Scalable Process Discovery and Conformance Checking

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 Considerable amounts of data, including process events, are collected and stored by organisations nowadays. Discovering a process model from such event data and verification of the quality of discovered models are important steps in process mining. Many discovery techniques have been proposed, but none of them combines scalability with strong quality guarantees. We would like such techniques to handle billions of events or thousands of activities, to produce sound models (without deadlocks and other anomalies), and to guarantee that the underlying process can be rediscovered when sufficient information is available. In this paper, we introduce a framework for process discovery that ensures these properties while passing over the log only once and introduce three algorithms using the framework. To measure the quality of discovered models for such large logs, we introduce a model-model and model-log comparison framework that applies a divide-and-conquer strategy to measure recall, fitness and precision. We experimentally show that these discovery and measuring techniques sacrifice little compared to other algorithms, while gaining the ability to cope with event logs of 100,000,000 traces and processes of 10,000 activities on a standard computer.

Keywords: big data, scalable process mining, block-structured process discovery, directly-follows graphs, algorithm evaluation, rediscoverability, conformance checking

1 Introduction

Considerable amounts of data are collected and stored by organisations nowadays. For instance, ERP systems log business transaction events, high tech systems such as X-ray machines record software and hardware events, and web servers log page visits. Typically, each action of a user executed with the system, e.g. a customer filling in a form or a machine being switched on, can be recorded by the system as an event; all events related to the same process execution, e.g. a customer order or an X-Ray diagnosis, are grouped in a *trace* (ordered by their time); an *event log* contains all recorded traces of the system. Process mining aims to extract information from such event logs, for instance social networks, business process models, compliance to rules and regulations, and performance information (e.g. bottlenecks) [6].

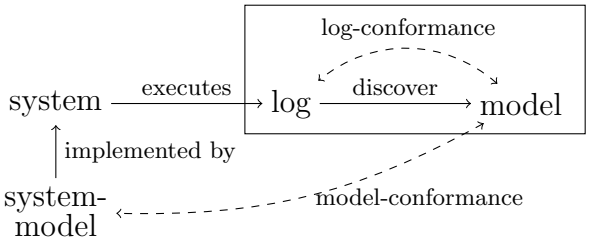


Figure 1: Process discovery and conformance checking in their context. The box contains a typical process mining project's scope.

In this paper, we focus on two process mining challenges: process discovery and conformance checking. Figure 1 shows the context of these two challenges: a real-life business process (a *system*) is running, and the executed process steps are recorded in an event log. In *process discovery*, one assumes that the inner workings of the system are unknown to the analyst and cannot be obtained otherwise. Therefore, process discovery aims to learn a process model from an event log, which describes the system as it actually happened (in contrast to what is assumed has happened) [2]. Two main challenges exist in process discovery: first, one would like to learn an easy to understand model that captures the actual behaviour. Second, the model should have a proper formal interpretation, i.e. have well-defined behavioural semantics and be free of deadlocks and other anomalies (be *sound*) [43]. In Section 2, we explore these challenges in more detail and explore how they are realised in existing algorithms and settings. Few existing algorithms solve both challenges together.

In contrast, *conformance checking* studies the differences between a process model and reality. We distinguish two types of conformance checking. First, the model can be compared with a log. Such log-conformance checking can provide insight into the real behaviour of an organisation, by highlighting which traces deviate from the model, and where in the model deviations occur [2]. Second, the model can be compared with a model of the system (but only if such a model is available). Model-conformance checking can be used to highlight differences between different snapshots of a process, or to verify that a process model conforms to a design made earlier [27]. Moreover, model-conformance checking can be used to evaluate discovery algorithms by choosing a system-model and quantifying the similarity between this chosen model and the models discovered by a discovery algorithms. In Section 2 we discuss both log and model conformance checking in more detail.

Large Event Logs. Current process discovery and conformance checking techniques work reasonably well on smaller event logs, but might have difficulties handling larger event logs. Commercial techniques such as Fluxicon Disco [34] and Celonis Process Mining offer the scalability to handle larger event logs, but usually do not provide strong quality guarantees or do not support parallelism. For instance, discovery techniques typically require the event log to fit in

main memory and require the log to be rather complete, i.e. most of the possible behaviour must be present. Reducing the log size to fit in memory, e.g. through sampling, may yield incomplete event logs, which for discovery may lead to over-fitting models (showing only the behaviour of the sample but not of the system) or under-fitting models (showing arbitrary behaviour beyond the sample log) (for discovery). For conformance checking, such logs may lead to skewed measurements [2].

Event logs can be "big" in two dimensions: many events and many activities (i.e. the different process steps). From our experiments (Section 6), we identified relevant gradations for these dimensions: for the number of activities we identified COMPLEX logs, i.e. containing hundreds of activities, and MORE COMPLEX logs, i.e. containing thousands of activities. For the number of events we identified MEDIUM logs, i.e. containing tens of thousands of events, LARGE logs, i.e. containing millions of events, and LARGER logs, i.e. containing billions of events.

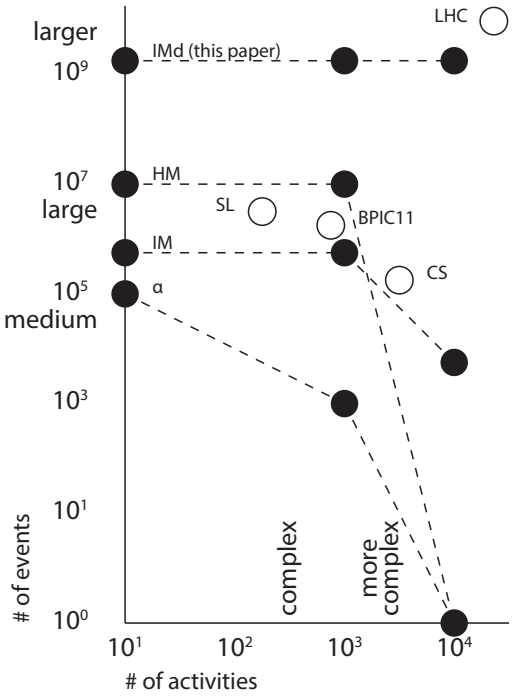


Figure 2: Scales used in this paper. The dots denote maximum number of events several discovery algorithms could handle (Section 6), for Inductive Miner (IM), the Heuristics Miner (HM) and the α-algorithm (α). These algorithms will be introduced in Section 2. The circles denote the mentioned logs.

In our experiments we observed that existing process discovery algorithms with strong quality guarantees, e.g. soundness, can handle MEDIUM logs (see IM in Figure 2; the algorithms will be introduced later). Algorithms not providing such guarantees (e.g. α, HM) can handle LARGE logs, but fail on LARGER logs. In the dimension of the number of different activities, experiments showed that most algorithms could not handle COMPLEX processes. Current conformance checking techniques in our experiments and [50] seem to be unable to handle MEDIUM *or* COMPLEX event logs.

Such numbers of events and activities might seem large for a complaint-handling process in an airline, however processes of much larger complexity

exist. For instance, even simple software tools contains hundreds or thousands of different methods. We obtained a LARGE and COMPLEX log (SL) that will be used in the evaluation. To study or reverse engineer such software, studies [39] have recorded method calls in event logs (at various levels of granularity), and process mining and software mining techniques have been used on small examples to perform the analyses [51,39]. COMPLEX logs can for instance be found in hospitals: the BPI Challenge log of 2011 (BPIC11) [28] was recorded in the emergency department of a Dutch hospital and contains over 600 activities [28]. Even though this log is just COMPLEX and MEDIUM, current discovery techniques have difficulties with this log (we will use it in the evaluation). Other areas in which MORE COMPLEX logs appear are click-stream data from web-sites, such as the web-site of a dot-com start-up, which produced an event log containing 3300 activities (CL) [37]. Even more difficult logs could be extracted from large machines, such as the Large Hadron Collider, in which over 25,000 distributed communicating components form just a part of the control systems [36], resulting in complicated behaviour that could be analysed using scalable process mining techniques. In the future, we aim to extract such logs and apply our techniques to it, but currently, we would only discover a model but were not able to process the discovered model further (no conformance checking and no visualisation on that scale). Nevertheless, we will show in our evaluation that our discovery techniques are able to handle such logs.

Problem Definition and Contribution. In this paper, we address two problems: applying process discovery to LARGER and MORE COMPLEX logs, and conformance checking to MEDIUM and COMPLEX logs. We introduce two scalable frameworks: one for process discovery, the *Inductive Miner - directly-follows framework* (IMD framework), and one for conformance checking: the *Projected Conformance Checking framework* (PCC framework). We instantiate these frameworks to obtain several algorithms, each with their specific purposes. For discovery, we show how to adapt an existing family of algorithms that offers several quality guarantees (the Inductive Miner framework (IM framework) [40]), such that is scales better and works on LARGER and MORE COMPLEX logs. We show that the recursion on event logs used by the IM framework can be replaced by recursion on an abstraction (i.e. the so-called *directly-follows graph* [40]), which can be computed in a single pass over the event log. We show that this principle can also be applied to incomplete event logs (when a discovery technique has to infer missing information) and logs with infrequent behaviour or noise (when a discovery algorithm has to identify and filter the events that would degrade model quality); we present corresponding algorithms. Incompleteness and infrequency/noise pose opposing challenges to discovery algorithms, i.e. not enough and too much information. For these purposes, we introduce different algorithms. For conformance checking, we introduce the configurable divide-and-conquer PCC framework to compare logs to models and models to models. Instead of comparing the complete behaviour over all activities, we decompose the problem into comparing behaviour for subsets of activities. For each such subset, a recall, fitness or precision measure is computed. The averages over these subsets provide

the final measures, while the subsets with low values give information about the location in the model/log/system-model where deviations occur.

Results. We conducted a series of experiments to test how well algorithms handle large logs and complex processes. We found that the IMd framework provides the scalability to handle all kinds of logs up to LARGER and MORE COMPLEX logs (see Figure 2). In a second series of experiments we investigated the ability of several discovery algorithms to rediscover the original system-model: we experimented to analyse the influence of log sizes, i.e. completeness of the logs, to analyse the influence of noise, i.e. randomly appearing or missing events in process executions, and to assess the influence of infrequent behaviour, i.e. structural deviations from the system-model during execution. We found that the new discovery algorithms perform comparable to existing algorithms in terms of model quality, while providing much better scalability. In a third experiment, we explored how the new discovery algorithms handle LARGE and COMPLEX real-life logs.

In all of these experiments, the new PCC framework was applied to assess the quality of the discovered models with respect to the log and where applicable the system, as existing conformance checking techniques could not handle MEDIUM or COMPLEX logs and systems. We compared the new conformance checking techniques to existing techniques, and the results suggest that the new techniques might be able to replace existing less-scalable techniques. In particular, model quality with respect to a log can now be assessed in situations where existing techniques fail, up to LARGE and COMPLEX logs.

Relation to earlier papers. This paper extends the work presented in [41]. We present the new PCC framework, which is able to cope with MEDIUM and COMPLEX logs and models. This allows us to analyse and compare the quality of the IMd framework to other algorithms on such event logs in detail.

Outline. First, process mining is discussed in more detail in Section 2. Second, process trees, directly-follows graphs and cuts are introduced in Section 3. In Section 4, the IMd framework and three algorithms using it are introduced. We introduce the PCC framework in Section 5. The algorithms are evaluated in Section 6 using this PCC framework. Section 7 concludes the paper.

2 Process Mining

In this section, we discuss conformance checking and process discovery in more detail.

2.1 Conformance Checking

The aim of conformance checking is to verify a process model against reality. As shown in Figure 1, two types of conformance checking exist: log-model conformance checking and model-model conformance checking. In log-model conformance checking, reality is assumed to be represented by an event log, while in

model-model conformance checking, a representation of the system is assumed to be present and to represent reality. Such a system is usually given as another process model, to which we refer to as *system-model*.

Log-Model Conformance Checking. To compare a process model to an event log, several quality measures have been proposed [9]. For instance, *fitness* expresses the part of the event log that is represented by the model, *log-precision* expresses the behaviour in the model that is present in the event log, *generalisation* expresses the likelihood that future behaviour will be representible by the model, and *simplicity* expresses the absence of complexity in a model [9] to represent its behaviour.

Several techniques and measures have been proposed to measure fitness and precision, such as token-based replay [54], alignments [9,10] and many more: for an overview, see [56]. Some techniques that were proposed earlier, such as token-based replay [54], cannot handle non-determinism well, i.e. silent activities (τ) and duplicate activities. Later techniques, such as alignments [9], can handle non-determinism by exhaustively searching for the model trace that has the least deviations from a given log trace (according to some cost function). However, even optimised implementations of these techniques cannot deal with MEDIUM or COMPLEX event logs and models [50]. To alleviate this problem, decomposition techniques have been proposed, using the general principles in [4], for instance using passages [3] or single-entry-single-exit decompositions [50]. The PCC framework uses the insights of [4] by checking conformance on small subsets of activities.

Model-Model Conformance Checking. For a more elaborate overview of this field, we refer to [27] and [14].

Typically, the quality of a process discovery algorithm is measured using log-conformance checking, i.e. a discovered model is compared to an event log. Alternatively, discovered model and system-model could be compared directly. Ideally, both would be compared on branching bisimilarity or even stronger notions of equivalence [33], thereby taking the moments of choice into account. However, as an event log describes a language and does not contain information about choices, the discovered model will lack this information as well and we consider comparison based on languages (trace equivalence, a *language* is the set of traces of a log, or the set of traces that a model can produce).

