Using Life Cycle Information in Process Discovery

Sander J.J. Leemans, Dirk Fahland, and Wil M.P. van der Aalst

Eindhoven University of Technology, the Netherlands
{s.j.j.leemans, d.fahland, w.m.p.v.d.aalst}@tue.nl

Abstract Understanding the performance of business processes is an important part of any business process intelligence project. From historical information recorded in event logs, performance can be measured and visualized on a discovered process model. Thereby the accuracy of the measured performance, e.g., waiting time, greatly depends on (1) the availability of start and completion events for activities in the event log, i.e. transactional information, and (2) the ability to differentiate between subtle control flow aspects, e.g. concurrent and interleaved execution. Current process discovery algorithms either do not use activity life cycle information in a systematic way or cannot distinguish subtle control-flow aspects, leading to less accurate performance measurements. In this paper, we investigate the automatic discovery of process models from event logs, such that performance can be measured more accurately. We discuss ways of systematically treating life cycle information in process discovery and their implications. We introduce a process discovery technique that is able to handle life cycle data and that distinguishes concurrency and interleaving. Finally, we show that it can discover models and reliable performance information from event logs only.

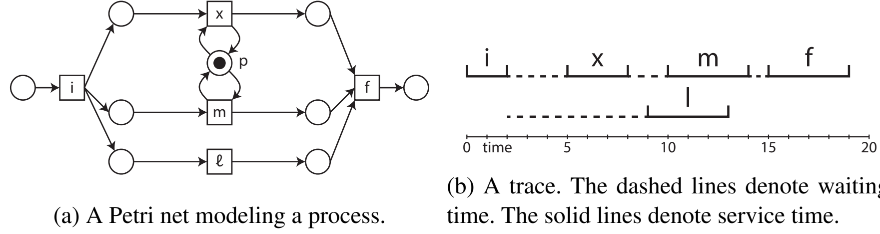
Keywords: process mining, process discovery, performance measurement, rediscoverability, concurrency

1 Introduction

One of the central tasks in business process intelligence is to understand the actual performance of a process and the impact of resource behaviour and process elements on overall performance. Event data logged by Business Process Management (BPM) systems or Enterprise Resource Planning (ERP) systems typically contains time stamped *transactional events* (start, completion, etc.) for each activity execution. Process mining allows to analyse this transactional data for performance. Typically, first a model of the process is discovered, which is then annotated with performance information.

Performance information might consist of several measures, for example service time (the time a resource is busy with a task), waiting time (the time between an activity becoming enabled and a resource starting to execute it), sojourn time (the sum of both) and synchronisation time (for concurrent activities, the time between completion of the first and completion of the last).

Figure 1a shows an example process in some imaginary hospital: after an *initial examination* (i), tissue samples are investigated in a *laboratory* (l). Meanwhile, the patient undergoes two tests: an *x-ray* (x) and an *mri* (m) test. When all tests are completed, the



(a) A Petri net modeling a process.

(b) A trace. The dashed lines denote waiting time. The solid lines denote service time.

Figure 1: A process with concurrency and interleaving and a trace.

patient meets the doctor for a *final* time (f). Figure 1b shows a patient (a *trace*) of this process where each activity is logged as an interval of a *start* (s) event and a *complete* (c) event; the dashed lines denote waiting time. The patient cannot perform the x and m tests at the same time, i.e. they are *interleaved* (due to place p), thus the waiting time before m starts after the completion of x. In contrast, the lab test l can be executed independently of the x and m tests, i.e. they are *concurrent*, so the waiting time before l starts at the completion of i (waiting time starts at the last time i became enabled). Without knowledge of the model, the waiting time of l cannot be positioned properly and thus waiting, sojourn and synchronisation times will be measured incorrectly: The waiting time is $9 - 2 = 7$ time units rather than $9 - 8 = 1$ unit. Therefore, in order to reliably compute performance measures, a process model is required to provide information on concurrency and interleaving.

The difficulty in discovering concurrency lies in the event logs: Most process discovery algorithms [14,19,16,2,15] assume that the event log contains events representing atomic executions of activities. On atomic task executions, concurrency and interleaving cannot be distinguished, and more information is required. In this paper, we assume that the event log contains examples of non-atomic executions of activities, i.e. for each activity instance the start and the completion time is known. The XES standard [8] is often used as an input format for event logs and supports this with the *lifecycle:transition* extension. Several process discovery techniques exist that take transactional data into account, such as Tsinghua-α (Tα) [20], Process Miner (PM) [17]. and several other approaches [4,9]. Unfortunately, none of these distinguishes concurrency and interleaving, and most [20,4,9] do not guarantee to return sound models, i.e. without deadlocks or other anomalies; both of which are prerequisites for reliable computation of performance measures.

