Decomposed Process Mining: The ILP Case

H.M.W. Verbeek and W.M.P. van der Aalst

Department of Mathematics and Computer Science,
Eindhoven University of Technology, Eindhoven, The Netherlands
{h.m.w.verbeek,w.m.p.v.d.aaalst}@tue.nl

Abstract. Over the last decade process mining techniques have matured and more and more organizations started to use process mining to analyze their operational processes. The current hype around "big data" illustrates the desire to analyze ever-growing data sets. Process mining starts from event logs—multisets of traces (sequences of events)—and for the widespread application of process mining it is vital to be able to handle "big event logs". Some event logs are "big" because they contain many traces. Others are big in terms of different activities. Most of the more advanced process mining algorithms (both for process discovery and conformance checking) scale very badly in the number of activities. For these algorithms, it could help if we could split the big event log (containing many activities) into a collection of smaller event logs (which each contain fewer activities), run the algorithm on each of these smaller logs, and merge the results into a single result. This paper introduces a *generic framework* for doing exactly that, and makes this concrete by implementing algorithms for decomposed process discovery and decomposed conformance checking using Integer Linear Programming (ILP) based algorithms. ILP-based process mining techniques provide precise results and formal guarantees (e.g., perfect fitness), but are known to scale badly in the number of activities. A small case study shows that we can gain orders of magnitude in run-time. However, in some cases there is tradeoff between run-time and quality.

Key words: Process discovery, conformance analysis, big data, decomposition

1 Introduction

The current attention for "big data" illustrates the spectacular growth of data and the potential economic value of such data in different industry sectors [1, 2]. Most of the data that are generated refer to *events*, e.g., transactions in financial systems, interactions in a social network, events in high-tech systems or sensor networks. The incredible growth of event data provides new opportunities for process analysis. As more and more actions of people, organizations, and devices are recorded, there are ample opportunities to analyze processes based on the footprints they leave on event logs. In fact, we believe that the analysis of purely *hand-made* process models will become less important given the omnipresence of event data [3].

Process mining aims to *discover*, *monitor*, and *improve* real processes by *extracting knowledge* from *event logs* readily available. Starting point for any process mining task is an event log. Each event in such an event log refers to an *activity* (i.e., a well-defined

step in some process) and is related to a particular *case* (i.e., a *process instance*). The events belonging to a case are *ordered* and can be seen as one "run" of the process. It is important to note that an event log contains only *example behavior*, i.e., we cannot assume that all possible runs have been observed. In fact, an event log often contains only a fraction of possible behavior [3].

Petri nets are often used in the context of process mining. Various algorithms employ Petri nets as the internal representation used for process mining. Examples are the region-based process discovery techniques [4, 5, 6, 7, 8], the α-algorithm [9], and various conformance checking techniques [10, 11, 12, 13]. Other techniques use alternative internal representations (C-nets, heuristics nets, etc.) that can easily be converted to (labeled) Petri nets [3].

In this paper, we present a generic framework for *decomposing* the following two main process mining problems:

Process discovery: Given an event log consisting of a collection of traces, construct a Petri net that "adequately" describes the observed behavior.

Conformance checking (or replay): Given an event log and a Petri net, diagnose the differences between the observed behavior (the event log) and the modeled behavior (the Petri net) by replaying the observed behavior on the model.

To exemplify the use of this framework, we have implemented a decomposed discovery algorithm and a decomposed replay algorithm on top of it that both use ILP-based techniques [8, 10]. We have chosen these ILP-based techniques as they provide formal guarantees and precise results. Since ILP-based techniques scale badly in the number of activities, there is the desire to speed-up analysis through smart problem decompositions.

The remainder of this paper is organized as follows. Section 2 briefly introduces basic concepts like event logs and Petri nets. Section 3 presents the generic framework, which consists of a collection of objects (like event logs and Petri nets) and a collection of algorithms (to import, export, visualize these objects, and to be able to create new objects from existing objects). Section 4 introduces the specific ILP-based discovery and replay algorithms implemented using our generic framework. Section 5 introduces a small case study, which shows that we can achieve better run-times, but that there are also tradeoffs between speed and quality. Section 6 concludes the paper.