One such technique is [12]. This approach translates the models into partially-ordered runs annotated with exclusive relationships (event structures), which can be generated from process models as well [12]. However, event structures have difficulties supporting loops by their acyclic nature and constructing them requires a full state-space exploration.

As noted in [27], many model-model comparison techniques suffer from exponential complexity due to concurrency and loops in the models. To overcome this problem, several techniques apply an abstraction, for instance using causal footprints [2,30], weak order relations [65] or behavioural profiles [58,38]. Another

technique to reduce the state space is decompose the model in pieces and perform the computations on these pieces individually [38]. Our approach (PCC framework, which applies to both log-model and model-model conformance checking) applies an abstraction using a different angle: we project on subsets of activities, thereby generalising over many of these abstractions, i.e. many relations between the projected activities are captured. references Moreover, our approach handles any formalism of which the executable semantics can be described by Deterministic Finite Automata (DFAs), which includes BPMN, UML-ADs, labelled Petri nets, and allows for models with duplicate activities, silent steps, and anomalies such as potential deadlocks.

2.2 Process Discovery

Process discovery aims at discovering a process model from an event log (see Figure 1). We first sketch some challenges that discovery algorithms face, after which we discuss existing discovery approaches.

Challenges. Several factors challenge process discovery algorithms. One such challenge is that the resulting process model should have well-defined behavioural semantics and be sound [2]. Even though an unsound process model or a model without a language, i.e. without a definition of traces the model expresses, might be useful for manual analysis, conformance checking and other automated techniques can obviously not provide accurate measures on such models [55,43]. The IMD framework uses its representational bias to provide a language and to guarantee soundness: it discovers an abstract hierarchical view on workflow nets [2], *process trees*, that is guaranteed to be sound [20].

Another challenge of process discovery is that for many event logs the different measures, e.g. fitness, log-precision, generalisation and simplicity, are competing, i.e. there might not exist a model that scores well on all criteria [21]. Thus, discovery algorithms have to balance these measures, and this balance might depend on the use case at hand, e.g. auditing questions are best answered using a model with high fitness, optimisations are best performed on a model with high log-precision, implementations might require a model with high generalisation, and human interpretation is eased by a simple model [21].

A desirable property of discovery algorithms is having the ability to rediscover the language of the system (*rediscoverability*); we assume the system and the system-model to have the same behaviour for rediscoverability. Rediscoverability is usually proven using assumptions on both system and event log: the system typically must be of a certain class, and the event log must contain enough correct information to describe the system well [40]. Therefore, three more challenges of process discovery algorithms are to handle 1) *noise* in the event log, i.e. random absence or presence of events [19], 2) *infrequent behaviour*, i.e. behaviour that occurs less frequent than 'normal' behaviour, i.e. the exceptional cases. For instance, most complaints sent to an airline are handled according to a model, but a few complaints are so complicated that they require ad-hoc solutions. This behaviour could be of interest or not, which depends on

the goal of the analysis [2]. 3) *incompleteness*, i.e. the event log does not contain "enough" information. The notion of what "enough" means depends on the discovery algorithm [13,40]. Even though rediscoverability is desirable, it is a formal property, and it is not easy to compare algorithms using it. However, the PCC framework allows to perform experiments to quantify how rediscoverability is influenced by noise, infrequent behaviour and incompleteness.

A last challenge arises from the main focus of this paper, i.e. highly scalable environments. Ideally, a discovery technique should linearly pass over the event log once, which removes the need to keep the event log in memory. In the remainder of this section, we discuss related process discovery techniques and their application in scalable environments.

Sound Process Discovery Algorithms. Process discovery techniques such as the *Evolutionary Tree Miner* (ETM) [20], the *Constructs Competition Miner* (CCM) [52], *Maximal Pattern Mining* (MPM) [46] and *Inductive Miner* (IM) [40] provide several quality guarantees, in particular soundness and some offer rediscoverability, but do not manage to discover a model in a single pass. ETM applies a genetic strategy, i.e. generates an initial population, and then applies random crossover steps, selects the 'best' individuals from the population and repeats. While ETM is very flexible towards the desired log-measures to which respect the model should be 'best' and guarantees soundness, it requires multiple passes over the event log and does not provide rediscoverability.

CCM and IM use a divide-and-conquer strategy on event logs. In the Inductive Miner framework (IM framework), first an appropriate cut of the process activities is selected; second, that cut is used to split the event log into sub logs; third, these sub logs are recursed on, until a base case is encountered. If no appropriate cut can be found, a fall-through ('anything can happen') is returned. CCM works similarly by having several process constructs compete with one another. While both CCM and the IM framework guarantee soundness and IM guarantees rediscoverability (for the class of models described in [42, Appendix A]), both require multiple passes through the event log (the event log is being split and recursed on).

MPM first constructs a prefix-tree of the event log. Second, it folds leaves to obtain a process model, thereby applying local generalisations to detect concurrency. The MPM technique guarantees soundness and fitness, allows for noise filtering and can reach high precision, but it does so at the cost of simplicity: typically, lots of activities are duplicated. Inherently, the MPM technique requires random access to the event log and a single pass does not suffice.

Other Process Discovery Algorithms. Other process discovery techniques are for instance the *α-algorithm* (α) and its derivatives [7,61,62], the *Heuristics Miner* [59] (HM), the *Integer Linear Programming* miner [63] (ILP) and several commercial tools, such as *Fluxicon Disco* (FD) [34] and *Perceptive Process Mining* (two versions: PM1 and PM2).

Some of these guarantee soundness, but do not support explicit concurrency (FD, PM1) [45]. The ILP miner guarantees fitness and can guarantee that the

model is empty after completion, but only for the traces seen in the event log, i.e. the models produced by ILP are usually not sound. However, most of these algorithms (α, HM, ILP, PM2) neither guarantee soundness nor even provide a final marking, which makes it difficult to determine their language (see [42, Appendix E]), thus their models are difficult to analyse automatically (though, such unsound models can still be useful for manual analysis).

Several techniques (e.g. α, HM) satisfy the single-pass requirement. These algorithms first obtain an abstraction from the log, which denotes what activities directly follow one another; in HM, this abstraction is filtered. Second, from this abstraction a process model is constructed. Both α and HM have been demonstrated to be applicable in highly-scalable environments: event logs of 5 million traces have been processed using map-reduce techniques [31]. Moreover, α guarantees rediscoverability, but neither α nor HM guarantees soundness. We show that our approach offers the same scalability as HM and α, but provides both soundness and rediscoverability.

Some commercial tools such as FD and PM1 offer high scalability, but do not support explicit concurrency [45]. Other discovery techniques, such as the language-based region miner [17,18] or the state-based region miner [25] guarantee fitness but neither soundness nor rediscoverability nor work in single pass.

Software Mining. In the field of software mining, similar techniques have been used to discover formal specifications of software. For instance, in [51] and [11], execution sequences of software runs (i.e. traces) are recorded in an event log, from which techniques extract e.g. valid execution sequences on the methods of an api. Such valid execution sequences can then be used to generate documentation. Process discovery differs from software mining in focus and challenges: Process discovery aims to find process models with soundness and concurrency and is challenged e.g. by deviations from the model (noise, infrequent behaviour) and readability requirements of the discovered models, while for software mining techniques, the system is fixed and challenges arise from e.g. nesting levels [51], programmed exceptions [64] and collaborating components [32].

Streams. Another set of approaches that aims to handle even bigger logs assumes that the event log is an unbounded stream of events. Some approaches such as [26,35] work on click-stream data, i.e. the sequence of web pages users visit, to extract for instance clusters of similar users or web pages. However, we aim to extract end-to-end process models, in particular containing parallelism. HM, α and CCM have been shown to be applicable in streaming environments [23,53], and any single pass discovery algorithm (thus the IMd framework as well) can be converted into a streaming algorithm, but will have to deal with the same discovery challenges as described before.

3 Preliminaries

To overcome the limitations of process discovery on large event logs, we will combine the single-pass property of directly-follows graphs with a divide-and-conquer

strategy. This section recalls these existing concepts. The new algorithms are introduced in Section 4.

3.1 Basic Notions

Event Logs. An *event log* is a multiset of *traces* that denote process executions. For instance, the event log $[\langle a, b, c\rangle, \langle b, d\rangle^2]$ denotes the event log in which the trace consisting of the activity a followed by the activity b followed by the activity c was executed once, and the trace consisting of b followed by d was executed twice.

Process Trees. A *process tree* is an abstract representation of a block-structured hierarchical process model, in which the leaves represent the *activities*, i.e. the basic process steps, and the *operators* describe how their children are to be combined [20]. τ denotes the activity whose execution is not visible in the event log. We consider four operators: $\times$, $\rightarrow$, $\wedge$ and $\circlearrowleft$. $\times$ describes the exclusive choice between its children, $\rightarrow$ the sequential composition and $\wedge$ the parallel composition. The first child of a loop $\circlearrowleft$ is the *body* of the loop, all other children are *redo* children. First, the body must be executed, followed by zero or more iterations of a redo child and the body. A formal definition is given in [42, Appendix A]; we give an example here: Figure 3 shows the Petri net corresponding to the process tree $\rightarrow(\times(\wedge(a, b), c), \times(\circlearrowleft(\rightarrow(d, e), f), g))$. Process trees are inherently sound.

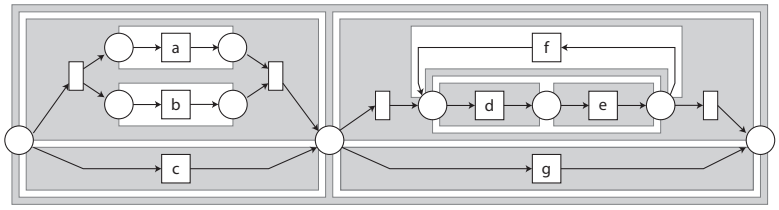


Figure 3: A block-structured hierarchical workflow net; the block-structure is denoted by filled regions (image taken from [44]).

Directly-Follows Graphs. A *directly-follows graph* can be derived from a log and describes what activities follow one another directly, and with which activities a trace starts or ends. In a directly-follows graph, there is an edge from an activity a to an activity b if a is followed directly by b. The weight of an edge denotes how often that happened. For instance, the directly-follows graph of our example log $[\langle a, b, c\rangle, \langle b, d\rangle^2]$ is shown in Figure 4. Note that the multiset of start activities is $[a, b^2]$ and the multiset of end activities is $[c, d^2]$. A directly-follows graph can be obtained in a single pass over the event log with minimal memory requirements [31].

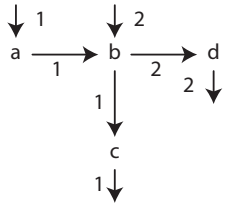


Figure 4: Example of a directly-follows graph.

Cuts, Characteristics and the Inductive Miner framework. A *partition* is a non-overlapping division of the activities of a directly-follows graph. For instance, $(\{a, b\}, \{c, d\})$ is a binary partition of the directly-follows graph in Figure 4. A *cut* is a partition combined with a process tree operator, for instance $(\rightarrow, \{a, b\}, \{c, d\})$. In the IM framework, finding a cut is an essential step: its operator becomes the root of the process tree, and its partition determines how the log is split.

The IM framework [40] discovers the main cut, and projects the given log onto the activity partition. In case of loops, each iteration becomes a new trace in the projected sub-log. Subsequently, for each sub-log its main cut is detected and recursion continues until reaching partitions with singleton elements; these become the leaves of the process tree. If no cut can be found, a generalising fall-through is returned that allows for any behaviour (a "flower model"). By the use of process trees, the IM framework guarantees sound models, and makes it easy to guarantee fitness. The IM framework is formalised in [42, Appendix A].

Suppose that the log is produced by a process which can be represented by a process tree T. Then, the root of T leaves certain characteristics in the log and in the directly-follows graph. The most basic algorithm that uses the IM framework, i.e. IM [40], searches for a cut that matches these characteristics perfectly. Other algorithms using the IM framework are the infrequent-behaviour-filtering Inductive Miner - infrequent (IMf) [43] and the incompleteness-handling Inductive Miner - incompleteness (IMc) [44].

3.2 Cut Detection

Cut definitions are given [42, Appendix A]. Here we describe how the cut detection works. Each of the four process tree operators $\times$, $\rightarrow$, $\wedge$ and $\circlearrowleft$ leaves a different characteristic footprint in the directly-follows graph. Figure 5 visualises these characteristics: for exclusive choice, the activities of one sub-tree will never occur in the same trace as activities of another sub-tree. Hence, activities of the different sub-trees form clusters that are not connected by edges in the directly-follows graph. Thus, the $\times$ cut is computed by taking the connected components of the directly-follows graph.

If two sub-trees are sequentially ordered, all activities of the first sub-tree strictly precede all activists of the second sub-tree; in the directly-follows graph we expect to see a chain of clusters without edges going back. The procedure to

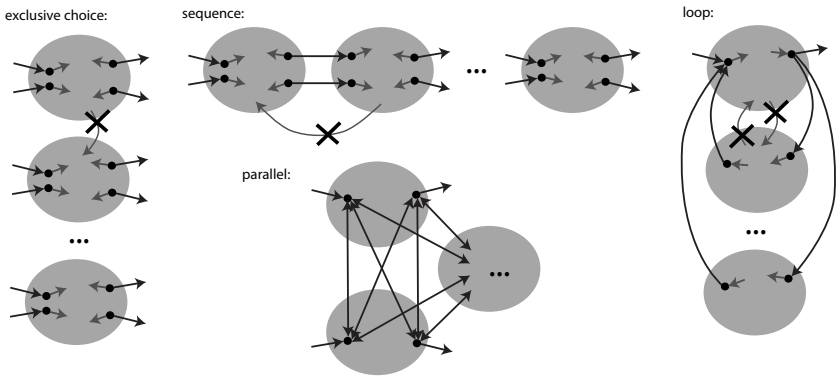


Figure 5: Cut characteristics.

discover a sequence cut is as follows: each activity starts as a singleton set. First, the strongly connected components of the directly-follows graph are computed and merged. By definition, two activities are in a strongly connected component if they are pairwise reachable, and therefore they cannot sequential. Second, pairwise unreachable sets are merged, as if there is no way to reach two nodes in the same trace, they cannot be sequential. Finally, the remaining sets are sorted based on reachability.