In the remainder of this paper, we first study the impact of transactional data on event logs and models. Second, we elaborate on the problem of incomplete/inconsistent transactional data in event logs and give a way to repair such event logs (Section 3). and we introduce an abstraction (collapsed process models) that enables reasoning about such data (Section 4). Third, we introduce a new process discovery algorithm based on the Inductive Miner (IM) framework [12] that uses this information to discover collapsed process models (Section 5). The new algorithm faces two challenges: first, the transactional data must be handled correctly; and second, it should distinguish concurrency from interleaving. In Section 6, we illustrate its functioning, study of the implications of this abstraction on any process discovery algorithm and on existing model quality measures, and discuss related work; Section 7 concludes the paper.

2 Transactional Information in Event Logs

A *trace* is a sequence of *events*, denoting for a case, e.g. a customer, what process steps (*activities*) were executed for that case. Events may carry additional attributes, such as timestamps and a *transaction type*. The latter indicates whether the activity started, completed, etc. An *activity instance* is the execution of an activity in a trace, and may consist of a start event and a completion event, as well as events of other transaction types. For instance, $t = \langle a_s^{11:50}, a_c^{11:53}, b_s^{12:03}, b_c^{12:50} \rangle$ denotes a trace of 4 events: first, an instance of activity a was started, second, an instance of a completed, after which an instance of activity b started and an instance of activity b completed. The timestamps in superscript denote the times at which the events occurred; we will omit timestamps if they are not relevant. An *event log* is a multiset of traces.

In trace t, it makes sense to assume that a_s and a_c are events of the same activity instance. However, this information is usually not recorded in the event log, and the techniques introduced in this paper neither need nor try to infer this information. In the following sections, we assume presence of at least start and completion events; the techniques describe in this paper will ignore other transaction types.

Consider the trace $t = \langle a_s \rangle$. As an activity instance of a was started but never completed, there is either an a_c event missing, or the a_s event should not have been recorded. Similar problems could have occurred when unmatched completion events appear. This raises the notion of a *consistent trace*, similar to [4]:

Definition 1. *A trace is* consistent *if and only if each start event has a corresponding completion event and vice versa.*

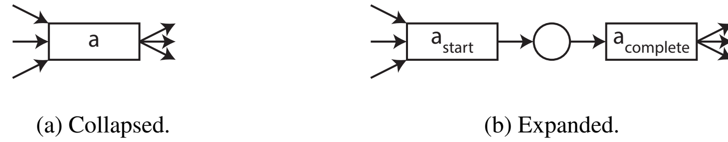
3 Preparing the Input

In real-life data sets, it is possible that some traces in an event log do not adhere to Definition 1. A trace can be checked for consistency easily with a single pass over the trace and some bookkeeping. Nevertheless, our approach requires consistent traces, so any inconsistency need to be dealt with.

We illustrate the decisions that have to be made using an example trace $t = \langle a_s^{11:30}$, $a_s^{12:40}$, $a_c^{13:50} \rangle$. Clearly, this trace is not consistent, as there are two start events of activity a and only one complete event. There are several ways to make t consistent:

- $\langle \rangle$
- $\langle a_s^{11:30} a_c^{13:50} \rangle$
- $\langle a_s^{12:40}, a_c^{13:50} \rangle$
- $\langle a_s^{11:30}, a_c, a_s^{12:40}, a_c^{13:50} \rangle$
- $\langle a_s^{11:30}, a_s^{12:40}, a_c, a_c^{13:50} \rangle$
- $\langle a_s^{11:30}, a_s^{12:40}, a_c^{13:50}, a_c \rangle$

Without further information, we cannot decide on the trace that matches reality in the best way possible. Additional information in the event log could be used, such as the *concept:instance* extension of the XES standard [8], which links start and complete events of activity instances. If this extension would indicate that events $a_s^{12:40}$ and $a_c^{12:40}$



(a) Collapsed. (b) Expanded.

Figure 2: Excerpt of a collapsed and its expanded workflow net.

form an activity instance, it makes sense to opt for $\langle a_s^{12:40}, a_c^{13:50}\rangle$ or $\langle a_s^{11:30}, a_c, a_s^{12:40}, a_c^{13:50}\rangle$. For our experiments, we choose option $\langle a_s^{11:30}, a_c, a_s^{12:40}, a_c^{13:50}\rangle$, i.e. each completion event is matched with the last occurring start event, and completion events are inserted right after unmatched start events. Completion events are handled symmetrically. Please note that by considering unmatched completion events as atomic, we can handle logs that contain only completion events using the same approach.

The pre-processing step ignores all events that contain other life cycle annotations than start or completion (see e.g. the *lifecycle:transition* extension of the XES 2.0 standard [8]). In the remainder of this paper, we only consider event logs consisting of consistent traces.

4 Transactional Information in Process Models

In standard Petri nets, a transition is considered atomic, i.e. when firing a transition its start is indistinguishable from its completion. This poses a problem for performance measurements if we map activities onto transitions, as a transition cannot take time.

A solution could be to use separate transitions for the start and completion of activities, such that their execution can be distinguished. This poses a new challenge to process models: if the start and completion transitions do not match, the model might allow for inconsistent traces, such as two starts followed by one completion. This may severely jeopardise the accuracy of performance measurements. Moreover, in order to measure performance, it must be known which combinations of start and completion transitions correspond to an activity.