2 Preliminaries

This section introduces basic concepts such as event logs, accepting Petri nets, discovery algorithms, log alignments, and replay algorithms.

2.1 Event logs

Event logs are bags (or multisets) of event sequences (or traces). Events in an event log may have many attributes (like the activity, the resource who executed the activity, the timestamp the execution was completed, etc.). In the context of this paper, we are only

Table 1. Activity log L_1^A in tabular form.

name	trace	frequency
c_1	$\langle a_1, a_2, a_4, a_5, a_6, a_2, a_4, a_5, a_6, a_4, a_2, a_5, a_7\rangle$	1
c_2	$\langle a_1, a_2, a_4, a_5, a_6, a_3, a_4, a_5, a_6, a_4, a_3, a_5, a_6, a_2, a_4, a_5, a_7\rangle$	1
c_3	$\langle a_1, a_2, a_4, a_5, a_6, a_3, a_4, a_5, a_7\rangle$	1
c_4	$\langle a_1, a_2, a_4, a_5, a_6, a_3, a_4, a_5, a_8\rangle$	2
c_5	$\langle a_1, a_2, a_4, a_5, a_6, a_4, a_3, a_5, a_7\rangle$	1
c_6	$\langle a_1, a_2, a_4, a_5, a_8\rangle$	4
c_7	$\langle a_1, a_3, a_4, a_5, a_6, a_4, a_3, a_5, a_7\rangle$	1
c_8	$\langle a_1, a_3, a_4, a_5, a_6, a_4, a_3, a_5, a_8\rangle$	1
c_9	$\langle a_1, a_3, a_4, a_5, a_8\rangle$	1
c_{10}	$\langle a_1, a_4, a_3, a_5, a_8\rangle$	1
c_{11}	$\langle a_1, a_4, a_2, a_5, a_6, a_4, a_2, a_5, a_6, a_3, a_4, a_5, a_6, a_2, a_4, a_5, a_8\rangle$	1
c_{12}	$\langle a_1, a_4, a_2, a_5, a_7\rangle$	3
c_{13}	$\langle a_1, a_4, a_2, a_5, a_8\rangle$	1
c_{14}	$\langle a_1, a_4, a_3, a_5, a_7\rangle$	1

interested in the activity an event refers to and abstract from other information. For this, we introduce the notion of a *classifier*, which maps every event onto its corresponding activity. As a result, we can map an entire trace onto an activity sequence, and an event log onto an activity log, where activity logs are bags of activity sequences.

Table 1 describes the activity log $L_1^A = [c_1, c_2, c_3, {c_4}^2, c_5, {c_6}^4, c_7, c_8, c_9, c_{10}, c_{11}, {c_{12}}^3, c_{13}, c_{14}]$ defined over $A_1 = \{a_1, \ldots, a_8\}$. Activity sequence $c_6 = \langle a_1, a_2, a_4, a_5, a_8\rangle$ occurs 4 times in L_1^A.

In the remainder of this paper, we will still often use the term event log, but we will use it often in conjunction with a classifier, which induces an activity log that will be used by the discovery and replay algorithms.

2.2 Accepting Petri nets

For discovery algorithms, labeled Petri nets would suffice. However, for replay algorithms, we also need information on the initial marking of the net and of its possible final (or accepting) markings. For this reason, we introduce the concept of accepting Petri nets, which correspond to labeled Petri nets with an initial marking and a collection of final markings.

For example, Fig 1 shows an accepting Petri net $N_1 = (P_1, T_1, F_1, l_1, \rhd_1, \Box_1)$ over A_1, where $\rhd_1 = [p_1]$ and $\Box_1 = \{[p_{10}]\}$. As a result of the labeling, the transitions t_4, t_6, and t_9 are invisible. The firing sequence $\langle t_1, t_2, t_4, t_5, t_7, t_{10}\rangle$ runs from the initial marking to the only final marking and generates the trace $\langle a_1, a_2, a_4, a_5, a_7\rangle$.