The activities of two parallel subtrees can occur in any intertwined order; we expect all possible connections to be present between the child-clusters in the directly-follows graph. To detect parallelism, the graph is negated: the negated graph gets no edge between two activities if both directly-follows edges between these activities are present. If either edge is missing, the negated graph will contain an edge between these two activities. In this negated graph, the partition of the parallel cut is the set of connected components.

In a loop, the directly-follows graph must contain a clear set of start and end activities; all connections between clusters must go through these activities. To detect a loop cut, first the connected components of the directly-follows graph are computed, while excluding the start and end activities. Please note that start and end activities by definition belong to the body of the loop. Second, for each component reachability is used to determine whether it is directed from a start activity to an end activity (body part), or directed the other way round (a redo).

4 Process Discovery Using a Directly-Follows Graph

Algorithms using the IM framework guarantee soundness, and some even rediscoverability, but do not satisfy the single-pass property, as the log is traversed and even copied during each recursive step. Therefore, we introduce an adapted framework: *Inductive Miner - directly-follows* (IMd framework) that recurses on the directly-follows graph instead of the event log. In this section, we first

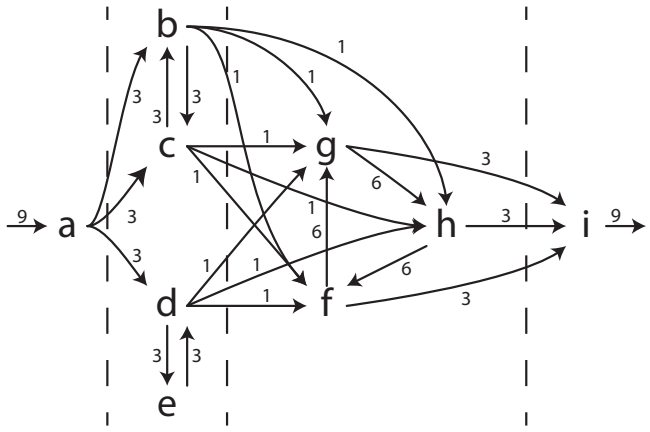


Figure 6: Directly-follows graph D_1 of L. In a next step, the partition $(\{a\}, \{b, c, d, e\}, \{f, g, h\}, \{i\})$, denoted by the dashed lines, will be used.

introduce the IMd framework and a basic algorithm using it. Second, we introduce two more algorithms: one to handle infrequent behaviour; another one that handles incompleteness.

4.1 Inductive Miner - directly-follows

As a first algorithm that uses the framework, we introduce *Inductive Miner - directly-follows* (IMd). We explain the stages of IMd in more detail by means of an example: Let L be $[\langle a, b, c, f, g, h, i\rangle, \langle a, b, c, g, h, f, i\rangle, \langle a, b, c, h, f, g, i\rangle, \langle a, c, b, f, g, h, i\rangle, \langle a, c, b, g, h, f, i\rangle, \langle a, c, b, h, f, g, i\rangle, \langle a, d, f, g, h, i\rangle, \langle a, d, e, d, g, h, f, i\rangle, \langle a, d, e, d, e, d, h, f, g, i\rangle]$. The directly-follows graph D_1 of L is shown in Figure 6.

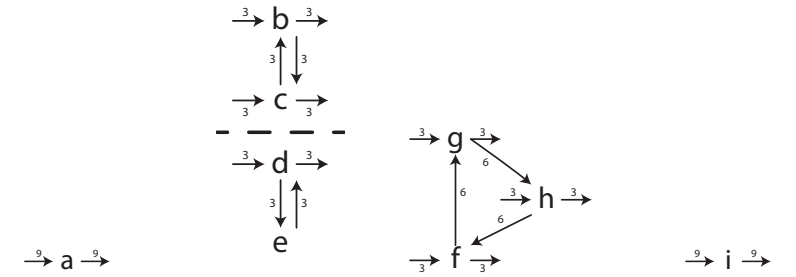
Cut Detection. IMd searches for a cut that perfectly matches the characteristics mentioned in Section 3. As explained, cut detection has been implemented using standard graph algorithms (connected components, strongly connected components), which run in polynomial time, given the number of activities ($O(n)$) and directly-follows edges ($O(n^2)$) in the graph.

In our example, the cut $(\rightarrow, \{a\}, \{b, c, d, e\}, \{f, g, h\}, \{i\})$ is selected: as shown in Figure 5, every edge crosses the cut lines from left to right. Therefore, it perfectly matches the sequence cut characteristic. Using this cut, the sequence is recorded and the directly-follows graph can be split.

Directly-Follows Graph Splitting. Given a cut, the IMd framework splits the directly-follows-graph in disjoint subgraphs. The idea is to keep the internal structure of each of the clusters of the cut by simply projecting a graph on the cluster. Figure 7 gives an example of how D_1 (Figure 6) is split using the

sequence cut that was discovered in our example. If the operator of the cut is $\rightarrow$ or $\circlearrowleft$, the start and end activities of a child might be different from the start and end activities of its parent. Therefore, every edge that enters a cluster is counted as a start activity, and an edge leaving a cluster is counted as an end activity. In our example, the start activities of cluster $\{f, g, h\}$ are those having an incoming edge not starting in $\{f, g, h\}$, and correspondingly for end activities. The result is shown in Figure 7a. In case of $\times$, no edges leave any cluster and hence the start and end activities remain unchanged. In case of $\wedge$, removed edges express the arbitrary interleaving of activities in parallel clusters; removing this interleaving information does not change with which activities a cluster may start or end, thus start and end activities remain unchanged.

The choices for a sequence cut and the split directly-follows graphs are recorded in an intermediate tree: $\rightarrow((D_2), (D_3), (D_4), (D_5))$, denoting a sequence operator with 4 unknown sub-trees that are to be derived from 4 directly-follows graphs.



(a) D_2 of $\{a\}$ (b) D_3 of $\{b, c, d, e\}$ (c) D_4 of $\{f, g, h\}$ (d) D_5 of $\{i\}$

Figure 7: Split directly-follows graphs of D_1. The dashed line is used in a next step and denotes another partition.

Recursion. Next, IMD recurses on each of the new directly-follows graphs (find cut, split, ...) until a base case (see below) is reached or no perfectly matching cut can be found. Each of these recursions returns a process tree, which in turn can be inserted as a child of an operator identified in an earlier recursion step.

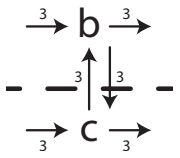
Base Case. Directly-follows graphs D_2 (Figure 7a) and D_5 (Figure 7d) contain base cases: in both graphs, only a single activity is left. The algorithm turns these into leaves of the process tree and inserts them at the respective spot of the parent operator. In our example, detecting the base cases of D_2 and D_5 yields the intermediate tree $\rightarrow(a, (D_3), (D_4), i)$, in which D_3 and D_4 indicate directly-follows graphs that are not base cases and will be recursed on later.

Fall-Through. Consider D_4 as, shown in Figure 7c. D_4 does not contain unconnected parts, so does not contain an exclusive choice cut. There is no sequence

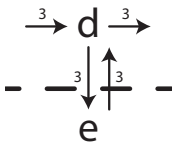
cut possible, as f, g and h form a strongly connected component. There is no parallel cut as there are no dually connected parts, and no loop cut as all activities are start and end activities. Thus, IMD selects a fall-through, being a process tree that allows for any behaviour consisting of f, g and h (a flower model $\circlearrowleft(\tau, f, g, h)$, having the language $(f|g|h)*$). The intermediate tree of our example up till now becomes $\rightarrow(a, (D_3), \circlearrowleft(\tau, f, g, h), i)$ (remember that τ denotes the activity of which the execution is invisible).

Example Continued. In D_3, shown in Figure 7b, a cut is present: $(\times, \{b, c\}, \{d, e\})$: no edge in D_3 crosses this cut. The directly-follows graphs D_6 and D_7, shown in Figures 8a and 8b, result after splitting D_3. The tree of our example up till now becomes $\rightarrow(a, \times((D_6), (D_7)), \circlearrowleft(\tau, f, g, h), i)$.

In D_6, shown in Figure 8a, a parallel cut is present, as all possible edges cross the cut, i.e. the dashed line, in both ways. The dashed line in D_7 (Figure 8b) denotes a loop cut, as all connections between $\{d\}$ and $\{e\}$ go via the set of start and end activities $\{d\}$. Four more base cases give us the complete process tree $\rightarrow(a, \times(\wedge(b, c), \circlearrowleft(d, e)), \circlearrowleft(\tau, f, g, h), i)$.



(a) D_6 of $\{b, c\}$ in D_3



(b) D_7 of $\{d, e\}$ in D_3

Figure 8: Split directly-follows graphs. Dashed lines denote cuts, which are used in the next steps.

To summarise: IMD selects a cut, splits the directly-follows graph and recurses until a base case is encountered or a fall-through is necessary. As each recursion removes at least one activity from the graph and cut detection is $O(n^2)$, IMD runs in $O(n^3)$, in which n is the number of activities in the directly-follows graph.

By the nature of process trees, the returned model is sound. By reasoning similar to IM [40], IMD guarantees rediscoverability on the same class of models (see [42, Appendix A]), i.e. assuming that the model is representable by a process tree without using duplicate activities, and it is not possible to start loops with an activity they can also end with [40]. This makes IMD the first single-pass algorithm to offer these guarantees.

4.2 Handling Infrequency and Incompleteness

The basic algorithm IMD guarantees rediscoverability, but, as will be shown in this section, is sensitive to both infrequent and incomplete behaviour. To solve this, we introduce two more algorithms using the IMD framework.

Infrequent Behaviour. Infrequent behaviour in an event log is behaviour that occurs less frequent than 'normal' behaviour, i.e. the exceptional cases. For instance, most complaints sent to an airline are handled according to a model, but a few complaints are so complicated that they require ad-hoc solutions. This behaviour could be of interest or not, which depends on the goal of the analysis.

Consider again directly-follows graph D_3, shown in Figure 7b, and suppose that there is a single directly-follows edge added, from c to d. Then, $(\times, \{b, c\}, \{d, e\})$ is not a perfectly matching cut, as with the addition of this edge the two parts $\{b, c\}$ and $\{d, e\}$ became connected. Nevertheless, as 9 traces showed exclusive-choice behaviour and only one did not, this single trace is probably an outlier and in most cases, a model ignoring this trace would be preferable.

To handle these infrequent cases, we apply a strategy similar to IMf [43] and use the IMd framework to define another discovery algorithm: *Inductive Miner - infrequent - directly-follows* (IMfD). Infrequent behaviour introduces edges in the directly-follows graph that violate cut requirements. As a result, a single edge makes it impossible to detect an otherwise very strong cut. To handle this, IMfD first searches for existing cuts as described in Section 4.1. If none is found (when IMd would select a fall through), the graph is filtered by removing edges which are infrequent with respect to their neighbours. Technically, for a parameter $0 \leq h \leq 1$, for an activity a we keep the outgoing edges that occur more than h times the most occurring outgoing edge of a (a formal definition is given in [42, Appendix A]). Start and end activities are filtered similarly.

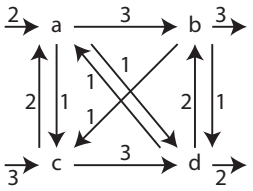


Figure 9: An incomplete directly-follows graph.

Incompleteness. A log in a "big-data setting" can be assumed to contain lots of behaviour. However, we only see example behaviour and we cannot assume to have seen all possible traces, even if we use the rather weak notion of directly-follows completeness [44] as we do here. Moreover, sometimes smaller subsets of the log are considered, for instance when performing slicing and dicing in the context of process cubes [5]. For instance, an airline might be interested in comparing the complaint handling process for several groups of customers, to gain insight in how the process relates to age, city and frequent-flyer level of the customer. Then, there might be combinations of age, city and frequent-flyer

level that rarely occur and the log for these customers might contain too little information.

If the log contains little information, edges might be missing from the directly-follows graph and the underlying real process might not be rediscovered. Figure 9 shows an example: the cut $(\{a, b\}, \{c, d\})$ is not a parallel cut as the edge (c, b) is missing. As the event log only provides example behaviour, it could be that this edge is possible in the process, but has not been seen yet. Given this directly-follows graph, IMd can only give up and return a fall-through flower model, which yields a very imprecise model. However, choosing the parallel cut $(\{a, b\}, \{c, d\})$ would obviously be a better choice here, providing a better precision.

To handle incompleteness, we introduce *Inductive Miner - incompleteness - directly-follows* (IMcD), which adopts ideas of IMc [44] into the IMd framework. IMcD first applies the cut detection of IMd and searches for a cut that perfectly matches a characteristic. If that fails, instead of a perfectly matching cut, IMcD searches for the most probable cut of the directly-follows graph at hand.

