearly, the Inductive Miner with maximal decomposition (D_{100}^{IM}) complements D_0^{HIM} much better than the Inductive Miner without decomposition (D_0^{IM}).

Trace	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
D_{100}^{IM}	+	−	−	−	+	+	+	+	+	−	−	+	+	+	+	+	−	−	+	+
D_0^{HIM}	+	−	+	−	+	+	+	+	+	+	−	+	+	+	+	−	+	−	+	+
D^{Furby}	+	−	−	−	+	+	+	+	+	−	−	+	+	+	+	−	−	−	+	+
D_0^{IM}	+	−	+	−	+	+	+	+	+	+	−	+	+	+	+	−	+	−	+	+

section has shown that one of them, the ILP Miner [4], can be successfully decomposed using the framework to significantly speed up analysis. But as the other five algorithms may have different characteristics, these positive results may not transfer to them. In this section, we discuss the use of the tool framework on these other five algorithms. The running times as reported in this section are obtained on the same computer as we did the evaluation in the previous section on.

Alpha Miner. The Alpha Miner [3] was the first discovery algorithm able to discover concurrency in process models. Under certain preconditions, the Alpha Miner can rediscover the net from the event log successfully (see [3]). Although the Alpha Miner is generally conceived to be a very simple and straightforward algorithm, its complexity is not that simple. Especially, the step that computes the maximal activity sets A_1 and A_2 such that (i) every activity in A_1 can be directly followed by any activity in A_2, (ii), every activity in A_1 cannot be directly followed by any activity in A_1, and (iii) every activity in A_2 cannot be directly followed by any activity in A_2, is potentially very time-consuming when there are many different activities. As a result, although for many event logs, decomposing the Alpha Miner may not result in a speed-up, for some event logs, it could. As an example, we take the *prGm6* log from the *BPM 2013* [11] data set. To discover the net from this log takes the monolithic Alpha Miner ~800 seconds. The maximally decomposed Alpha Miner decomposes this log into 57 sublogs, and to discover the 57 subnets takes only ~87 seconds. This shows that the Alpha Miner may benefit from the decomposition. However, the overhead of the decomposed Alpha Miner spoils this gain of >700 seconds as it takes *days* (if not *weeks* or worse) to reduce the resulting merged net (Section 3.4.3). The main bottleneck is the algorithm used to reduce the structural redundant places [21], which *uses ILPs* to compute these structural redundant places.

Heuristics Miner. The Heuristics Miner [16] is a very efficient discovery algorithm, with a very low complexity. For this discovery algorithm, we do not expect any speed-ups when using decomposition. As an example, it takes the monolithic Heuristics Miner ~5 seconds to discover a net from the *prGm6* log, whereas the maximally decomposed Heuristics Miner takes almost 95 seconds, of which it needs ~36 seconds to discover the 57 subnets. Hence, although on average the Heuristics Miner takes less time to discover a single subnet, it takes more to discover them all, and the overhead of the decomposition is significant.

Hybrid ILP Miner. Like the ILP Miner, the Hybrid ILP Miner [17] uses ILPs to solve the discovery problem. Although (i) the actual ILP when using both discovery algorithms may be different, and (ii) the Hybrid ILP Miner is typically faster, we expect the Hybrid ILP Miner to also benefit from the decomposition. If we take the *prGm6* log again as example, the monolithic Hybrid ILP Miner crashes as the ILP solver runs out of memory in 3 hours, whereas the maximally decomposed Hybrid ILP miner finishes after ~60 000 seconds (~16 hours and 40 minutes). Hence, although the maximally decomposed Hybrid ILP Miner takes >16 hours, at least it does discover a net, whereas the monolithic Hybrid ILP Miner fails to do so

Inductive Miner. The Inductive Miner [5] is the youngest member of the process discovery family, and has quickly grown to be the most often used process discovery algorithm in ProM. In its way, this discovery algorithm uses a divide-and-conquer approach to tackle the discovery problem at hand. Nevertheless, it requires the algorithm some efforts to decide which division is best at some point in time. As a result, it takes the monolithic Inductive Miner ~29 seconds to discover a net from the *prGm6* log, whereas the maximally decomposed Inductive Miner takes almost 52 seconds, of which it needs ~33 seconds to discover the 57 best clusters, ~8 seconds to discover the 57 subnets, and 6 seconds to do the necessary reductions. As such, the Inductive Miner may benefit as well from the decomposition, but like with the Heuristics Miner, the overhead of the decomposition may outweigh the benefits.