2.3 Discovery algorithms

A discovery algorithm [9, 4, 5, 6, 7, 8] is an algorithm that takes an event log (with classifier) as input, and generates an accepting Petri net as output. It is typically assumed that the behavior of the resulting accepting Petri net *fits* the behavior as captured

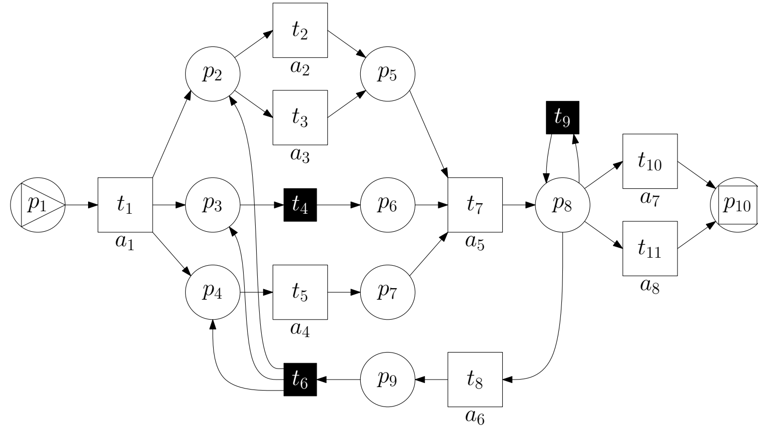

Fig. 1. An example accepting Petri net $N_1 = (P_1, T_1, F_1, l_1, \triangleright, \square)$.

by the event log in some way. However, also other quality dimensions like precision, generalization and simplicity need to be considered [3].

2.4 Log alignments

Log and trace alignments are used by the replay algorithms to match the trace at hand with a valid firing sequence from the accepting Petri net. For this reason, a *trace alignment* contains a series of *moves*, which can be divided into three types:

Synchronous move: The trace and the net agree on the next action, as the next event (activity) in the trace matches an enabled transition in the net.

Log move: The next action is the next activity in the trace, which cannot be mimicked by the net.

Model move: The next action is an enabled transition in the net, which is not reflected by an event in the log.

A trace alignment is *valid* if the sequence of activities in synchronous and log moves equals the trace, if all transitions in synchronous moves have the corresponding activity as label, and if the sequence of transitions in synchronous and model moves equals some valid firing sequence in the net (which starts in the initial marking and ends in some final marking). We cannot always guarantee this latter requirement (valid firing sequence) when merging trace alignments. A trace alignment for which only the first two requirements hold is called a *weak* trace alignment. A *log alignment* maps every trace of the log onto a trace alignment.

2.5 Replay algorithms

A replay algorithm is an algorithm that takes an event log (with classifier) and an accepting Petri net as input, and generates a log alignment as output. Typically, a replay algorithm assigns costs to the different moves. These costs are configurable. However,

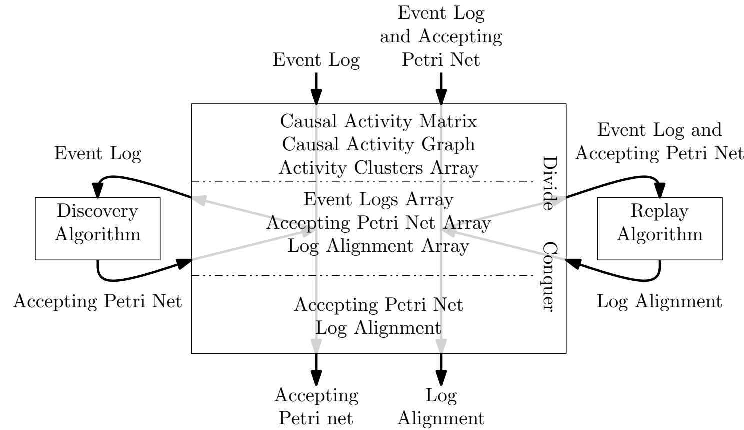

Fig. 2. Framework overview.

in this paper we use the following default costs. Synchronous moves have cost 0 as these correspond to a perfect match. A log move costs 5 and a visible model move costs 2. Model moves corresponding to invisible transitions have cost 0 (as these are not captured by the log) [10].

3 Generic Divide and Conquer Framework

This section introduces the generic framework for decomposed discovery and replay. Key ingredients for the framework are: *objects* (matrices, graphs, cluster arrays, etc.) and *algorithms* to process these objects (importing, exporting, visualizing, and creating a new object from existing objects) [14].