A common solution to these issues is to expand each transition (a, Figure 2a) in two transitions, connected with a place: one transition denotes the start (a_s), the other transition denotes the completion (a_c) of the transition (Figure 2b) [3]. The connection between the two newly created transitions is kept, and therefore the model does not allow inconsistent traces. We refer to this existing technique as *expanding* a process model; the un-expanded process model is a *collapsed* model.

Process Trees. Process trees are abstract representations of block-structured workflow nets [2]. Process trees are particularly suitable for process discovery, as they are by definition free of deadlocks and other anomalies (*sound* [1]). A process tree describes a language, and it consists of a hierarchy of operators being the nodes of the tree, and activities being the leaves of the tree. An activity describes the singleton language of that activity, while an operator describes how the languages of its children are to be combined. In [12], the four operators sequence ($\rightarrow$), exclusive choice ($\times$), concurrency ($\wedge$) and loop ($\circlearrowleft$) are considered. For example, the process tree $\rightarrow(a, \times(\wedge(b, c), \circlearrowleft(d, e)), f)$ has, among other things, the following traces $\langle a, b, c, f\rangle$,

$\langle a, c, b, f \rangle, \langle a, d, f \rangle, \langle a, d, e, d, e, d, f \rangle$. We use $\mathcal{L}(T)$ to denote the language of process tree T.

Collapsed Process Trees. Process trees can be mapped onto Petri nets using the translation presented in [12]. Hence, they face the problem of atomicity as well. Therefore, we lift the expanding technique to process trees by introducing a variant that keeps the link between starts and completes: *collapsed process trees*. A collapsed process tree can be expanded into a normal process tree, e.g $\times(a, b)$ expands to $\times(\rightarrow(a_s, a_c), \rightarrow(b_s, b_c))$.

Definition 2. *A* collapsed process tree *is a process tree in which each activity -a- denotes the process tree* $\rightarrow(a_s, a_c)$.

5 Inductive Miner - life cycle

In this section, we introduce an algorithm (Inductive Miner - life cycle (IMLC)) that is able to handle life cycle data and distinguishes concurrency and interleaving. In this section, we first recall principles of recursive process discovery with IM. Second, we describe how transactional data is dealt with in this framework (Section 5), and introduce a way to distinguish interleaving from concurrency (Section 5). Finally, we give an example (Section 5) and we describe the implementation (Section 5).

Inductive Miner

The divide-and-conquer framework Inductive Miner [12] (IM) recursively applies four steps. (1) Select a *cut*: a division of activities in the event log and a process tree operator, e.g. $(\wedge, \{a, b\}, \{c\})$. (2) Split the log into smaller sub-logs according to this cut. (3) Recurse on each sub-log. The recursion ends when a base case, e.g. a log containing only a single activity, is left. (4) If no cut can be selected, a *fall through* is returned. We will make use of this conceptual framework. However, in its current form, IM simply treats each event as a separate activity instance.

In order to select a cut, IM considers the directly-follows graph, which contains which activities were directly followed by other activities. Each of the process tree operators leaves a specific footprint in the directly-follows graph. Thus, IM detects cuts by identifying these footprints. Figure 3 shows the footprints of the process tree operators (ignoring the dashed box); [12] provides details and log splitting procedures.

IMLC uses the IM framework and needs to adapt all four steps to cope with transactional events and to detect interleaving. We first introduce how IMLC handles transactional events, after which we introduce how it detects interleaving.

Transactional Events

In order to handle transactional events, changes are required in three of the four steps of IM; log splitting requires no changes. As base case detection involves little changes to cope with transactional events, it will not be discussed in this paper.

Cut Detection. Most parts of cut detection remain roughly as in previous works [12], however cut detection in IM relies on directly-follows graphs, which are constructed differently in case of transactional events. The method presented here is based on ideas used in e.g. Tα and PM, but differs in details. (Collapsed) activity a follows (collapsed) activity b directly, if in the event log an (expanded) event of a follows an (expanded)

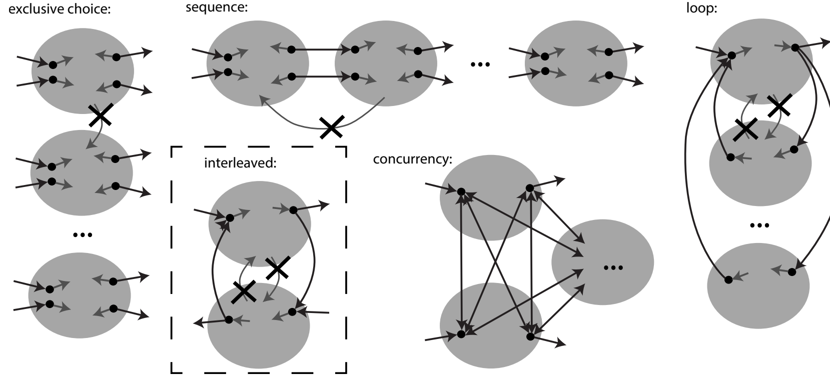
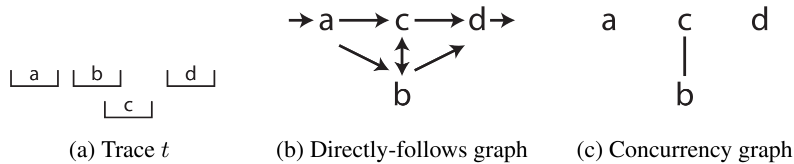