IMcD does so by first estimating the most probable behavioural relation between any two activities in the directly-follows graph. In Figure 9, the activities a and b are most likely in a sequential relation as there is an edge from a to b. a and c are most likely in parallel as there are edges in both directions. Loops and choices have similar local characteristics. For each pair of activities x and y the probability $P_r(x, y)$ that x and y are in relation R is determined. The best cut is then a partition into sets of activities X and Y such that the average probabilities that $x \in X$ and $y \in Y$ are in relation R is maximal. For a formal definition, please refer to [44].

In our example, the probability of cut $(\wedge, \{a, b\}, \{c, d\})$ is the average probability that (a, c), (a, d), (b, c) and (b, d) are parallel. IMcD chooses the cut with highest probability, using optimisation techniques. This approach gives IMcD a run time exponential in the number of activities, but still requires a single pass over the event log.

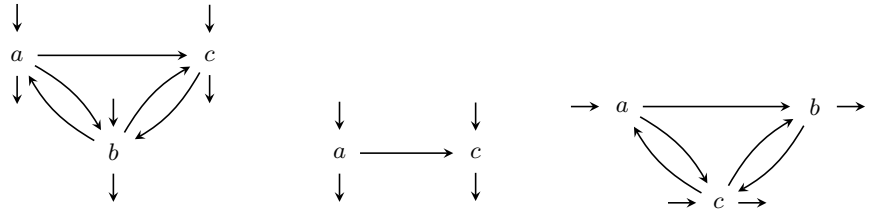
4.3 Limitations

The IMd framework imposes some limitations on process discovery. We discuss limiting factors on the challenges identified in Section 2: rediscoverability, handling incompleteness, handling noise and handling infrequent behaviour, and balancing fitness, precision, generalisation and simplicity.

Limitations on rediscoverability of the IMd framework are similar to the IM framework: the system must be a process tree and adhere to some restrictions, and the log must be directly-follows complete (as discussed before). If the system does not adhere to the restrictions, then IMd framework will not give up but rather try to discover as much process-tree like behaviour as possible. For instance, if a part of the process is sequential, then IMd framework might be able to discover this, even though the other parts of the process are not block-structured. Therefore, in practice, such models can be useful [22,16]. To formally

investigate what happens on non-block structured models would be an interesting subject of further study. For the remaining non-block structured parts, the flexibility of IMD framework easily allows for future customisations, e.g. [48].

If the log is incomplete, in some cases log-based discovery techniques might handle this incompleteness better. For instance, take the process tree $P_1 = \wedge(a, b, c)$ and an event log $L = \{\langle a, b, c\rangle, \langle c, b, a\rangle, \langle b, a, c\rangle, \langle a, c, b\rangle\}$. Figure 10a shows the directly-follows graph of L. Log L is not directly-follows complete with respect to P_1, as the edge (c, a) is missing. Both IM and IMD will first detect a concurrent cut $(\wedge, \{a, c\}, \{b\})$. The sub-logs after splitting by IM are $\{\langle a, c\rangle, \langle c, a\rangle\}$ and $\{\langle b\rangle\}$, in which the missing directly-follows edge a, c pops up and enables the rediscovery of P_1. In IMD, however, the directly-follows graph is split, resulting in the directly-follows graph for a, c shown in Figure 10b, from which P_1 cannot be rediscovered. In Section 6, we will illustrate that the effect of this limitation is limited on larger examples.



(a) directly-follows graph of L.

(b) after splitting by IMD.

(c) the directly-follows graph of both P_2 and L_2.

Figure 10: A directly-follows graph that does not suffice to discover concurrency where the log does (a, b) and an ambiguous directly-follows graph (c).

If the log contains noise and/or infrequent behaviour, then the IMD framework might choose a wrong cut at some point (as discussed in Section 4.2), possibly preventing rediscovery of the system. The noise handling abilities of log-based and directly-follows based algorithms differ in details; in both, noise and infrequent behaviour manifest as superfluous edges in a directly-follows graph. On one hand, in IM, such wrong edges might pop up during recursion by reasoning similar to the incompleteness case (which could be harmful), while using the same reasoning, log-based algorithms might have more information available to filter such edges again (which could be beneficial). In the evaluation, we will investigate this difference further.

Given an event log, both types of algorithms have to balance fitness, precision, generalisation and simplicity. For directly-follows based algorithms, this balance might be different.

For instance, a desirable property of discovery algorithms is the ability to preserve fitness, i.e. to discover a model that is guaranteed to include all behaviour

seen in the event log. For directly-follows based algorithms, this is challenging. For instance, Figure 10c shows a complete directly-follows graph of the process tree $P_2 = \wedge(\rightarrow(a, b), c)$. However, it is also the directly-follows graph of the event log $L_2 = \{\langle a, c, b, c, a, b\rangle, \langle c\rangle\}$. Hence, if a fitness-preserving directly-follows based discovery algorithm would be applied to the directly-follows graph in Figure 10c, this algorithm could not return P_2 and has to seriously underfit/generalise to preserve fitness since the behaviour of both needs to be included. Hence, P_2 could never be returned. Therefore, we chose the IMd framework to not guarantee fitness, while the IM framework by its log splitting indirectly takes such concurrency dependencies into account. Please note that this holds for any pure directly-follows based process discovery algorithm (see the limitations of the α-algorithm). Generalisation, i.e. the likelihood that future behaviour will be representable by the model, is similarly influenced.

Algorithms of the IM framework can achieve a high log-precision if it can avoid fall-throughs such as the flower model [43]. Thus, IM framework achieves the highest log-precision if it can find a cut. The same holds for IMd framework, and therefore we expect log-precision to largely depend on the cut selection. In the evaluation, we will investigate log-precision further.

The influence of directly-follows based algorithms on simplicity highly depends on the chosen simplicity measure: both IM framework and IMd framework return models in which each activity appears once.

5 Comparing Models to Logs and Models

We want to evaluate and compare our algorithm to other algorithms regarding several criteria.

- First, we want to compare algorithms based on the size of event logs they can handle, as well as the quality of the produced models. In particular both recall/fitness and precision (with respect to the given system-model or log) need to be compared, as trivial models exist that achieve either perfect recall or perfect precision, but not both.
- Second, we want to assess under which conditions the new algorithms achieve rediscoverability, i.e. under which conditions the partial or incorrect information in the event log allows to obtain a model that has exactly the same behaviour as the original system that produced the event log. More formally, under which conditions (and up to which sizes of systems) has the discovered model the same language as the original system.

Measuring model quality is crucial in typical process mining workflows as shown in the introduction. However, as discussed in Section 2, existing techniques for measuring model quality (on log or model) cannot handle LARGE and COMPLEX logs and processes. Our technique has to overcome two difficulties: (1) it has to compute precision and recall of very large, possibly infinite languages; and (2) it should allow a fine-grained measurement of precision and recall allowing to identify particular activities or behavioural relations where the the model and the log/other model differ.

We first introduce this technique for measuring recall and precision of two models - with the aim of analysing rediscoverability of process mining algorithms (Section 5.1). Second, we adopt this technique to also compare a discovered model to a (possibly very large) event log (Section 5.2). We use the techniques in our evaluation in Section 6.

5.1 Model-Model Comparison

The *recall* of a model S and a model M describes the part of the behaviour of S that is captured by M (compare to the conventional fitness notion in process mining), while *precision* captures the part of the behaviour of M that is also possible in S.

For a complex model S and a complex model M, the direct language-based comparison of the two models by constructing and comparing their state spaces might suffer from the state explosion problem, and hence be prohibitively expensive. Therefore, the framework approximates recall and precision by measuring them on subsets of activities, i.e. we avoid the state explosion problem by considering small submodels, and averaging over all such subsets. The framework is applicable to any process model formalism and process discovery algorithm, as long as the languages of the models used can be described as deterministic finite automata (DFAs). We first introduce the general idea of the framework, after which we describe its steps in more detail.

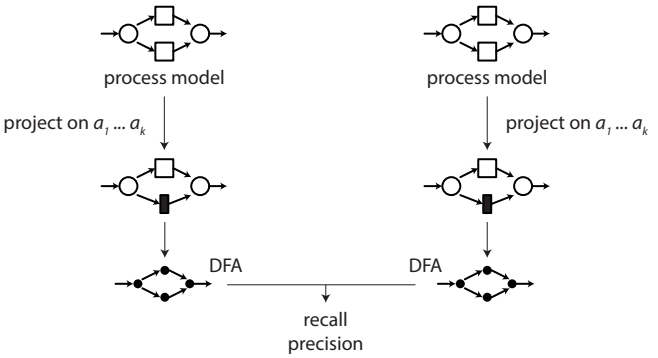


Figure 11: Evaluation framework for process discovery algorithms.

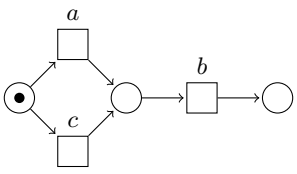
Framework Figure 11 gives an overview of the model-model evaluation framework; formally, the framework takes as input a model S, a model M and an integer k. S and M must be process models, but can be represented using any formalism with executable semantics.

To measure recall and precision of S and M, we introduce a parameterised technique in which k defines the size of the subsets of activities for which recall

and precision shall be computed. Take a subset of activities $A = \{a_1 \dots a_k\}$, such that $A \subseteq \Sigma(M) \cup \Sigma(S)$, and $|A| = k$. Then, S and M are projected onto A, yielding $S|_A$ and $M|_A$ (we will show how the projection is performed below). From these projected $S|_A$ and $M|_A$, deterministic finite automata (DFAs) are generated, which are compared to quantify recall and precision. These steps are repeated for all such subsets A, and the average recall and precision over all subsets is reported.

As we aim to apply this method to test rediscoverability, a desirable property is that precision and recall should be 1 if and only if $\mathcal{L}(S) = \mathcal{L}(M)$. Theorem 1, given later, states that this is the case for the class of process trees used in this paper.

As a running example, we will compare two models, being a process tree and a Petri net. Both are shown in Figure 12.



$\times(\wedge(a, b), \circlearrowleft(c, d))$

(a) Process tree S.

(b) Petri net M.

Figure 12: Example models S and M.

In the remainder of this section, we describe the steps of the framework in more detail after which we give an example and prove Theorem 1.

Projection Many process formalisms allow for projection on subsets of activities; we give a definition for process trees here and sketch projection for Petri nets in [42, Appendix E].

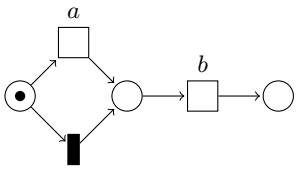
A process tree can be projected on a set of activities $A = \{a_1 \dots a_k\}$ by replacing every leaf that is not in A with τ: (in which $\oplus$ is any process tree operator)

$$a|_A = \text{if } a \in A \text{ then } a \text{ else } \tau$$
$$\tau|_A = \tau$$
$$\oplus(M_1 \dots M_n)|_A = \oplus(M_1|_A \dots M_n|_A)$$

After projection, a major problem reduction (and speedup) can be achieved by applying structural language-preserving reduction rules to the process tree, such as the rules described in [42, Appendix B].

In principle, any further language preserving state-space reduction rules can be applied; we will not explore further options in this paper.

If we project our example process tree and Petri net onto activities a and b, we obtain the models as shown in Figure 13.



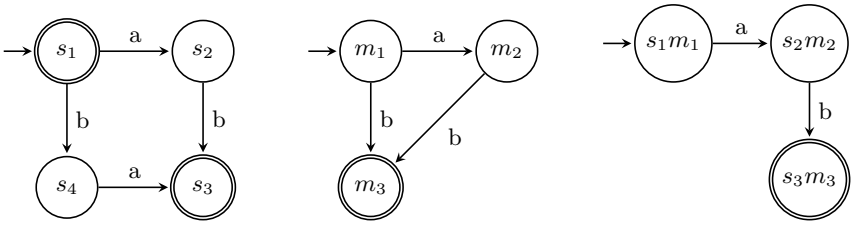
$\times(\wedge(a,b),\circlearrowleft(\tau,\tau))$ (a) $S|_{a,b}$. $\times(\wedge(a,b),\tau)$ (b) $S|_{a,b}$ reduced. (c) $M|_{a,b}$.

Figure 13: Example models S and M projected/reduced.

Process Model to Deterministic Finite Automaton An *automaton* describes a language based on an alphabet Σ. The automaton starts in its initial state; from each state, transitions labelled with activities from Σ denote the possible steps that can be taken from that state. A state can be an accepting state, which denotes that a trace which execution ends in that state is accepted by the automaton. An automaton with a finite set of states is a *non-deterministic finite automaton* (NFA). In case that the automaton does not contain a state from which two transitions with the same activity leave, the automaton is a *deterministic finite automaton* (DFA). Each NFA can be translated into a DFA and a language for which a DFA exist is a *regular* language; for each DFA, there exists a reduced unique minimal version [47].

Process tree are defined using regular expressions in [42, Appendix A], which can be transformed straightforwardly into an NFA (we used the implementation [49], which provides a shuffle-operator). Secondly, a simple procedure transforms the NFA into a DFA [47].

The translation of our example $S|_{\{a,b\}}$ and $M|_{\{a,b\}}$ to DFAs results in the automata shown in figures 14a and 14b.



(a) DFA($S|_{\{a,b\}}$). (b) DFA($M|_{\{a,b\}}$). (c) DFAc($S, M, \{a,b\}$).

Figure 14: DFAs for S and M projected to $\{a,b\}$ and reduced, and their conjunction.