Figure 2 shows an overview of the framework, which contains many of the objects supported by the framework. The algorithms in the framework allow the user to create new objects from existing objects. The Discovery and Replay Algorithms on the sides symbolize existing discovery and replay techniques that can be connected to the framework.

3.1 Objects

To decompose an event log or an accepting Petri nets into sublogs or subnets, we need to divide the set of activities into so-called *clusters*. Then, we can create a sublog and a subnet for every cluster. There are different approaches that can be used to create an array of clusters. In this paper, we take a three-step approach:

1. Discover from the event log (or create from the accepting Petri net) a *causal activity matrix*. This matrix M maps every combination of two activities onto a real number in $[-1.0, 1.0]$, where $M[a, a'] \approx 1.0$ means that we are quite sure that there is a causal dependency from a to a', $M[a, a'] \approx -1.0$ means that we are sure that there

is no causal dependency from a to a', and $M[a, a'] \approx 0.0$ means that we do not really know whether or not there is a causal dependency from a to a'.
2. Create a causal dependency graph from this matrix by taking the causal dependencies of which we are most certain.
3. Create an array of clusters from this graph using the technique as described in [14].

After having obtained the clusters, we can decompose the event log and accepting Petri net in an *event log array* and an *accepting Petri net array*, where the i-th sublog and subnet correspond to the i-th cluster in the array.

For the discovery case, we can then run the target discovery algorithm on every sublog, and merge the resulting subnets into a single accepting Petri net.

For the replay case, we can then replay every sublog on the corresponding subnet, and merge the resulting log alignments into a single log alignment. Although this approach provides various formal guarantees (e.g., the fraction of fitting cases computed in this manner is exact [14]), there may be some complications:

- Experiments have shown [15] that replaying a sublog on a subnet *which is obtained by removing all transitions that do not correspond to an activity in the cluster* can take more time than replaying the original log on the original net. For this reason, the framework also supports subnets that are obtained by making all transitions that do not correspond to an activity in the cluster *invisible*. This also reduces the number of activities in the subnets, but keeps the structure of the net intact.
- The current implementation of the replay algorithms uses integer costs. This is a problem, as we need to divide the costs of an activity evenly over the clusters where the activity appears (see [14]). If the activity appears in three clusters, how can we divide the costs of its log move (5) evenly over the clusters? For this reason, we introduce a *cost factor* in the framework. Every cost in the replay problem will first be multiplied by this factor, and the replay algorithms will run using these multiplied costs.

3.2 Algorithms

All objects in the framework can be exported, imported, and visualized. Furthermore, it is possible to create new objects from existing objects. Some of the algorithms supported in our framework:

Create Accepting Petri Net: Create an accepting Petri net from a Petri net.
Create Activity Clusters: Create a (valid maximally decomposed [14]) activity cluster array from an accepting Petri net.
Create Clusters: Create an activity cluster array from a causal activity graph.
Create Matrix: Create a causal activity matrix from an accepting Petri net.
Determine Activity Costs: Determine the cost factor for an activity cluster array.
Discover Accepting Petri Nets: Discover an accepting Petri net array from an event log array using a (wrapped) existing discovery algorithm.
Discover Clusters: Discover an activity cluster array from an event log.
Discover Matrix: Discover a causal activity matrix from an event log.
Filter Graph: Filter a causal activity graph from a causal activity matrix.

Filter Log Alignments: Filter a log alignment array using an accepting Petri net array. All uncovered transition are filtered out.

Merge Accepting Petri Nets: Merge an accepting Petri net array into an accepting Petri net. One supported way to merge nets (the one used in this paper) is by copying all objects (transitions, places, arcs) into a new net, and then fuse all visible transitions with the same label. However, other ways to merge nets are also supported.

Merge Log Alignments: Merge a log alignment array into a log alignment. The result will be a weak alignment. The supported way to merge alignments is to (1) accumulate costs and (2) 'zip' both alignments in such a way that in case of a conflict always the cheapest option is 'zipped in'.

Replay Event Logs: replay an event log array on an accepting Petri net array.