Evolutionary Tree Miner. The Evolutionary Tree Miner [18] is a genetic discovery algorithm, which uses replay [22] to check the quality of a discovered net. Being an evolutionary algorithm, the higher the quality of a net, the higher the chances that it survives. As such, provided sufficient space and time, it will provide a high-quality net, provided that such a net exists. However, it often lacks the sufficient time, as by default it stops after 10 minutes, after which it returns the best net found so far. Although from a user-perspective this deadline of 10 minutes is understandable, it will typically be too short for the algorithm to return a high-quality net. On the examples as introduced in the previous section, this discovery algorithm will typically stop because of this 10-minute deadline. As a result, it will take 10 minutes on the *prGm6* log, whereas the maximally decomposed Evolutionary Tree miner will take at least 570 minutes (10 minutes for every of the 57 sublogs). As such, decomposing this discovery algorithm only makes sense if the deadline is set to a far larger value than 10 minutes, or if it is removed altogether. The latter forces the discovery algorithm to stop on another stopping criteria, which all have to do with the quality of the best net.

5.4. Summarizing the findings

The *DrFurby Classifier* [23] uses the tool framework and was the winning submission for the 'Process Discovery Contest

@ BPM 2016' [14]. By replacing the Inductive Miner without decomposition by the Inductive Miner with maximal decomposition, we effectively reduced the number of errors (misclassified traces) by the *DrFurby Classifier* from 9 to 7 in the final test logs. If we also include the calibration logs, then the number of errors is reduced from 21 to 13. This shows that the tool framework has effectively improved the effectiveness of the *DrFurby Classifier*, and has made this submission the winning submission [14], as the two runner-ups both misclassified eight traces.

Discovery algorithms other than the ILP Miner can benefit from the tool framework as well, including the Alpha Miner [3], the Inductive Miner [5] and the Hybrid ILP Miner [17]. For all these algorithms, discovering the subnets from the sublogs may take less time than discovering a net from the entire log.

However, the overhead of the tool framework may spoil any computation time benefits obtained by the decomposition for the first two. We have seen that for the Alpha Miner the net reduction (Section 3.4.3) may be a problem. Apparently, the nets as discovered by the Alpha Miner may be problematic for the implemented reduction techniques, especially those techniques that use ILPs. For the Inductive Miner, the reduction techniques are less of a problem, but the discovery of the best cluster may take too long to benefit from the decomposition. The Heuristics Miner [16] does not benefit from the decomposition, as its complexity is very low. The Evolutionary Tree Miner [18] takes so much time that is typically capped with a 10-minute deadline to have it deliver a result within reasonable time. Using decomposition with this algorithm only makes sense if time is not the limiting factor in the discovery.

6. CONCLUSIONS

This paper presents the *Divide And Conquer* framework. This framework fully supports the decomposed discovery as introduced in [9], and has been implemented in ProM 6. As such, the framework allows for easy decomposed discovery, using existing discovery algorithms. 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.

Adding a new miner to the framework is easy, provided that the miner either results in (i) a net with an explicit initial marking and an explicit set of final markings, or (ii) 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 ILP Miner [4] benefits clearly from the framework. Logs that take the ILP Miner more than a week, can be discovered within half an hour with a decomposed ILP Miner. This shows that decomposition indeed can speed up a complex discovery algorithm significantly. Other discovery algorithms may also benefit, like the Hybrid ILP Miner [17], the Alpha Miner [3] and the Inductive Miner [5]. However, for the latter two algorithms, any computation time benefit obtained by the decomposition may be spoiled by the overhead of the decomposition. As a result, we need to investigate whether we can reduce this overhead for these algorithms.

It is a fact that using decomposed discovery may lead to different results. However, the *DrFurby Classifier* [23] submission to the 'Process Discovery Contest @ BPM 2016' [14] shows that this may very well have a positive effect. By making the state-of-the-art Inductive Miner [5] decomposed in this submission, the number of misclassified traces could be reduced from 9 to 7, thereby winning the contest [14] as the two runner-ups had both eight misclassifications.

Although this approach does not provide any guarantees, it clearly shows that the quality of the nets (defined as the percentage of correctly classified traces) obtained by decomposed discovery may exceed the quality of the nets as obtained by non-decomposed discovery. However, there is guarantee for such a positive effect.

Because of the formal guarantees as provided by the decomposition, the results of the decomposed discovery provide a valuable and reliable alternative. As examples, 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.

Future work on the framework includes additional non-maximal decomposition algorithms and possible improvements on the imposed 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.

REFERENCES

[1] van der Aalst, W.M.P. (2016) *Process Mining: Data Science in Action* (2nd edn). Springer.

[2] Günther C.W. and Verbeek H.M.W. (2014) XES Standard Definition. Technical Report BPM-14-09.

[3] van der Aalst, W.M.P., Weijters, A.J.M.M. and Maruster, L. (2004) Workflow mining: discovering process models from event logs. *IEEE Trans. Knowl. Data Eng.*, **16**, 1128–1142.