Figure 3: Cut footprints in the directly-follows graph, see [12].

event of b without both a start and completion event of the same activity between them. For instance, consider trace $t = \langle a_s, a_c, b_s, c_s, b_c, c_c, d_s, d_c \rangle$; visualised in Figure 4a. In t, consider the event c_s. Obviously, c_s follows b_s directly, so in Figure 4b, c directly follows b. Moreover, c_s follows event a_c directly, as there is only a completion event of b in between them, which is not a full activity instance. In contrast, between a_c and d_s there are two full activity instances (b and c), thus a is not directly followed by d in t.

Any activity instance of which the start event is not preceded by the completion event of another activity instance is called a *start activity*. In t, a is the only start activity; b, c and d are not, as there occurs an a_c event before them. Similarly, d is the only end activity of t. In Figure 4b, these start and end activities are denoted by incoming and outgoing edges.

Fall Throughs If no cut can be found, a fall through is to be selected. IMLC contains several fall throughs, of which the last one is a model that expresses all behaviour in the event log, and therefore guarantees fitness. For non-collapsed process trees, the flower model $\circlearrowleft(\tau, a, b, c, \ldots)$ serves this purpose, as it allows for any behaviour of the included activities $a\ b\ c\ \ldots$. For collapsed process trees, a model allowing for all behaviour may not exist, as in a collapsed process tree, no activity can be concurrent with itself (see Section 6). Hence, IMLC counts the maximum number of times an activity is concurrent with itself in the event log, and constructs a model accordingly. For instance, in the event log $\{\langle a_s, a_s, a_c, b_s, a_c, b_c \rangle\}$, at most 2 a's and 1 b are concurrent with themselves. Then, the fall through collapsed model that IMLC chooses is $\wedge(\circlearrowleft(\tau, a), \circlearrowleft(\tau, a), \circlearrowleft(\tau, b))$. This model can produce any behaviour of two a's and one b all concurrent to each other.



(a) Trace t (b) Directly-follows graph (c) Concurrency graph

Figure 4: Trace t and its corresponding graphs.

Interleaving

Besides handling transactional data, IMLC is able to detect interleaving. We first introduce the corresponding process tree operator, then describe how to detect it. The interleaving operator $\leftrightarrow$ takes any number of subtrees ($\geqslant 2$) and combines their languages.

Definition 3. *Let $T_1 \dots T_n$ be process trees. Let $p(n)$ be the set of all permutations of the numbers $\{1 \dots n\}$. Then $\mathcal{L}(\leftrightarrow(T_1, \dots, T_n)) = \bigcup_{(i_1 \dots i_n) \in p(n)} \mathcal{L}(\rightarrow(T_{i1} \dots T_{in}))$.*

Note that the union can also be expressed as an exclusive choice over each sequence $\rightarrow(T_{i1} \dots T_{in})$.

Detection Strategy. This structure is exploited in the detection of $\leftrightarrow$, and IMLC applies a three-stage strategy: (1) An interleaving cut is detected using the footprint of $\leftrightarrow$ (see Figure 3). However, detection of this footprint is insufficient to conclude interleaving, as e.g. the footprint does not guarantee that each child is executed at most once. Therefore, we denote the detection with an additional operator (*maybe-interleaved* ($\Leftrightarrow$)), e.g. the cut $(\leftrightarrow, \{a\}, \{b\})$ is detected but $(\Leftrightarrow, \{a\}, \{b\})$ is reported. (2) Using the $\Leftrightarrow$ cut, the event log is split and recursion continues as usual. (3) After recursion, interleaving is derived from the structure of the process tree, e.g. each occurrence of $\Leftrightarrow(\rightarrow(T_1, T_2), \rightarrow(T_2, T_1))$ is replaced by $\leftrightarrow(T_1, T_2)$.

(1) Detection of $\leftrightarrow$. To detect interleaving, IMLC uses the footprint of the $\leftrightarrow$ operator in the directly-follows graph. This footprint is shown in Figure 3: from each end activity, an edge must be present to all start activities of all other circles, and vice versa. Other directly-follows edges between circles are not allowed.

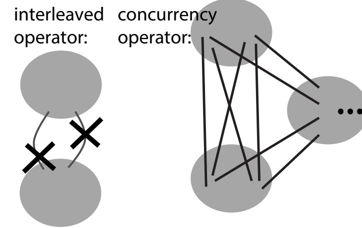


Figure 5: Cut footprints in the concurrency graph.