Split Accepting Petri Net: Create an accepting Petri net array from an activity cluster array and an accepting Petri net. The most straightforward supported way to split a net into subnets is by copying the net and removing all parts that do not correspond to the cluster at hand. However, as mentioned, this may result in excessive replay times. An alternative supported way is by copying the net and making all visible transitions that do not correspond to the cluster at hand invisible. This alternative way is used in this paper.

Split Event Log: Create an event log array from an activity cluster array and an event log. The most straightforward supported way to split a log is by starting with an empty log and adding all events that correspond to the cluster at hand. However, this may introduce additional causal dependencies between exit and entry activities of loops. An alternative supported way is by copying the log and renaming all events that do not correspond to the cluster at hand with a special activity, say γ. This paper uses the first way.

Please note that every one of these plug-ins may have its own set of parameters. Some parameters will be mentioned (and given actual values) later on.

3.3 Implementation

The entire framework and all algorithms have been implemented in the `DivideAndConquer` package[1] of ProM 6[2]. For every framework object (like a causal activity matrix), this package does not only implement the object itself, but it also implements an import plug-in, an export plug-in, and at least one visualizer plug-in. Every algorithm is implemented as a regular plug-in, and comes with variants for both the UITopia context (that is, the ProM 6 GUI) and the headless context (which allows the plug-ins to be used from scripts and the like).

4 Decomposed Discovery and Replay using ILP

Fig. 3 shows how the "Discover with ILP using Decomposition" plug-in has been implemented on top of the framework, using a Petri net. In this Petri net, the places corre-

[1] The `DivideAndConquer` package is available through `https://svn.win.tue.nl/trac/prom/browser/Packages/DivideAndConquer`

[2] ProM 6 is available through `http://www.promtools.org/prom6`

spond to the framework objects, whereas the transitions correspond to framework plug-ins. The numbers in the plug-ins indicates the order in which the “Discover with ILP using Decomposition” plug-in invokes the framework plug-ins, where “Merge Clusters” (step 4) is an optional step that is only needed to merge all clusters into a single cluster. This optional step is only used to run the discovery algorithm on the entire event log, as filtering the event log on a single cluster containing all activities would result in the same event log. This way, it is certain that the discovery plug-in used is similar in both the decomposed and the non-decomposed setting.

However, there is one exception, one situation where the decomposed discovery uses a different setting than the regular discovery. The regular ILP-based discovery algorithm depends on the causal dependencies that are detected in an event log. For every casual dependency, it will search for a corresponding place [16]. As a result, if no causal dependency is detected from one activity to another, then the algorithm will not search for a place from the corresponding first transition to the second. In the presence of loops, this may be problematic in the decomposed setting, especially if small clusters are involved. Because of the loop, a transition a' that exits the cluster will typically be followed by a transition a that enters it. If in the cluster a can directly be followed by a', no causal dependency between a and a' will be detected as a can be directly followed by a' and vice versa. For this reason, the decomposed discovery algorithm does not use these causal dependencies to search for places. Instead, it uses a way of searching which is described as *basic* in [16].

Fig. 4 shows the same for the “Replay with ILP using Decomposition” plug-in. Note that this plug-in is more complex than the “Discover with ILP using Decomposition”, which is mainly caused by the two catches on the replay mentioned earlier:

1. It is better to obtain subnets by hiding external (to the cluster at hand) transitions. Steps 6, 7 and 11 are the result of this. Step 6 filters the subnets by hiding external transitions, whereas step 7 filters the subnets by removing external transitions. Step 11 uses the result of step 7 to filter the subalignments, which is required by step 12, as merging the log alignments assumes that all external transitions have been removed from the subalignments.
2. We need to scale up costs before the replay, and scale them down afterwards. Step 8 determines the cost factor, that is, the amount by which so scale up an down, step 9 scales the costs up before replay, and step 10 scales them down again.

Except for the underlying ILP-based discovery algorithm (step 6 in Fig. 3) and the underlying ILP-based replay algorithm (step 9 in Fig. 4), all plug-ins have been implemented in the framework, that is, in the `DivideAndConquer` package. This includes the “Discover with ILP using Decomposition” and “Replay with ILP using Decomposition” plug-ins.