Comparing Deterministic Finite Automata Precision is defined as the part of behaviour in $M|_A$ that is also in $S|_A$. Therefore, first the conjunction

Table 1: Outgoing edge counting of our running example.

recall for activity subset $\{a, b\}$			
state in DFA($S|_{\{a,b\}}$)	outgoing edges	state in DFAc($S, M, \{a, b\}$)	outgoing edges
s_1	3	s_1m_1	1
s_2	1	s_2m_2	1
s_3	1	s_3m_3	1
s_4	1	-	0
precision for activity subset $\{a, b\}$			
state in DFA($M|_{\{a,b\}}$)	outgoing edges	state in DFAc($S, M, \{a, b\}$)	outgoing edges
m_1	2	s_1m_1	1
m_2	1	s_2m_2	1
m_3	1	s_3m_3	1

DFA($S|_A$)∩DFA($M|_A$) (which we abbreviate to DFAc(S, M, A)) of these DFAs is constructed, which accepts the traces accepted by both $S|_A$ and $M|_A$. Figure 14c shows the conjunctive DFA of our running example. Then, precision is measured similarly to several existing precision metrics, such as [8]: we count the outgoing edges of all states in DFA($M|_A$), and compare that to the outgoing edges of the corresponding states (s, m) in DFAc(S, M, A) (for ease of notation, we consider an automaton as a set of states here):

$$precision(S, M, A) = \frac{\sum_{m \in \text{DFA}(M|_A)} \sum_{(s,m) \in \text{DFAc}(S,M,A)} \text{outgoing edges of } (s, m) \text{ in DFAc}(S, M, A)}{\sum_{m \in \text{DFA}(M|_A)} \sum_{(s,m) \in \text{DFAc}(S,M,A)} \text{outgoing edges of } m \text{ in DFA}(M|_A)}$$

If DFA($M|_A$) has no edges at all (i.e. describes the empty language), we define

$$precision(S, M, A) = \begin{cases} 0 & \text{if DFAc}(S, M, A) \text{ has edges} \\ 1 & \text{if DFAc}(S, M, A) \text{ has no edges} \end{cases}$$

Please note that we count acceptance as an outgoing edge, and that states may be counted multiple times if they are used multiple times in DFAc. Recall is defined as the part of behaviour in $S|_A$ that is not in $M|_A$, i.e. $recall(S, M, A) = precision(M, S, A)$. In our example (see Table 1), recall for (a, b) is $\frac{1+1+1}{3+1+1+1} = 0.5$; precision is $\frac{1+1+1}{2+1+1} = 0.75$.

Over all activities For an alphabet Σ, finally the previous steps are repeated for each set of activities $\{a_1 \dots a_k\} \subseteq \Sigma$ of size k and the results are averaged:

$$recall(S, M, k) = \frac{\sum_{A \subseteq \Sigma(S) \cup \Sigma(M) \wedge |A|=k} recall(S, M, a_1 \dots a_k)}{|\{A \subseteq \Sigma(S) \cup \Sigma(M) \wedge |A| = k\}|}$$

$$precision(S, M, k) = recall(M, S, k)$$

Note that we assume a closed-world here, i.e. the alphabet Σ is assumed to be the same for S and M. If an activity is missing from M, we therefore consider M to express that the activity can never happen.

Framework Guarantees Using these definitions, we prove that the framework is able to detect language equivalence between process trees of the class that can be rediscovered by IM and IMD. This theorem will be useful in later evaluations, where from recall and precision being 1, we can conclude that the system was rediscovered.

Theorem 1. *Let S and M be process trees without duplicate activities and without τs. Then, $recall(S, M, 2) = 1 \wedge precision(S, M, 2) = 1 \Leftrightarrow \mathcal{L}(S) = \mathcal{L}(M)$.*

The proof strategy is to prove the two directions of the implication separately, using that for such trees, there exists a language-unique normal form [40, Corollary 15]. For a detailed proof, see [42, Appendix C]. As for sound free-choice unlabeled workflow nets without short loops the directly-follows graph defines a unique language [57], Theorem 1 applies to these nets as well.

Corollary 2. *Let S and M be sound free-choice unlabelled workflow nets without short loops. Then, $recall(S, M, 2) = 1 \wedge precision(S, M, 2) = 1 \Leftrightarrow \mathcal{L}(S) = \mathcal{L}(M)$.*

Unfortunately, this theorem does not hold for general process trees. For instance, take $S = \times(a, b, c, \tau)$ and $M = \times(a, b, c)$. For $k = 2$, the framework will consider the subtrees $\times(a, b, \tau)$, $\times(a, c, \tau)$ and $\times(b, c, \tau)$ for both S and M, thus will not spot any difference: $recall = 1$ and $precision = 1$, even though the languages of S and M are clearly different. Only for $k = 3$, the framework will detect the difference.

In Section 6, we use the algorithm framework to test incompleteness, noise and infrequent behaviour on large models. Before that, we first show that the ideas of the framework can also be used to compare models to event logs.

5.2 Log-Model Comparison

In order to evaluate models with respect to event logs, the framework in Section 5.1 is adapted as follows: Figure 15 shows an overview: the framework starts from an event log L and a model M (in a process mining setting, M would have been discovered from L). First, L and M are projected on a set of activities A. A log is projected by removing the non-projected events, e.g. $\{\langle a, b, c\rangle, \langle c, d\rangle\}|_{\{a,b\}} = \{\langle a, b\rangle, \langle\,\rangle\}$. Second, from both projections a DFA is constructed. Precision is computed as in Section 5.2, i.e. by comparing $\text{DFA}(L|_A)$ and $\text{DFA}(M|_A)$. For fitness, it is desirable that the frequencies of traces are taken into account, such that a trace that appears 10,000 times in L contributes more to the fitness value than a trace that appears just once. Therefore, we compute fitness as the fraction of traces of $L|_A$ that can be replayed on $\text{DFA}(M|_A)$:

$$fitness(L, M, A) = \frac{|[t \in L|_A | t \in \text{DFA}(M|_A)]|}{|L|_A|}$$

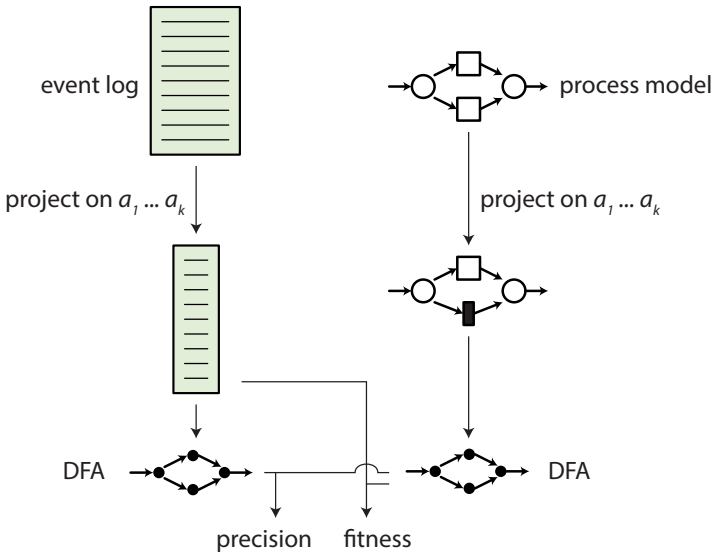


Figure 15: Evaluation approach for logs vs models.

If the log contains no traces, we define fitness to be 1.

Note that L is a multiset: if a trace appears multiple times in L, it contributes multiple times as well. This is repeated for all subsets of activities A of a certain length k, similarly to the model-model comparison. Note that besides a fitness/precision number, the subsets A also provide clues where deviations in the log and model occur.

6 Evaluation

To understand the impact of "big data" event logs on process discovery and quality assessment, we conducted a series of experiments to answer the following research questions:

RQ1 What is the largest event log (number of events/traces or number of activities) that process discovery algorithms can handle?

RQ2 Are single-pass algorithms such as the algorithms of the IMd framework able to rediscover the system? How large do event logs have to be in order to enable this rediscovery? How do these algorithms compare to classical algorithms?

RQ3 Can the system also be rediscovered if an event log contains unstructured noise or structured infrequent behaviour? How does model quality of the newly introduced algorithms suffer compared to other algorithms?

RQ4 Can PCC framework handle logs that existing measures cannot handle? How do both sets of measures compare on smaller logs?

To answer the first three questions, we conducted four similar experiments, as shown in Figure 16: we choose a system, generate an event log and discover a process model, after which we measure how well the discovered model represents the system using log-precision and recall. Of these experiments, one focuses on scalability, i.e. the ability in handling big event logs and complex systems (Section 6.1), one on handling incompleteness (Section 6.2), one on handling noise (Section 6.3) and one on handling infrequent behaviour (Section 6.4).

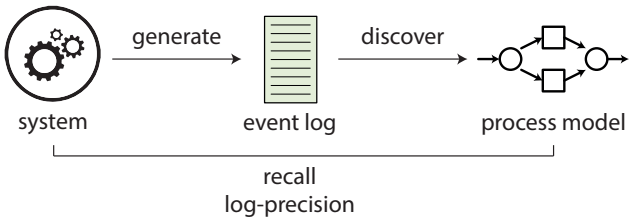


Figure 16: Set-up of our evaluation.

To answer RQ4, we conducted another experiment: we use real-life logs, apply discovery algorithms and measure fitness and log-precision, using both the PCC framework and existing measures (Section 6.5). All algorithms of the IMD framework and PCC framework are implemented as plug-ins of the ProM framework[1], taking as input a directly-follows graph. Directly-follows graphs were generated using an external Python script. For more details on the set-up, please refer to [42, Appendix D].

6.1 Scalability of IMD vs Other Discovery Algorithms

First, we compare the IMD algorithms with several other discovery algorithms in their ability to handle big event logs and complex systems using limited main memory.

Set-up. All algorithms were tested on the same set of XES event logs, which have been created randomly from three process trees, of (A) 40 activities, (B) 1,000 activities and (C) 10,000 activities. The three trees have been generated randomly.

For each tree, we first generate a random log of t traces, starting t at 1. Second, we test whether an algorithm returns a model for that log when allocated 2GB of main memory, i.e. the algorithm terminates with a result and does not crash. If successful, we multiply t by 10 and repeat the procedure. The maximum t is recorded for each algorithm and process tree A, B and C.

[1] Available for download at `http://promtools.org`

Besides the new algorithms introduced in this paper, the following algorithms were included in the experiment:

α-algorithm	(α)	[7]	ProM 6.5.1a
Heuristics Miner	(HM)	[60]	ProM 6.5.1a
Integer Linear Programming	(ILP)	[63]	ProM 6.5.1a
immediately_follows_cnet_from_log	(P-IF)	[24]	PMLAB
pn_from_ts	(P-PT)	[24]	PMLAB
Inductive Miner	(IM)	[40]	ProM 6.5.1a
Inductive Miner - infrequent	(IMf)	[43]	ProM 6.5.1a
Inductive Miner - incompleteness	(IMc)	[44]	ProM 6.5.1a
IM - directly-follows	(IMd)	this paper	ProM 6.5.1a
IM - infrequent - directly-follows	(IMfD)	this paper	ProM 6.5.1a
IM - incompleteness - directly-follows	(IMcD)	this paper	ProM 6.5.1a

The soundness-guaranteeing algorithms ETM, CCM and MPM were not included, as ETM is a non-deterministic algorithm and requires long run times to discover reasonable models, and as for CCM and MPM, there is no implementation publicly available. It would be interesting to test these as well.

Event Logs. The complexities of the event logs are shown in Table 2; they were generated randomly from trees A, B or C. From this table, we can deduce that the average trace length in (A) is 37 events, in (B) 109 and in (C) 764; [42, Appendix F] shows additional statistics. Thus, the average trace length increases with the number of activities.

The largest log we could generate for A was 217GB (10^8 traces), limited by disk space. For the trees B and C, the largest logs we could generate were 10^6 and 10^5 traces, but now limited by RAM. For the bigger logs, the traces were directly transformed into a directly-follows graph and the log itself was not stored. In Table 2, these logs are marked with *.

Compared with [41], tree B was added and the logs of trees A and C were regenerated. Therefore, the log generated for these trees are slightly different from [41]. However, the conclusions were not influenced by this.

Results. Table 3 shows the results. Results that could not be obtained are marked with * (for instance, IMc and IMcD ran for over a week without returning a result).

This experiment clearly shows the scalability of the IMd framework, which handles larger and more complex logs easily (IMd and IMfD handle 10^8 traces, $7 * 10^{10}$ events and 10^4 activities). Moreover, it shows the inability of existing approaches to handle larger and complex logs: the most scalable other algorithms were IM and IMf, that both handled only 1,000 traces. Furthermore, it shows the limited use sampling would have on such logs (logs manageable for other algorithms, i.e. 1,000 traces for tree C, do not contain all activities yet). We discuss the results in detail in [42, Appendix G].

Table 2: log complexity (* denotes that a directly-follows graph was generated).

	A: 40 activities		B: 1,000 activities COMPLEX		C: 10,000 activities MORE COMPLEX	
traces	events	activities	events	activities	events	activities
1	21	21	190	52	81	66
10	309	40	922	359	8,796	1,932
10^2	3,567	40	9,577	802	77,664	7,195
10^3	37,415	40	112,821	973	780,535	9,589
10^4	370,687	40	1,106,495	999	7,641,398	9,991
10^5	3,697,424	40	10,908,461	1,000	76,663,981	10,000
10^6	36,970,718	40	109,147,057	1,000	764,585,193	*10,000
10^7	369,999,523	40	1,090,802,965	*1,000	7,644,466,866	*10,000
10^8	3,700,046,394	40	10,908,051,834	*1,000	76,477,175,661	*10,000

Table 3: scalability: maximum number of traces an algorithm could handle.