Notice that if the start and end activities overlap, this footprint might overlap with the concurrency footprint, e.g. $\leftrightarrow(a, b)$ has the same footprint as $\wedge(a, b)$. Therefore, IMLC also considers the *concurrency graph*. As shown in [20], transactional data allows for direct detection of concurrency: whenever two activity instances overlap in time (such as b and c in Figure 4a), their activities are concurrent. Figure 4c shows the concurrency graph of our example. For this step, it suffices to keep track of the number of started-but-not-yet-completed activity instances. Interleaving and concurrency have clearly different footprints in the concurrency graph (see Figure 5).

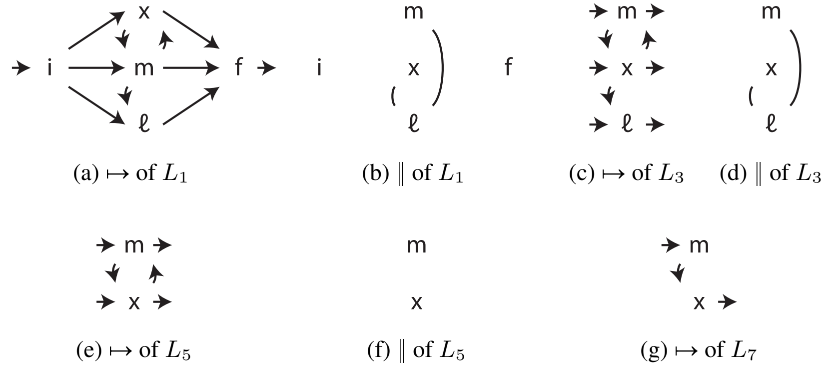
(2) Log Splitting for $\Leftrightarrow$. For $\Leftrightarrow$, the log is split by dividing traces based on the activities with which they start. For instance, consider the event log $L = \{\langle x_s, x_c, m_s, m_c \rangle, \langle m_s, m_c, x_s, x_c \rangle\}$ and the cut $(\Leftrightarrow, \{x\}, \{m\})$. Based on this cut, L is split into the sub-logs $\{\langle x_s, x_c, m_s, m_c \rangle\}$ and $\{\langle m_s, m_c, x_s, x_c \rangle\}$.

(3) From $\Leftrightarrow$ to $\leftrightarrow$. Intuitively, log splitting 'unravels' the interleaved execution: the sub-trees of $\{x\}$ and $\{m\}$ can be executed in any order; for each such ordering, a different sub-log is returned. After unraveling the order, recursion continues and, by the interleaving semantics, we expect $\rightarrow$ operators to appear in both branches. Therefore, after recursion, each occurrence of the pattern $\Leftrightarrow(\rightarrow(T_1, T_2), \rightarrow(T_2, T_1))$ (or an n-ary generalisation thereof) is replaced with $\leftrightarrow(T_1, T_2)$. In case further recursion shows that

interleaving was not present, i.e. the pattern does not show up, any remaining $\overset{?}{\leftrightarrow}$ operator is replaced with an $\times$ operator; the subtrees remain untouched. Effectively, this allows a single activity label to occur in multiple branches of the model.

Example

We illustrate IMLC using an example, derived from Figure 1a. Consider log $L_1 = \{\langle i_s, i_c, m_s, m_c, x_s, l_s, x_c, l_c, f_s, f_c\rangle, \langle i_s, i_c, l_s, x_s, x_c, m_s, l_c, m_c, f_s, f_c\rangle\}$. The directly-follows and concurrency graphs of L_1 are shown in figures 6a and 6b. Cut detection will find the cut $(\rightarrow, \{i\}, \{x, m, l\}, \{f\})$, after which L_1 is split into $L_2 = \{\langle i_s, i_c\rangle\}$, $L_3 = \{\langle m_s, m_c, x_s, l_s, x_c, l_c\rangle, \langle l_s, x_s, x_c, m_s, l_c, m_c\rangle\}$ and $L_4 = \{\langle f_s, f_c\rangle\}$. The partial result up till this point is $\rightarrow(\text{IMLC}(L_2), \text{IMLC}(L_3), \text{IMLC}(L_4))$. Next, IMLC recurses on the first and last branch, both of which result in a base case, after which the partial result becomes $\rightarrow(i, \text{IMLC}(L_3), f)$. Next, IMLC recurses on L_3; figures 6c and 6d show the corresponding graphs. Based on the concurrency graph, IMLC selects the cut $(\wedge, \{m, x\}, \{l\})$. Using this cut, L_3 is split into $L_5 = \{\langle m_s, m_c, x_s, x_c\rangle, \langle x_s, x_c, m_s, m_c\rangle\}$ and $L_6 = \{\langle l_s, l_c\rangle, \langle l_s, l_c\rangle\}$. A recursion on the base case L_6 yields the partial result $\rightarrow(i, \wedge(\text{IMLC}(L_5), l), f)$. Next, IMLC recurses on L_5; figures 6e and 6f show its graphs. As the directly-follows graph shows interconnected activities m and x that are not concurrent according to the concurrency graph, IMLC selects $(\overset{?}{\leftrightarrow}, \{m\}, \{x\})$. Log L_5 is split into $L_7 = \{\langle m_s, m_c, x_s, x_c\rangle\}$ and $L_8 = \{x_s, x_c, m_s, m_c\}$. The partial result becomes $\rightarrow(i, \wedge(\overset{?}{\leftrightarrow}(\text{IMLC}(L_7), \text{IMLC}(L_8)), l), f)$. As IMLC recurses on L_7, using the directly-follows graph of Figure 6g, the cut $(\rightarrow, \{m\}, \{x\})$ is selected, and by log splitting, two recursions and two base cases, the intermediate result becomes $\rightarrow(i, \wedge(\overset{?}{\leftrightarrow}(\rightarrow(m, x), \text{IMLC}(L_8)), l), f)$. A similar recursion on L_8 yields the result $\rightarrow(i, \wedge(\overset{?}{\leftrightarrow}(\rightarrow(m, x), \rightarrow(x, m)), l), f)$. Finally, the post-processing step transforms this result into $\rightarrow(i, \wedge(\leftrightarrow(m, x), l), f)$, which corresponds to Figure 1a.