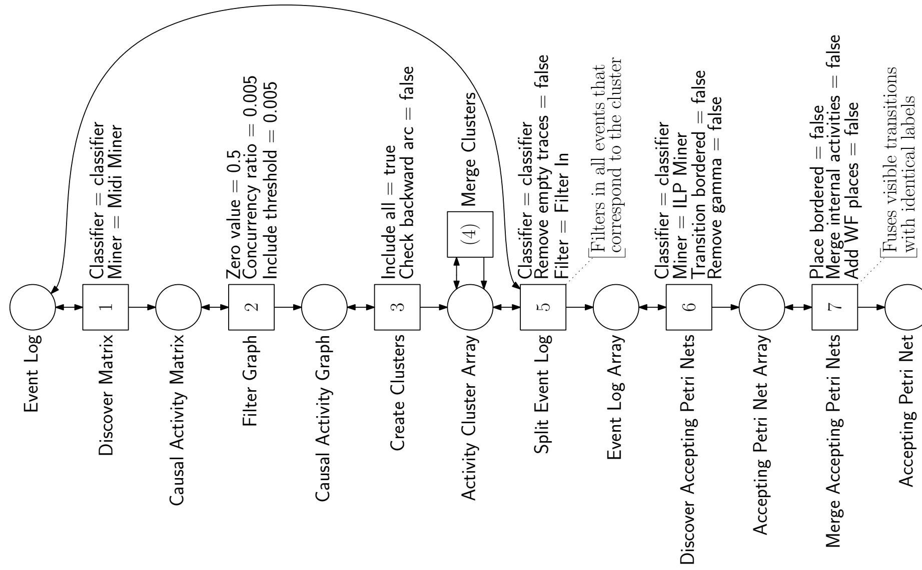

Fig. 3. Discover with ILP using Decomposition.

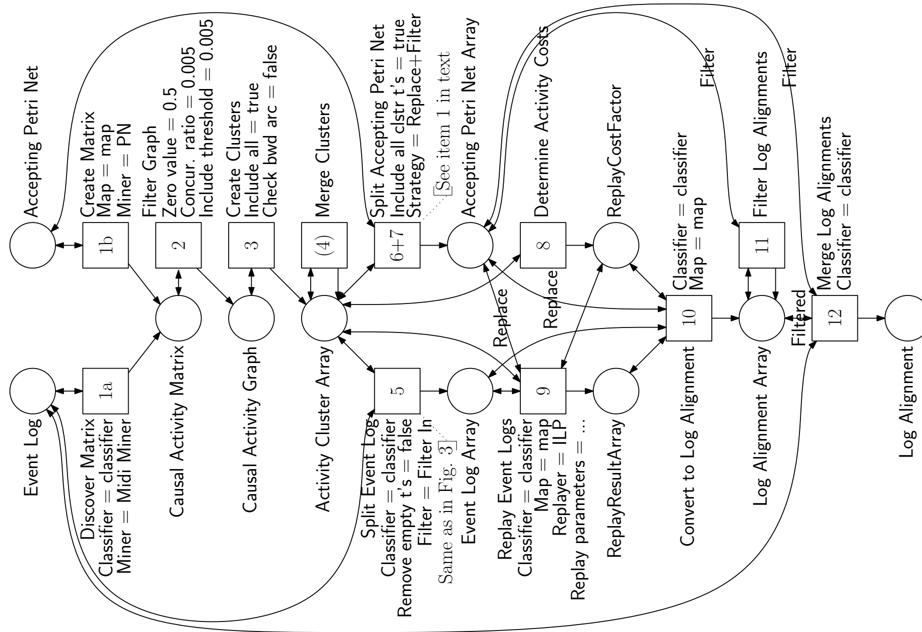

Fig. 4. Replay with ILP using Decomposition.