[4] van der Werf, J.M.E.M., van Dongen, B.F., Hurkens, C.A.J. and Serebrenik, A. (2009) Process discovery using integer linear programming. *Fundam. Inf.*, **94**, 387–412.

[5] Leemans, S.J.J., Fahland, D., and van der Aalst, W.M.P. (2013) Discovering Block-Structured Process Models from Event Logs—A Constructive Approach. In Colom, J.-M. and Desel, J. (eds.), *Application and Theory of Petri Nets and Concurrency*, Lecture Notes Computer Science, 7927, pp. 311–329.

[6] Dumas, M., van der Aalst, W.M.P. and ter Hofstede, A.H.M. (2005) *Process-Aware Information Systems: Bridging People and Software Through Process Technology*, Wiley & Sons.

[7] Munoz-Gama, J., Carmona, J. and van der Aalst, W.M.P. (2014) Single-entry single-exit decomposed conformance checking. *Inf. Syst.*, **46**, 102–122.

[8] Verbeek, H.M.W., Buijs, J.C.A.M., van Dongen, B.F. and van der Aalst, W.M.P. (2010) ProM 6: The Process Mining Toolkit. *Proc. BPM Demonstration Track 2010*, pp. 34–39. CEUR-WS.org.

[9] van der Aalst, W.M.P. (2013) Decomposing Petri nets for process mining: a generic approach. *Distrib. Parallel Database*, **31**, 471–507.

[10] Maruster, L., Weijters, A.J.M.M., van der Aalst, W.M.P. and van den Bosch, A. (2006) A rule-based approach for process discovery: dealing with noise and imbalance in process logs. *Data Min. Knowl. Discov.*, **13**, 67–87.

[11] Munoz-Gama, J., Carmona, J. and van der Aalst, W.M.P. (2013) Conformance Checking in the Large: Partitioning and Topology. In Daniel, F., Wang, J. and Weber, B. (eds), *Proc. 11th Int. Conf. Business Process Management (BPM 2013)*, Beijing, China, August 26–30, Lecture Notes Computer Science, 8094, pp. 130–145.

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

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

[14] Carmona, J., de Leoni, M., Depaire, B. and Jouck, T. Process discovery contest @ BPM 2016. https://www.win.tue.nl/ieeetfpm/lib/exe/fetch.php?media=shared:downloads:processdiscoverycontest.pdf (accessed October 13, 2016).

[15] Girault, C. and Valk, R. (2001) *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications*, Springer New York, Inc., Secaucus, NJ, USA.

[16] Weijters, A.J.M.M. and van der Aalst, W.M.P. (2003) Rediscovering workflow models from event-based data using Little Thumb. *Integr. Comput. Aided Eng.*, **10**, 151–162.

[17] van Zelst S.J., van Dongen B.F. and van der Aalst W.M.P. (2015) ILP-Based Process Discovery using Hybrid Regions. *Proc. Int. Workshop on Algorithms & Theories for the Analysis of Event Data, ATAED 2015, Satellite event of the conferences: 36th Int Conf. Application and Theory of Petri Nets and Concurrency Petri Nets 2015 and 15th Int. Conf. Application of Concurrency to System Design (ACSD 2015)*, Brussels, Belgium, June 22–23, pp. 47–61. CEUR-WS.org.

[18] Buijs J.C.A.M. (2014) Flexible evolutionary algorithms for mining structured process models. PhD Thesis, Eindhoven University of Technology.

[19] Hompes, B.F.A., Verbeek, H.M.W. and van der Aalst, W.M.P. (2015) Finding Suitable Activity Clusters for Decomposed Process Discovery. In Ceravolo, P., Russo, B. and Accorsi, R. (eds), *Data-Driven Process Discovery and Analysis: 4th Int. Symposium, SIMPDA 2014*, Milan, Italy, November 19–21, Revised Selected Papers, Lecture Notes Business Information, 237, pp. 32–57.

[20] Murata, T. (1989) Petri nets: properties, analysis and applications. *Proc. IEEE*, **77**, 541–580.

[21] Berthelot, G. (1987) Transformations and Decompositions of Nets. In Brauer, W., Reisig, W. and Rozenberg, G. (eds), *Advances in Petri Nets 1986 Part I: Petri Nets, Central Models and their Properties*, Lecture Notes Computer Science, 254, pp. 360–376.

[22] van der Aalst, W.M.P., Adriansyah, A. and van Dongen, B.F. (2012) Replaying history on process models for conformance checking and performance analysis. *Data Min. Knowl. Discov.*, **2**, 182–192.

[23] Verbeek, H.M.W. and Mannhardt, F. (2016) The DrFurby Classifier submission to the Process Discovery Contest @ BPM 2016. BPM Center Report BPM-16-08. BPMCenter.org.