(a) $\mapsto$ of L_1 (b) $\|$ of L_1 (c) $\mapsto$ of L_3 (d) $\|$ of L_3

(e) $\mapsto$ of L_5 (f) $\|$ of L_5 (g) $\mapsto$ of L_7

Figure 6: The directly-follows $\mapsto$ and concurrency $\|$ graphs for a few (sub-)logs.

Implementation

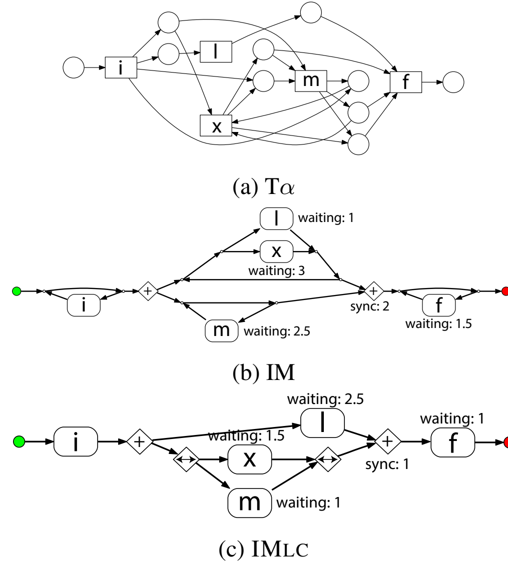
IMLC and the pre-processing step described in Section 3 are available as plug-ins of the ProM framework [7]. To guarantee compatibility with existing plug-ins, IMLC returns models being collapsed trees, i.e. the leaves are activities (for instance $\wedge(a, b)$). A separate plug-in ("expand collapsed process tree") is available to expand these trees according to Definition 2.

6 Discussion

In this section, we first study some guarantees offered by IMLC and illustrate some results using real-life data. Second, we discuss the theoretical limits of any process discovery algorithm that uses the abstraction of Section 4.

Guarantees
As IMLC uses process trees, any model returned by it is guaranteed to be sound. Furthermore, by the collapsed/expanding concept, all traces that such a model can produce are consistent.

Fitness and termination are guaranteed by the Inductive Miner framework for consistent traces: Theorem 3 of [12] holds because Definition 1 of [12] holds: case distinction on the patterns of Figure 3 ensures that we add an $\leftrightarrow$ node in the model as a shorthand for the $\overset{?}{\leftrightarrow}(\rightarrow(\ldots), \rightarrow(\ldots))$ construct (which enumerates all interleaved sequences) only when partitioning the log into several sublogs based on the start activity (preserves fitness to each interleaving); each sublog is strictly smaller (termination).



(a) Tα

(b) IM

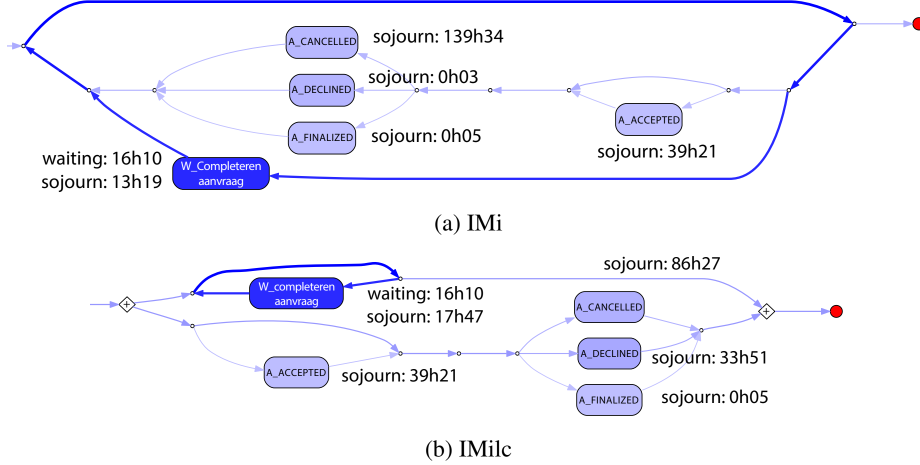
(c) IMLC

Figure 7: Results on PN of Figure 1a.