Table 2. Case study results.

description	event log	place search	nr. of clusters	time (in seconds) average	95% interval		replay costs
regular discovery	aligned	causal	1	2245.72	2223.46	2267.99	N/A
decomposed discovery	aligned	basic	34	97.33	95.92	98.74	N/A
regular replay	aligned	N/A	1	12.04	11.86	12.22	0.00
decomposed replay	aligned	N/A	11	12.83	12.68	12.97	0.00
regular replay	original	N/A	1	92.75	91.57	93.94	14.49
decomposed replay	original	N/A	11	68.37	67.84	68.90	9.39

5 Case Study

For the case study, we use an event log[3] based on the BPI Challenge 2012 [17]. As the ILP discovery algorithm requires an event log that is free of noise, we have aligned the event log to the model that was used in [15]. This may sound artificial, but we feel that for testing the discovery this is justified, as one of the requirements of the original ILP discovery algorithm is that the log is free of noise. For testing the discovery, we will only use the aligned event log, for testing the replay, we will use both event logs. For sake of completeness, we mention that the aligned log contains $13,807$ traces, $383,836$ events, and 58 activities.

The case study was performed on a Dell Optiplex 9020 desktop computer with Inter(R) Core(TM) i7-4770 CPU @ 3.40 GHz processor, 16 GB of RAM, running Windows 7 Enterprise 64-bit, Service Pack 1, and using revision 13851 of the `DivideAndConquer` package.

Table 2 shows the results of the case study. For the decomposed plug-ins in this table, the times reported are the times it takes the entire plug-in (all steps in Fig. 3 or Fig. 4) to finish after the user has provided the required additional information, like the classifier. For the regular plug-ins, the times reported are the times it takes only the discovery (only step 6 in Fig. 3) or only the replay (only step 9 in Fig. 4) to finish.

5.1 Discovery

Regular discovery (without decomposition) using the ILP miner takes about 37.5 minutes, whereas decomposed discovery takes only 1.5 minutes (using 34 clusters). Both result in the same accepting Petri net, which shows that the decomposed discovery clearly outperforms the regular discovery for this case.

As a side note, we mention that we have also ran the regular discovery on the aligned log using the basic place search (cf. Section 4). This run took just under 3 hours (178 minutes), and resulted in a so-called *flower* model [3]. This indicates that this way of searching for places in the ILP discovery algorithm is not suitable for regular discovery, as it takes a long time and yields bad results.

[3] The event logs used for this case study can be downloaded from `https://svn.win.tue.nl/trac/prom/browser/Documentation/DivideAndConquer`

5.2 Replay

Regular replay takes about 12 seconds for the aligned event log, resulting in no costs, and takes 1.5 minutes for the original event log, resulting in 14.49 costs.[4] In contrast, the decomposed replay takes about 12 seconds for the aligned event log, using 11 clusters and resulting in no costs, and takes about 68 seconds for the original event log, using also 11 clusters but resulting in 9.39 costs. Note that decomposed replay by definition provides a lower bound for the alignment costs (due to local optimizations) [14]. How good the lower bound is depends on the decomposition.

So, the decomposed replay is faster on the regular event logs, but may result in only a lower bound of the actual answer (9.39 instead of 14.49 for the original event log).

6 Conclusions

This paper introduced a generic framework for decomposed discovery and replay. The framework is based on objects (matrices, graphs, arrays, etc.) required for the decomposed discovery and replay, and a collection of algorithms to create these objects, either from file or from existing objects. The framework can be used for any decomposed discovery and replay technique.

To illustrate the applicability of the generic framework, we showed how the ILP-based discovery and replay algorithms can be supported by it. The resulting ILP-based discovery is straightforward, but the resulting ILP-based replay algorithm is more complex because of efficiency and implementation issues. Both ILP-based process discovery and conformance checking are supported by our framework.

For the BPI2012 Challenge log, the ILP-based decomposed discovery algorithm has shown to be much faster than the regular discovery algorithm, while resulting in the same model. For the same log, the ILP-based replay algorithm has shown to be faster, but resulting in a less accurate answer. Clearly, there is often a tradeoff between running times and quality.

At the moment, the framework only supports a limited set of discovery and replay algorithms. As an example, only the α-algorithm and the ILP-based algorithm are supported as discovery algorithms. In the near future, we plan to extend this set to include other complex algorithms, like, for example, the evolutionary tree miner and the inductive miner.