	A: 40 activities	B: 1,000 activities	C: 10,000 activities
	traces	traces	traces
α	10,000	100	1
HM	1,000,000	1,000,000	1
ILP	1,000	100	1
P-IF	10,000	*0	*1
P-PT	10,000	*0	*1
IM	100,000	100,000	1,000
IMD	100,000,000	100,000,000	100,000,000
IMF	100,000	100,000	1,000
IMFD	100,000,000	100,000,000	100,000,000
IMC	† 100,000	1	*10
IMCD	† 100,000,000	1	*10

Time. Timewise, it took a day to obtain a directly-follows graph from the log of 10^8 traces of tree A, (using the pre-processing Python script) after that discovering a process model was a matter of seconds for IMD and IMFD. For the largest logs they could handle, P-IF, α, HM, IM, IMF and IMC took a few minutes; P-PT took days, ILP a few hours. In comparison, on the logs that ILP could handle, creating a directly-follows graph took a few seconds, just as applying IMD.

6.2 The Influence of Incompleteness on Rediscoverability

To answer RQ2, i.e. whether single-pass algorithms are able to rediscover the system, how large logs need to be in order to enable rediscovery, and how these algorithms compare to classical algorithms, we performed a second experiment. In complex processes, such as B and C, information can be missing from logs if the logs are not large enough: Table 2 shows that in our example, 10^5 traces were necessary for B to have all activities appear in the log.

Set-up. For each model generated in the scalability experiment, we measure recall and precision with respect to tree A, B or C using the PCC framework. Given the results of the scalability experiment, we include the algorithms IM, IMf, IMc, HM, IMd, IMfD, IMcD, and a baseline model allowing for any behaviour (a *flower model*).

As HM does not guarantee to return a sound model, nor provides a final marking, we obtain a final marking using the method described in [42, Appendix E]. However, even with this method we were unable to determine the languages of the models returned by HM, thus these were excluded.

Results. Figure 17 shows that for model A (40 activities) both IM and IMd rediscover the language of A (a model that has 1.0 model-precision and recall with respect to A) on a log of 10^4 traces. IMd could rediscover the language of B at 10^8 traces, IM did not succeed as the largest log it could handle (10^5 traces) did not contain enough information to rediscover the language of B. The largest log we generated for tree C, i.e. containing 10^8 traces, did not contain enough information to rediscover the language of C: IMd discovered a model with a recall of 1.0 and a model-precision of 0.97. Corresponding results have been obtained for IMf/IMfD and IMc/IMcD; see [42, Appendix H] for all details. The flower model provided the baseline for precision: it achieved recall 1.0 at 10^1 (A) and 10^2 (B) traces, and achieves a model-precision of 0.8.

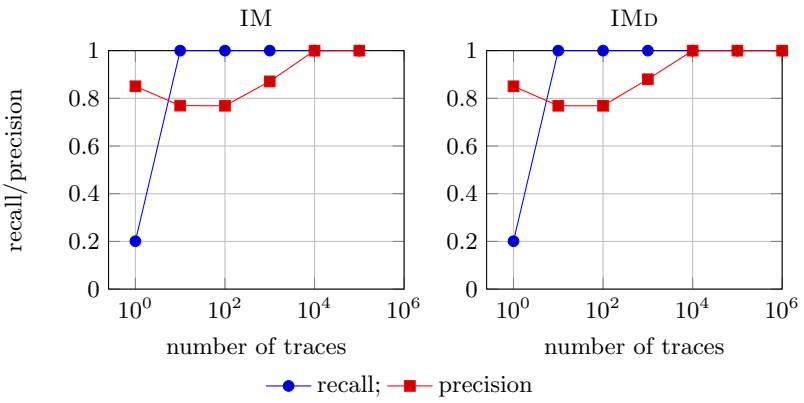


Figure 17: Incompleteness results for process tree A (40 activities).

We conclude that algorithms of the IMd framework are capable of rediscovering the original system, even in case of very large systems (trees B and C), and that these algorithms do not require larger logs than other algorithms to do so: IM and IMd rediscovered the models on logs of the same sizes; for smaller logs IMd performed slightly better than IMfD. Overall, IMd and IMfD have a similar robustness to incomplete event logs as their IM counterparts, which makes

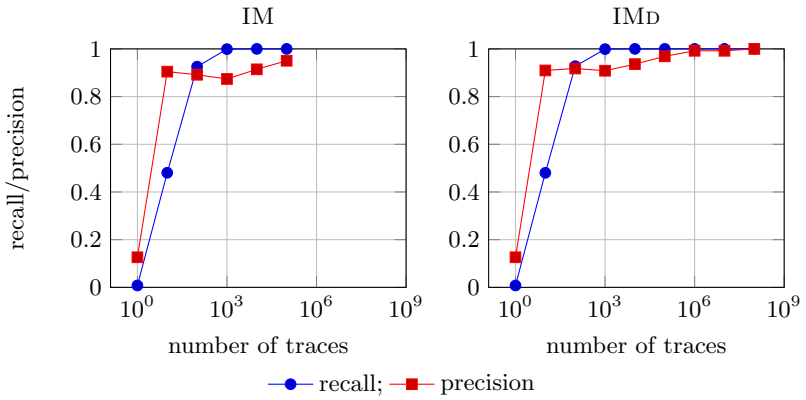


Figure 18: Incompleteness results for process tree B (1000 activities).

them more robust than other algorithms as well [44]. We discuss the results in detail in [42, Appendix G].

6.3 The Influence of Noise on Rediscoverability

To answer RQ3, we tested how noise in the event log influences rediscovery. We took the event log of 10^4 traces of tree B, as that log was well-handled by several algorithms but did not reach perfect recall and precision in the incompleteness experiment, indicating that discovery is possible but challenging. To this 10^4 traces, we add n noisy traces with some noise, for 10-fold increasing n from 1 to 10^5, i.e. the logs have 10,001 to 110,000 traces. A noisy trace is obtained from a normal trace by adding or removing a random event (both with 0.5 probability). By the representational bias of the process trees used in the generation, such a trace is guaranteed to not fit the original model. To each of these logs, we applied IM, IMF, IMD and IMFD and measured recall and precision with respect to the unchanged system B. No artificial RAM limit was enforced.

Notice that when 10^5 noisy traces are added, only 9% of the traces remains noise free. The directly-follows graph of this noisy log contains 118,262 directly-follows edges, while the graph of the model would just have 32,012. Moreover, almost all activities are observed as start activities (946) and end activities (929) in this log (vs 244/231 in the model). It is clear that without serious noise filtering, no algorithm could make any sense of this log.

Results. Figure 19 compares the 2 noise filtering algorithms IMF and IMFD on the logs of B with various noise levels. Surprisingly, IMFD performs better than IMF: IMFD achieves consistently higher precision at only slight drop in recall compared to IMF whose precision drops to 0.8, which is close to the flower model (i.e., no actual restriction of behaviour). The perfect recall obtained by IM on large lots can be explained by the fall-throughs of IMD and IM: if no cut can

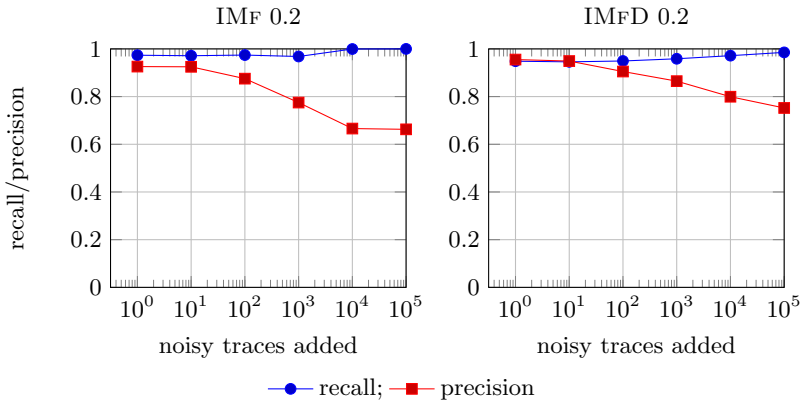


Figure 19: Algorithms applied to logs with noisy traces (tree B (1,000 activities)).

be found, a flower model is selected. For IM and IMd, we consistently observed lower precision scores for all models compared to both IMf and IMfD but a consistent fitness of 1.0 (which is easily explained by their lack of noise handling capabilities); exact numbers and more details are available in [42, Appendix H].

A manual inspection of the models returned shows that all models still give information on the overall structure of the system, while for larger parts of the model no structure could be discovered and a flower sub-model was discovered. In this limited experiment, IMfD is the clear winner: it keeps precision highest in return for a little drop in recall. We suspect that this is due to IMfD using less information than IMf and therefore the introduced noise has a larger impact (see Section 4.3). More experiments need to evaluate this hypothesis.

6.4 The Influence of Infrequent Behaviour on Rediscoverability

To answer the second part of RQ3 we investigated how infrequent behaviour, i.e. structured deviations from the system, influences rediscovery. The set-up of this experiment is similar to the noise experiment, i.e. t deviating traces are added. Each deviating trace contains one structural deviation from the model, e.g. for an $\times$, two children are executed. For more details, please refer to [42, Appendix D].

Similar to the noise experiment, the log with 10^5 added infrequent traces has a lot of wrong behaviour: without infrequent behaviour, its directly-follows graph would contain $32,012$ edges, 244 start activities and 231 end activities, but the deviating log contained $118,262$ edges, 946 start activities and 929 end activities. That means that if one would randomly pick an edge from the log, there would be only 27% chance that the chosen edge would be according to the model. Exact numbers and more details are available in [42, Appendix H].

Results. Figure 20 shows the results of IMf and IMfD on logs of process tree B with various levels of added infrequent behaviour. Similar to the noise exper-

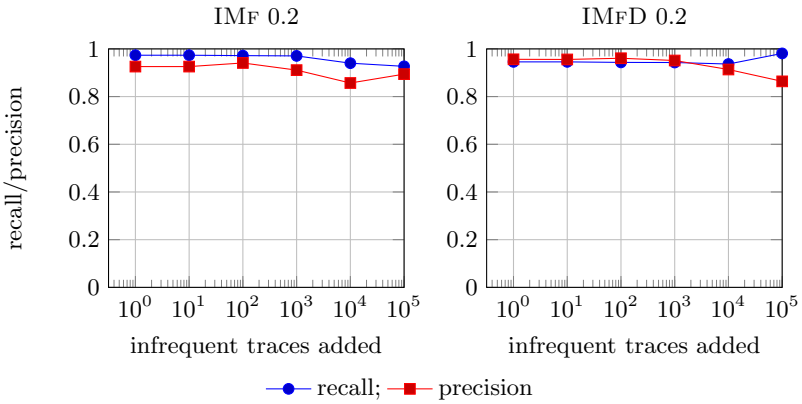


Figure 20: Infrequent behaviour results of process tree B (1000 activities).

iments, IMfD compared to IMf trades recall (0.95 vs 0.99) for log-precision (0.95 vs 0.90). Of the non-filtering versions IM and IMd, both got a recall of 0.99, IM a model-precision of around 0.90, and IMd 0.92, thus IMd performs a bit better in this experiment.

We suspect that two of the inserted types of infrequent behaviour positively influenced the results: skipping a child of a $\rightarrow$ or $\wedge$ has no troublesome impact on the directly-follows graph for the IMd framework, but log splitting will introduce a (false) empty trace for the IM framework. The IM framework algorithms must decide to ignore this empty trace in later recursions, while IMd framework algorithms simply don't see it. Altogether, IMfD performs remarkably well given event logs containing structured deviations from the system.

6.5 Real-Life Model-Log Evaluation

To test real-life performance of the new algorithms and to answer RQ4, i.e. whether the newly introduced fitness and precision measures can handle larger logs and how they compare to existing measures, we performed a fourth experiment.

Experimental Set-up. In this experiment, we take four real-life event logs. To these event logs, we apply the algorithms α, HM, IM, IMf, IMd and IMfD and analyse the resulting models manually. The algorithms CCM and MPM are not publicly available and were excluded.

Secondly, in order to evaluate the PCC framework, we apply the PCC framework and existing fitness [1] and log-precision [10] measures to the discovered models. The models by HM and α were unsound and had to be excluded (we introduced some heuristics for unsound models, but they did not help in this case. For more details, see [42, Appendix E]). Furthermore, IM and IMd do not

apply noise filtering and therefore their models are often flower models, so these were excluded as well.

To provide a baseline for log-precision, we add a flower model to the comparison: a flower model allows for all behaviour, so intuitively has the worst precision. Therefore, we scale (normalised) the log-precision according to this baseline:

$$\text{scaled log-precision} = 1 - \frac{1 - \text{log-precision of model}}{1 - \text{log-precision of flower model}}$$

Intuitively, *scaled precision* denotes the linear precision gain with respect to a flower model, i.e. 0 for the flower model itself and 1 for perfect precision. The PCC framework-fitness measure and the existing fitness measure [1] are conceptually similar, so they are not scaled.