Rediscoverability, i.e. whether IMLC is able to rediscover the language of a system underlying the event log, is still guaranteed for systems consisting of $\times$, $\rightarrow$, $\wedge$ and $\circlearrowleft$. Under which assumptions rediscoverability holds for $\leftrightarrow$ requires further research.

Illustrative Results
We applied IMLC, IM, and Tα to the event log of Figure 1a and Section 5, enriched with time stamps: $L_1 = \{\langle i_s^1, i_c^2, m_s^3, m_c^4, x_s^5, l_s^6, x_c^7, l_c^8, f_s^9, f_c^{10}\rangle, \langle i_s^2, i_c^3, l_s^4, x_s^5, x_c^6, m_s^7, l_c^8, m_c^9, f_s^{10}, f_c^{11}\rangle\}$; Figure 7 shows the results. IM misses the concurrency relation between m and l and does restrict the number of times each activity can be executed. The model produced by Tα can only fire i (it is not sound), so no performance measure can be computed.

For the two other models, we measured waiting and synchronisation time by first removing the deviations using an alignment [3], after which we obtained average synchronisation and waiting times by considering the last completed non-concurrent activity instance, as described in [21]. Even on such a small log, the measured waiting and synchronisation times differ wildly, illustrating the need for reliable performance measures. In case x and m are indeed interleaved instead of concurrent, we argue that the measured times on the model returned by IMLC are correct.



(a) IMi

(b) IMilc

Figure 8: Excerpts of models obtained from BPIC12.

As a second experiment, we created a secondary algorithm that applies infrequent-behaviour filtering, similar to IMi [11]. This algorithm, *Inductive Miner - infrequent & life cycle* (IMilc) was applied to the BPI challenge log of 2012 [6], filtered to contain a subset of activities (starting with A and W), and only the cases denoted as 'successful' in the log. This event log describes a mortgage application process in a Dutch financial institution. Figure 8 shows an exerpt of the results obtained by IMilc and IMi, enriched with average waiting and sojourn times. The model obtained by applying tα is unsound and therefore, performance could not be computed.. Waiting time can be computed deterministically only for *W_completeren aanvraag*, as that is the only activity in the event log having start events. In the model by IMi, *W_Completeren aanvraag* has a waiting time of 16 and a sojourn time of 13. This is inconsistent, as sojourn time = waiting time + service time. Manual inspection reveals that this activity overlaps with the A activities in this excerpt, which is correctly captured by concurrency. IMi (Figure 8a) did not detect the concurrency, and therefore some sojourn times are measured with respect to completion events of different activities, making the results unreliable.

Limitations of Collapsed Models

The idea of collapsed tasks implies some representational bias on several process formalisms; we identified three main restrictions. First, as the start and complete transitions acts as a transactional envelope for an activity, no restrictions can be posed on start and completion transitions themselves. For instance, take the partial workflow net shown in Figure 9, in which $approve_s$ must happen before *send* starts. This restriction is not expressible in collapsed process models, as it inherently involves targeting the 'hidden' start in a collapsed activity, regardless of the formalism used.

Second, unbounded concurrency cannot be expressed in most formalisms. Consider the infinite set of traces $\mathcal{L} = \{\langle a_s, a_s, \ldots a_c, a_c \rangle\}$, i.e. a can be parallel with itself arbitrarily often. The YAWL [10] language supports unbounded concurrency by means of 'multiple-instance activi-

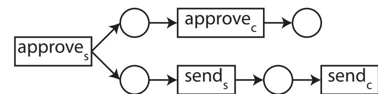


Figure 9: A partial workflow net in which $approve_s$ happens before $send$.

ties'. However, correctly handling multi-instance activities requires an emptiness test [13] which is expressible in neither process trees nor regular Petri nets. This restriction also implies that a flower model that produces only consistent traces cannot exist.

Third, the language of any collapsed process model can obviously only contain consistent traces (Definition 2). Even though, as shown in Section 3, inconsistent traces show inherent ambiguity, input traces might be inconsistent and therefore, traditional perfect fitness might be unachievable. e.g. there is no collapsed process model to represent $\langle a_s, a_s \rangle$. We argue that fitness measures should be robust against such ambiguities, but adapting measures is outside the scope of this paper.