Furthermore, the current framework is currently restricted to using a maximal decomposition of a net (or an event log). A coarser decomposition may be faster, as the algorithm may run faster on a single slightly larger cluster instead of on a collection of small clusters. The approach supports any valid decomposition; therefore, we are looking for better techniques to select a suitable set of clusters. Experiments show that it is possible to create clusters that provide a good trade-off between running times and quality.

[4] Please note that, by changing the cost structure as suggested in [14], we can accumulate costs when merging subalignments into a single alignment. However, we do not have a way yet to accumulate fitness when merging subalignments. For this reason, we restrict ourselves to costs here.

References

1. Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C., Byers, A.H.: Big Data: The Next Frontier for Innovation, Competition, and Productivity. Technical report, McKinsey Global Institute (June 2011)
2. Hilbert, M., López, P.: The World's Technological Capacity to Store, Communicate, and Compute Information. Science **332**(6025) (April 2011) 60–65
3. Aalst, W.M.P.v.d.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. 1st edn. Springer Publishing Company, Incorporated (2011)
4. Aalst, W.M.P.v.d., Rubin, V., Verbeek, H.M.W., Dongen, B.F.v., Kindler, E., Günther, C.W.: Process mining: a two-step approach to balance between underfitting and overfitting. Software and Systems Modeling **9**(1) (2010) 87–111
5. Bergenthum, R., Desel, J., 0001, R.L., Mauser, S.: Process Mining Based on Regions of Languages. In Alonso, G., Dadam, P., Rosemann, M., eds.: BPM. Volume 4714 of Lecture Notes in Computer Science., Springer (2007) 375–383
6. Solé, M., Carmona, J.: Process Mining from a Basis of State Regions. In: Proceedings of the 31st International Conference on Applications and Theory of Petri Nets. PETRI NETS'10, Berlin, Heidelberg, Springer-Verlag (2010) 226–245
7. Carmona, J., Cortadella, J., Kishinevsky, M.: A Region-Based Algorithm for Discovering Petri Nets from Event Logs. In: Business Process Management (BPM2008). (2008) 358–373
8. Werf, J.M.E.M.v.d., Dongen, B.F.v., Hurkens, C.A.J., Serebrenik, A.: Process Discovery using Integer Linear Programming. Fundam. Inform. **94**(3-4) (2009) 387–412
9. Aalst, W.M.P.v.d., Weijters, A.J.M.M., Maruster, L.: Workflow Mining: Discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering **16** (2003) 2004
10. Adriansyah, A., Dongen, B.F.v., Aalst, W.M.P.v.d.: Conformance Checking using Cost-Based Fitness Analysis. In Chi, C., Johnson, P., eds.: IEEE International Enterprise Computing Conference (EDOC 2011), IEEE Computer Society (2011) 55–64
11. Muñoz-Gama, J., Carmona, J.: A Fresh Look at Precision in Process Conformance. In Hull, R., Mendling, J., Tai, S., eds.: Business Process Management (BPM 2010). Volume 6336 of Lecture Notes in Computer Science., Springer-Verlag, Berlin (2010) 211–226
12. Muñoz-Gama, J., Carmona, J.: Enhancing precision in Process Conformance: Stability, confidence and severity. In: CIDM, IEEE (2011) 184–191
13. Rozinat, A., Aalst, W.M.P.v.d.: Conformance checking of processes based on monitoring real behavior. Inf. Syst. **33**(1) (March 2008) 64–95
14. Aalst, W.M.P.v.d.: Decomposing Petri nets for process mining: A generic approach. Distributed and Parallel Databases **31**(4) (2013) 471–507
15. Verbeek, H.M.W., Aalst, W.M.P.v.d.: Decomposing Replay Problems: A Case Study. In Moldt, D., ed.: PNSE+ModPE. Volume 989 of CEUR Workshop Proceedings., CEUR-WS.org (2013) 219–235
16. Wiel, T.v.d.: Process mining using integer linear programming. Master's thesis, Eindhoven University of Technology, Department of Mathematics and Computer Science (2010) http://alexandria.tue.nl/extra1/afstversl/wsk-i/wiel2010.pdf.
17. Dongen, B.F.v.: BPI Challenge 2012 (2012) http://dx.doi.org/10.4121/uuid: 3926db30-f712-4394-aebc-75976070e91f.