Logs. We analysed four real-life logs. The first log (BPIC11) originates from the emergency department of a Dutch hospital [28]. BPIC11 describes a fairly unstructured process, and contains 624 activities, 1,143 traces (patients) and 150,291 events. In our classification, it is a COMPLEX and MEDIUM log. The second log (BPIC12) originates from a Dutch financial institution, and describes a mortgage application process [29]. It contains 23 activities, 13,087 traces (clients) and 164,506 events, and is therefore a MEDIUM log. BPIC12 was filtered to only contain events having the "complete" life cycle transition, i.e. "schedule" and "start" events were removed. The resulting log was included, as well as the logs for three activity-subsets, ($\text{BPIC}|_A$, $\text{BPIC}|_O$ and $\text{BPIC}|_W$) e.g. $\text{BPIC}|_A$ only contains activities prefixed by A. The third log (SL) originates from the repeated execution of software: SL was obtained by recording method calls executed by RapidMiner, using 5 operators, each corresponding to plug-ins of RapidProM. The recording was performed using the Kieker tool (see `http://kieker-monitoring.net`), and repeated 25 times with different input event logs. In total, the event log (SL) has 25 traces, 5,869,492 events and 271 activities, which makes it a LARGE. Its traces are particularly long: up to 1,151,788 events per trace. The fourth log (CS) originates from a web-site of a dot-com start-up, and represents click-stream data, i.e. every web-site visitor is a trace, and each page visited is an events [37]. CS contains 77,513 traces, 358,278 events and 3,300 activities, which makes it MORE COMPLEX and MEDIUM. As described in the introduction, much bigger event logs exist. We were able to run IMD and IMFD on LARGER and MORE COMPLEX logs, but we were unable to compute metrics or even visualise the resulting models. Therefore, we do not report on such logs in this evaluation.

Results for Process Discovery. The first step in this experiment was to apply several process discovery algorithms.

On BPIC11, IM, IMF, IMD and IMFD produced a model.

On BPIC12, all the algorithms produced a model. We illustrate the results of the experiments on BPIC12, filtered for activities starting with A ($\text{BPIC}|_A$):

Figure 21 shows the model returned by IMf; Figure 22 the model by IMfD. This illustrates the different trade-offs made between IM and IMd: these models are very similar, except that for IMf, three activities can be skipped. Operating on the directly-follows abstraction of the log, IMfD was unable to decide to make these activities skippable, which lowers fitness a bit (0.816 vs 0.995) but increases precision (1 vs 0.606).

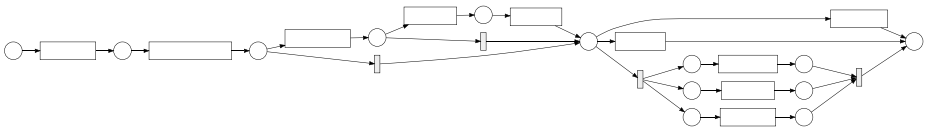


Figure 21: IMf applied to BPIC12$|_A$ (without activity names).

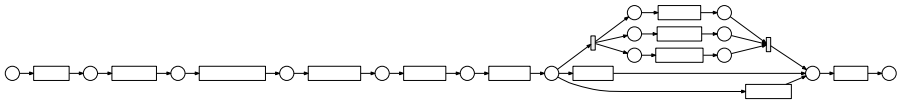
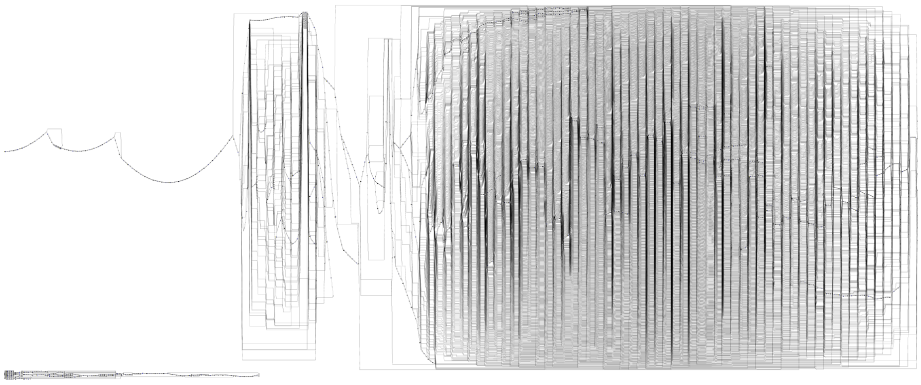


Figure 22: IMfD applied to BPIC12$|_A$ (without activity names).

On the SL log, HM, IM, IMf, IMd and IMfD produced a model, and α could not proceed beyond passing over the event log. The models discovered by HM, IMf and IMfD are shown in Figure 23 (we uploaded these models to `http://www.processmining.org/blogs/pub2015/scalable_process_discovery_and_evaluation`). The model discovered by HM has 2 unconnected parts, of which one part cannot be executed. Hence it is not a workflow model, thus not sound and, as discussed before, difficult to be analysed automatically. In the models discovered IMf and IMfD, the five RapidProM operators are easily recognisable. However, the models are too complex to be analysed in detail by hand[2]. Therefore, in further analysis steps, problematic parts of the models by IMf and IMfD could be identified, the log filtered for them, and the analysis repeated.

On the CS log, IM, IMf, IMd and IMfD produced a model. IMd and IMfD returned a model in less than 30 seconds using less than 1GB of RAM, while IM and IMf took more than an hour and used 30GB of RAM. As CS has five times more activities than SL, we could not visualise it. This illustrates that scalable process discovery is a first step in scalable process mining: the models we obtained are suitable for automatic processing, but human analysis without further visualisation techniques is very challenging.

[2] Anecdotically: the vector-images of these models were too large to be displayed by Adobe Illustrator or Adobe Acrobat.



(a) Impression of the model discovered by HM.

(b) Impression of the model discovered by IMf.

(c) Impression of the model discovered by IMfD.

Figure 23: Models discovered from a software execution log.

Results for Log-Conformance Checking. Table 4 shows the results, extended with the approximate running time of the techniques.

Fitness scores according to the PCC framework differ from the fitness scores by [1] by at most 0.05 (except for BPIC12$|_A$ IMfD). Thus, this experiment suggests that the new fitness measurement could replace the alignment-based fitness [1] metric, while being generally faster on both smaller and larger logs, though additional experiments may be required to verify this hypothesis. More importantly, the PCC framework could handle logs (BPIC11, SL, CS) that the existing measure could not handle.

Comparing the scaled precision measures, the PCC framework and the existing approach agree on the relative order of IMf and IMfD for BPIC12$|_A$ and BPIC12$|_O$, disagree on BPIC12 and are incomparable on BPIC11, SL and CS due to failure of the existing measure. For BPIC12$|_W$, IMfD performed *worse* than the flower model according to [10] but *better* according to our measure. This model, shown in Figure 24, is certainly more restrictive than a flower model, which is correctly reflected by our new precision measure. Therefore, likely the approach of [10] encounters an inaccuracy when computing the precision score. For BPIC12, precision [10] ranks IMf higher than IMfD, whereas our precision ranks IMfD higher than IMf. Inspecting the models, we found that IMf misses one activity from the log while IMfD has all activities. Apparently, our new measure penalises more for a missing activity, while the alignment-based existing measure penalises more for a missing structure.

Table 4: Log-measures compared on real-life logs.

		existing techniques				this paper (PCC framework)			
		fitness [1]	log-precision [10] measured	scaled	time	fitness	log-precision measured	scaled	time
BPIC11	IMF	out of memory				0.627	0.764	0.472	25s
	IMFD	out of memory				0.997	0.766	0.477	1m
	flower	1.000	0.002	0.000	5h	1.000	0.553	0.000	25s
BPIC12$\|_A$	IMF	0.995	0.606	0.940	$\leq$1s	0.999	0.967	0.931	$\leq$1s
	IMFD	0.816	1.000	1.000	$\leq$1s	0.700	1.000	1.000	$\leq$1s
	flower	1.000	0.227	0.000	$\leq$1s	1.000	0.520	0.000	$\leq$1s
BPIC12$\|_O$	IMF	0.991	0.508	0.351	$\leq$1s	0.981	0.809	0.407	$\leq$1s
	IMFD	0.861	0.384	0.187	$\leq$1s	0.862	0.794	0.360	$\leq$1s
	flower	1.000	0.242	0.000	$\leq$1s	1.000	0.678	0.000	$\leq$1s
BPIC12$\|_W$	IMF	0.876	0.690	0.553	$\leq$1s	0.875	0.836	0.611	$\leq$1s
	IMFD	0.914	0.300	-0.010	$\leq$1s	0.923	0.823	0.581	$\leq$1s
	flower	1.000	0.307	0.000	$\leq$1s	1.000	0.578	0.000	$\leq$1s
BPIC12	IMF	0.967	0.364	0.290	20m	0.978	0.668	0.092	$\leq$1s
	IMFD	1.000	0.189	0.095	25m	1.000	0.693	0.161	$\leq$1s
	flower	1.000	0.104	0.000	30m	1.000	0.634	0.000	$\leq$1s
SL	IMF	out of memory				0.584	0.246	-0.158	30m
	IMFD	out of memory				0.924	0.385	0.055	30m
	flower	out of memory				1.000	0.349	0.000	35m
CS	IMF	out of memory				0.999	0.580	0.023	1h
	IMFD	out of memory				0.999	0.585	0.036	6.5h
	flower	out of memory				1.000	0.570	0.000	55m

A similar effect is visible for SL: IMF achieves a lower precision than the flower model. Further analysis revealed that several activities were missing from the model by IMF. The following example illustrates the effect: let $L = \{\langle a, b\rangle\}$ be a projected log and $M = a$ a projected model. Then, technically, their conjunction is empty and hence both precision and recall are 0. This matches intuition, as they have no trace in common. This sensitivity to missing activities is inherent to language-based measuring techniques. From the model discovered by IMF, 45 activities are missing, which means that of the 36,585 pairs of activities that are considered for precision and recall, in 11,160 pairs a missing activity is involved.

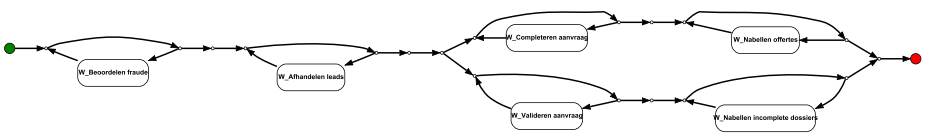


Figure 24: IMFD applied to BPIC$|_W$.

This experiment does not suggest that our new measure can directly replace the existing measures, but precision seems to be able to provide a categorisation, such as good/mediocre/bad precision, compared to the flower model.

Altogether we showed that our new fitness and precision metrics are useful to quickly assess the quality of a discovered model and decide whether to continue analyses with it or not, in particular on event logs that are too large for current techniques. In addition to simply providing an aggregated fitness and precision value, both existing and our new technique allow for more fine-grained diagnostics of *where* in the model and event log fitness and precision are lost. For instance, by looking at the subsets $a_1 \dots a_k$ of activities with a low fitness or precision score, one can identify the activities that are not accurately represented by the model, and then refine the analysis of the event log accordingly.

For most event logs, IMfD seems to perform comparably to IMf. However, please notice that by the nature of fitness and log-precision, for each event log there exists a trivial model that scores perfectly on both, i.e. the model consisting of a choice between all traces. As such a model provides neither any new information nor insight, generalisation and simplicity have to be taken into account as well. As future work, we would like to adapt generalisation metrics to be applicable to large event logs and complex processes as well.

7 Conclusion

Process discovery aims to obtain process models from event logs, while conformance checking aims to obtain information from the differences between a model and either an event log or a system-model. Currently, there is no process discovery technique that works on LARGER and MORE COMPLEX logs, i.e. containing billions of events or thousands of activities, and that guarantees both soundness and rediscoverability. Moreover, current log-conformance checking techniques cannot handle MEDIUM and COMPLEX logs, and as current process discovery evaluation techniques are based on log-conformance checking, algorithms cannot be evaluated for MEDIUM and COMPLEX logs. In this paper, we pushed the boundary on what can be done with LARGER and MORE COMPLEX logs.

For process discovery, we introduced the *Inductive Miner - directly-follows* (IMd) framework and three algorithms using it. The input of the framework is a directly-follows graph, which can be obtained from any event log in linear time, even using highly-scalable techniques such as Map-Reduce. The IM framework uses a divide-and-conquer strategy that recursively builds a process model by splitting the directly-follows graph and recursing on the sub-graphs until encountering a base case.

We showed that the memory usage of algorithms of the IMd framework is independent of the number of traces in the event log considered. In our experiments, the scalability was only limited by the logs we could generate. The IMd framework managed to handle over 70 billion events, while using only 2GB of RAM; some other techniques required the event log to be in main memory and therefore could handle at most 1 - 10 million events. Besides scalability, we also

investigated how the new algorithms compare qualitatively to existing techniques that use more knowledge, but also have higher memory requirements. The new algorithms handled systems of 10,000 activities in polynomial time, and were robust to incompleteness, noise and infrequent behaviour. Moreover, they always return sound models and suffered little loss in quality compared to multi-pass algorithms; in some cases we even observed quality improvements.

For conformance checking, we introduced the *projected conformance checking framework* (PCC framework), that is applicable to both log-model and model-model conformance checking. The PCC framework measures recall/fitness and precision, by projecting both system and system-model/log onto subsets of activities to determine their recall/fitness and precision. Using this framework, one can measure recall/fitness and precision of arbitrary models with a bounded state space of (almost) arbitrary size.

The PCC framework's model-model capabilities enable a novel way to evaluate discovery techniques that scales well and provides new insights. We applied this to test robustness of various algorithms to incompleteness, noise and infrequent behaviour. Moreover, we showed that the log-model version of the PCC framework allows to measure fitness and precision of a model with respect to an event log, even in cases where classical techniques fail, and can give detailed insights into the location of deviations in both log and model.