Related Work
Several process discovery techniques take transactional data into account, e.g. [20,17,4,9]: transactional data is used to aid in directly-follows relation construction. For instance, transactional data enables explicit concurrency detection in low information settings [20]. IMLC uses a similar idea, but slightly differs in details, e.g. in IMLC, two activity instances can be both directly-following as well as concurrent. However, none of the other approaches distinguishes concurrency and interleaving, and most [4,20,9] do not guarantee to return sound models. Unsound models, as shown in Section 6, cannot be used to measure performance in some cases.

Of the mentioned approaches, only PM [17] guarantees to return sound models: it constructs structured models similar to process trees based on the directly-follows relation over transactional data. However, the particular approach does not generalise the behaviour of the log [17], is not robust to noise [5], and does not distinguish concurrency and interleaving.

7 Conclusion

We investigated an aspect of measuring business process performance by discovering process models with performance information from event logs with transactional data, i.e., start and complete events that are recorded for each activity instance. We have shown that performance information depends on whether activities have been executed truly concurrently or interleaved. All existing process discovery algorithms assume no difference between concurrency and interleaving and thus may yield inaccurate performance results.

We presented a first process discovery technique that can distinguish concurrency and interleaving in the presence of transactional data, i.e. start and completion events, using the Inductive Miner [12] framework. The algorithm guarantees soundness and fitness; a first evaluation showed that it can return more accurate performance information than the state of the art.

An open question remaining is under which assumptions rediscoverability holds for IMLC, and how discovery can benefit from other life cycle transitions, e.g. assign, reassign, schedule, suspend etc. For instance, an *enqueue* event [18] might reveal when queueing commenced and hence provide even more information about dependencies with other activities. Another point of future research is how expanding and collapsing

influences the existing model/log evaluation criteria fitness, precision and generalisation.

References

1. van der Aalst, W.M.P., van Hee, K.M., ter Hofstede, A.H.M., Sidorova, N., Verbeek, H.M.W., Voorhoeve, M., Wynn, M.T.: Soundness of workflow nets: classification, decidability, and analysis. Formal Asp. Comput. 23(3), 333–363 (2011)
2. van der Aalst, W., Weijters, A., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE Trans. Knowl. Data Eng. 16(9), 1128–1142 (2004)
3. Adriansyah, A.: Aligning Observed and Modeled Behavior. Ph.D. thesis, Eindhoven University of Technology (2014)
4. Burattin, A., Sperduti, A.: Heuristics miner for time intervals. ESANN (2010)
5. De Weerdt, J., De Backer, M., Vanthienen, J., Baesens, B.: A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs. Information Systems 37, 654–676 (2012)
6. van Dongen, B.: BPI Challenge 2012 Dataset (2012), `http://dx.doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f`
7. van Dongen, B., de Medeiros, A., Verbeek, H., Weijters, A., van der Aalst, W.: The ProM Framework: A new era in process mining tool support. ICATPN 3536, 444–454 (2005)
8. Günther, C., Verbeek, H.: XES v2.0 (2014), `http://www.xes-standard.org/`
9. Günther, C., Rozinat, A.: Disco: Discover your processes. CEUR Workshop Proceedings, vol. 940, pp. 40–44. CEUR-WS.org (2012)
10. ter Hofstede, A.H.M., van der Aalst, W.M.P., Adams, M., Russell, N.: Modern Business Process Automation - YAWL and its Support Environment. Springer (2010)
11. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs containing infrequent behaviour. LNBIP, vol. 171, pp. 66–78. Springer (2013)
12. Leemans, S., Fahland, D., van der Aalst, W.: Discovering block-structured process models from event logs - a constructive approach. LNCS, vol. 7927, pp. 311–329. Springer (2013)
13. Linz, P.: An introduction to formal languages and automata. Jones & Bartlett Learning (2011)
14. Redlich, D., Molka, T., Gilani, W., Blair, G.S., Rashid, A.: Constructs competition miner: Process control-flow discovery of BP-domain constructs. LNCS, vol. 8659, pp. 134–150 (2014)
15. Redlich, D., Molka, T., Gilani, W., Blair, G.S., Rashid, A.: Scalable dynamic business process discovery with the constructs competition miner. CEUR-WP, vol. 1293, pp. 91–107 (2014)
16. Schimm, G.: Process miner - a tool for mining process schemes from event-based data. LNCS, vol. 2424, pp. 525–528. Springer (2002)
17. Schimm, G.: Mining exact models of concurrent workflows. Computers in Industry 53(3), 265–281 (2004)
18. Senderovich, A., Leemans, S., Harel, S., Gal, A., Mandelbaum, A., van der Aalst, W.: Discovering queues from event logs with varying levels of information. BPI. accepted (2015)
19. Solé, M., Carmona, J.: Process mining from a basis of state regions. LNCS, vol. 6128, pp. 226–245. Springer (2010)
20. Wen, L., Wang, J., van der Aalst, W., Huang, B., Sun, J.: A novel approach for process mining based on event types. JIIS 32(2), 163–190 (2009)
21. Wolffensperger, R.: Static and Dynamic Visualization of Quality and Performance Dimensions on Process Trees. Master's thesis, Eindhoven University of Technology (2015)