Altogether, we have presented the first steps of process mining workflows on very large data sets: discovering a model and assessing its quality. However, as we encountered in our evaluation, we envision further steps in the processing and visualisation of large models, such as using natural language based techniques [12]. To ease the analyses in contexts of big data, our algorithm evaluation framework could be combined with the approach in [15], by having our framework detecting the problematic sets of activities, and the approach in [15] focusing on these sub-models. For instance, it would be interesting to approximate performance measures on the model without computing alignments.

Furthermore, it would be interesting to study the influence of k on the PCC framework, both practically and theoretically. As shown in Section 5.1, there exist cases for which the language equivalence can only be guaranteed if k is at least the number of nodes minus one. However, besides for the classes for which Theorem 1 or Corollary 2 holds, there might be other classes of models for which a smaller k suffices.

References

1. van der Aalst, W., Adriansyah, A., van Dongen, B.: Replaying history on process models for conformance checking and performance analysis. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 2(2), 182–192 (2012)
2. van der Aalst, W.M.P.: Process Mining - Discovery, Conformance and Enhancement of Business Processes. Springer (2011), http://dx.doi.org/10.1007/978-3-642-19345-3

3. van der Aalst, W.M.P.: Decomposing process mining problems using passages. In: Petri Nets 2012. pp. 72–91 (2012), http://dx.doi.org/10.1007/978-3-642-31131-4_5
4. van der Aalst, W.M.P.: Decomposing Petri nets for process mining: A generic approach. Distributed and Parallel Databases 31(4), 471–507 (2013), http://dx.doi.org/10.1007/s10619-013-7127-5
5. van der Aalst, W.M.P.: Process cubes: Slicing, dicing, rolling up and drilling down event data for process mining. In: AP-BPM 2013. pp. 1–22 (2013), http://dx.doi.org/10.1007/978-3-319-02922-1_1
6. van der Aalst, W.M.P., et al.: Process mining manifesto. In: Business Process Management Workshops 2011. pp. 169–194 (2011), http://dx.doi.org/10.1007/978-3-642-28108-2_19
7. 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)
8. Adriansyah, A.: Aligning Observed and Modeled Behavior. Ph.D. thesis, Eindhoven University of Technology (2014)
9. Adriansyah, A., van Dongen, B.F., van der Aalst, W.M.P.: Conformance checking using cost-based fitness analysis. In: IEEE EDOC 2011. pp. 55–64 (2011), http://dx.doi.org/10.1109/EDOC.2011.12
10. Adriansyah, A., Munoz-Gama, J., Carmona, J., van Dongen, B.F., van der Aalst, W.M.P.: Alignment based precision checking. In: Business Process Management Workshops 2012. pp. 137–149 (2012), http://dx.doi.org/10.1007/978-3-642-36285-9_15
11. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: POPL SIGPLAN-SIGACT 2002. pp. 4–16 (2002), http://doi.acm.org/10.1145/503272.503275
12. Armas-Cervantes, A., Baldan, P., Dumas, M., García-Bañuelos, L.: Behavioral comparison of process models based on canonically reduced event structures. In: BPM 2014. pp. 267–282 (2014), http://dx.doi.org/10.1007/978-3-319-10172-9_17
13. Badouel, E.: On the α-reconstructibility of workflow nets. In: Petri Nets'12. LNCS, vol. 7347, pp. 128–147. Springer (2012)
14. Becker, M., Laue, R.: A comparative survey of business process similarity measures. Computers in Industry 63(2), 148–167 (2012), http://dx.doi.org/10.1016/j.compind.2011.11.003
15. van Beest, N.R.T.P., Dumas, M., García-Bañuelos, L., Rosa, M.L.: Log delta analysis: Interpretable differencing of business process event logs. In: BPM 2015. pp. 386–405 (2015), http://dx.doi.org/10.1007/978-3-319-23063-4_26
16. Benner-Wickner, M., Brückmann, T., Gruhn, V., Book, M.: Process mining for knowledge-intensive business processes. In: I-KNOW 2015. pp. 4:1–4:8 (2015), http://doi.acm.org/10.1145/2809563.2809580
17. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process mining based on regions of languages. Business Process Management pp. 375–383 (2007)
18. Bergenthum, R., Desel, J., Mauser, S., Lorenz, R.: Synthesis of Petri nets from term based representations of infinite partial languages. Fundam. Inform. 95(1), 187–217 (2009)
19. Buijs, J.C.A.M.: Flexible Evolutionary Algorithms for Mining Structured Process Models. Ph.D. thesis, Eindhoven University of Technology (2014)
20. Buijs, J., van Dongen, B., van der Aalst, W.: A genetic algorithm for discovering process trees. In: IEEE Congress on Evolutionary Computation. pp. 1–8 (2012)

21. Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: On the role of fitness, precision, generalization and simplicity in process discovery. In: OTM. LNCS, vol. 7565, pp. 305–322 (2012), http://dx.doi.org/10.1007/978-3-642-33606-5_19
22. Burattin, A.: PLG2: multiperspective processes randomization and simulation for online and offline settings. CoRR abs/1506.08415 (2015), http://arxiv.org/abs/1506.08415
23. Burattin, A., Sperduti, A., van der Aalst, W.M.P.: Control-flow discovery from event streams. In: IEEE Congress on Evolutionary Computation. pp. 2420–2427 (2014), http://dx.doi.org/10.1109/CEC.2014.6900341
24. Carmona, J., Solé, M.: PMLAB: an scripting environment for process mining. In: BPM Demos. CEUR-WP, vol. 1295, p. 16 (2014)
25. Cortadella, J., Kishinevsky, M., Lavagno, L., Yakovlev, A.: Deriving Petri nets from finite transition systems. Computers, IEEE Transactions on 47(8), 859–882 (1998)
26. Datta, S., Bhaduri, K., Giannella, C., Wolff, R., Kargupta, H.: Distributed data mining in peer-to-peer networks. IEEE Internet Computing 10(4), 18–26 (2006), http://doi.ieeecomputersociety.org/10.1109/MIC.2006.74
27. Dijkman, R.M., van Dongen, B.F., Dumas, M., García-Bañuelos, L., Kunze, M., Leopold, H., Mendling, J., Uba, R., Weidlich, M., Weske, M., Yan, Z.: A short survey on process model similarity. In: Seminal Contributions to Information Systems Engineering, 25 Years of CAiSE, pp. 421–427 (2013), http://dx.doi.org/10.1007/978-3-642-36926-1_34
28. van Dongen, B.: BPI Challenge 2011 Dataset (2011), http://dx.doi.org/10.4121/uuid:d9769f3d-0ab0-4fb8-803b-0d1120ffcf54
29. van Dongen, B.: BPI Challenge 2012 Dataset (2012), http://dx.doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f
30. van Dongen, B.F., Dijkman, R.M., Mendling, J.: Measuring similarity between business process models. In: Seminal Contributions to Information Systems Engineering, 25 Years of CAiSE, pp. 405–419 (2013), http://dx.doi.org/10.1007/978-3-642-36926-1_33
31. Evermann, J.: Scalable process discovery using map-reduce. In: IEEE Transactions on Services Computing. vol. to appear (2014)
32. Gabel, M., Su, Z.: Javert: fully automatic mining of general temporal properties from dynamic traces. In: ACM SIGSOFT 2008. pp. 339–349 (2008), http://doi.acm.org/10.1145/1453101.1453150
33. van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. J. ACM 43(3), 555–600 (1996), http://doi.acm.org/10.1145/233551.233556
34. Günther, C., Rozinat, A.: Disco: Discover your processes. In: BPM (Demos). pp. 40–44 (2012)
35. Hay, B., Wets, G., Vanhoof, K.: Mining navigation patterns using a sequence alignment method. Knowl. Inf. Syst. 6(2), 150–163 (2004)
36. Hwong, Y., Keiren, J.J.A., Kusters, V.J.J., Leemans, S.J.J., Willemse, T.A.C.: Formalising and analysing the control software of the compact muon solenoid experiment at the large hadron collider. Sci. Comput. Program. 78(12), 2435–2452 (2013), http://dx.doi.org/10.1016/j.scico.2012.11.009
37. Kohavi, R., Brodley, C.E., Frasca, B., Mason, L., Zheng, Z.: Kdd-cup 2000 organizers' report: Peeling the onion. SIGKDD Explorations 2(2), 86–98 (2000), http://doi.acm.org/10.1145/380995.381033

38. Kunze, M., Weidlich, M., Weske, M.: Querying process models by behavior inclusion. Software and System Modeling 14(3), 1105–1125 (2015), http://dx.doi.org/10.1007/s10270-013-0389-6
39. Leemans, M., van der Aalst, W.: Process mining in software systems. ACM/IEEE International Conference on Model Driven Engineering Languages and Systems p. to appear (2015)
40. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs - A constructive approach. In: Petri Nets 2013. pp. 311–329 (2013), http://dx.doi.org/10.1007/978-3-642-38697-8_17
41. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Scalable process discovery with guarantees. In: BPMDS 2015. pp. 85–101 (2015), http://dx.doi.org/10.1007/978-3-319-19237-6_6
42. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Scalable process discovery and conformance checking. BPM Center Report BPM-16-03, BPMcenter.org (2016)
43. Leemans, S., Fahland, D., van der Aalst, W.: Discovering block-structured process models from event logs containing infrequent behaviour. In: Business Process Management Workshops. pp. 66–78 (2013)
44. Leemans, S., Fahland, D., van der Aalst, W.: Discovering block-structured process models from incomplete event logs. In: Petri nets 2014. vol. 8489, pp. 91–110 (2014), http://dx.doi.org/10.1007/978-3-319-07734-5_6
45. Leemans, S., Fahland, D., van der Aalst, W.: Exploring processes and deviations. In: Business Process Management Workshops. p. to appear (2014)
46. Liesaputra, V., Yongchareon, S., Chaisiri, S.: Efficient process model discovery using maximal pattern mining. In: BPM 2015. pp. 441–456 (2015), http://dx.doi.org/10.1007/978-3-319-23063-4_29
47. Linz, P.: An introduction to formal languages and automata. Jones & Bartlett Learning (2011)
48. Lu, X., Fahland, D., van den Biggelaar, F.J., van der Aalst, W.M.: Label refinement for handling duplicated tasks in process discovery. In: BPM. p. submitted (2016)
49. Møller, A.: dk.brics.automaton – finite-state automata and regular expressions for Java (2010), http://www.brics.dk/automaton/
50. Munoz-Gama, J., Carmona, J., van der Aalst, W.M.P.: Single-entry single-exit decomposed conformance checking. Inf. Syst. 46, 102–122 (2014), http://dx.doi.org/10.1016/j.is.2014.04.003
51. Pradel, M., Gross, T.R.: Automatic generation of object usage specifications from large method traces. In: ASE 2009. pp. 371–382. IEEE Computer Society (2009), http://dx.doi.org/10.1109/ASE.2009.60
52. Redlich, D., Molka, T., Gilani, W., Blair, G.S., Rashid, A.: Constructs competition miner: Process control-flow discovery of bp-domain constructs. In: BPM 2014. LNCS, vol. 8659, pp. 134–150 (2014), http://dx.doi.org/10.1007/978-3-319-10172-9_9
53. Redlich, D., Molka, T., Gilani, W., Blair, G.S., Rashid, A.: Scalable dynamic business process discovery with the constructs competition miner. In: SIMPDA 2014. CEUR-WP, vol. 1293, pp. 91–107 (2014)
54. Rozinat, A., van der Aalst, W.M.P.: Conformance checking of processes based on monitoring real behavior. Inf. Syst. 33(1), 64–95 (2008), http://dx.doi.org/10.1016/j.is.2007.07.001
55. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and more focused control-flow analysis for business process models through SESE decomposition. In: ICSOC 2007. pp. 43–55 (2007), http://dx.doi.org/10.1007/978-3-540-74974-5_4

56. Weerdt, J.D., Backer, M.D., Vanthienen, J., Baesens, B.: A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs. Inf. Syst. 37(7), 654–676 (2012)
57. Weidlich, M., van der Werf, J.: On profiles and footprints - relational semantics for Petri nets. In: Petri Nets. pp. 148–167 (2012)
58. Weidlich, M., Polyvyanyy, A., Mendling, J., Weske, M.: Causal behavioural profiles - efficient computation, applications, and evaluation. Fundam. Inform. 113(3-4), 399–435 (2011)
59. Weijters, A., van der Aalst, W., de Medeiros, A.: Process mining with the heuristics miner-algorithm. BETA Working Paper series 166, Eindhoven University of Technology (2006)
60. Weijters, A., Ribeiro, J.: Flexible Heuristics Miner. In: CIDM. pp. 310–317 (2011)
61. Wen, L., van der Aalst, W., Wang, J., Sun, J.: Mining process models with non-free-choice constructs. Data Mining and Knowledge Discovery 15(2), 145–180 (2007)
62. Wen, L., Wang, J., Sun, J.: Mining invisible tasks from event logs. Advances in Data and Web Management pp. 358–365 (2007)
63. van der Werf, J., van Dongen, B., Hurkens, C., Serebrenik, A.: Process discovery using integer linear programming. Fundam. Inform. 94(3-4), 387–412 (2009)
64. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: mining temporal API rules from imperfect traces. In: ICSE 2006. pp. 282–291 (2006), http://doi.acm.org/10.1145/1134325
65. Zha, H., Wang, J., Wen, L., Wang, C., Sun, J.: A workflow net similarity measure based on transition adjacency relations. Computers in Industry 61(5), 463–471 (2010), http://dx.doi.org/10.1016/j.compind.2010.01.001