Author/s:
Augusto, A;Dumas, M;La Rosa, M

Title:
Automated Discovery of Process Models with True Concurrency and Inclusive Choices

Date:
2021-01-01

Citation:
Augusto, A., Dumas, M. & La Rosa, M. (2021). Automated Discovery of Process Models with True Concurrency and Inclusive Choices. Leemans, S (Ed.) Leopold, H (Ed.) Lecture Notes in Business Information Processing, 406 LNBIP, pp.43-56. SPRINGER INTERNATIONAL PUBLISHING AG. https://doi.org/10.1007/978-3-030-72693-5_4.

Persistent Link:
https://hdl.handle.net/11343/258885

Automated Discovery of Process Models with True Concurrency and Inclusive Choices

Adriano Augusto[1], Marlon Dumas[2], and Marcello La Rosa[1]

[1] University of Melbourne, Australia
{a.augusto, marcello.larosa}@unimelb.edu.au
[2] University of Tartu, Estonia
marlon.dumas@ut.ee

Abstract. Enterprise information systems allow companies to maintain detailed records of their business process executions. These records can be extracted in the form of event logs, which capture the execution of activities across multiple instances of a business process. Event logs may be used to analyze business processes at a fine level of detail using process mining techniques. Among other things, process mining techniques allow us to discover a process model from an event log – an operation known as automated process discovery. Despite a rich body of research in the field, existing automated process discovery techniques do not fully capture the concurrency inherent in a business process. Specifically, the bulk of these techniques treat two activities A and B as concurrent if sometimes A completes before B and other times B completes before A. Typically though, activities in a business process are executed in a true concurrency setting, meaning that two or more activity executions overlap temporally. This paper addresses this gap by presenting a refined version of an automated process discovery technique, namely Split Miner, that discovers true concurrency relations from event logs containing start and end timestamps for each activity. The proposed technique is also able to differentiate between exclusive and inclusive choices. We evaluate the proposed technique relative to existing baselines using 11 real-life logs drawn from different industries.

1 Introduction

Enterprise information systems, such as Enterprise Resource Planning (ERP) systems, maintain detailed records of each execution of the business processes they support. These records can be extracted in the form of event logs. An event log is a set of event records capturing the execution of activities across a set of instances of a process.

Process mining techniques allow us to exploit event logs in order to analyze business processes at a fine level of detail. Among other things, process mining techniques allow us to discover a process model from an event log – an operation known as *automated process discovery*. Despite a rich body of research in the field, existing automated process discovery techniques do not fully capture the concurrency inherent in business processes. Indeed, the bulk of automated process discovery techniques operate under an interleaved concurrency model – a model of concurrency where two events are concurrent if they occur in either order. Specifically, existing techniques treat two activities A and B as concurrent if sometimes A completes before B and other times B completes

before A. The interleaved concurrency model is suitable in systems where actions are atomic. However, in a business process, activities have a duration and the execution of two or more activities may overlap temporally. In other words, business processes contain *true concurrency*. The failure of existing automated process discovery techniques to take into account this true concurrency leads them to miss certain concurrency relations. For example, when an activity A always completes before activity B (because B takes longer) even though A and B overlap, existing techniques treat A and B as sequential. If A is then followed by C and C usually completes after B (but overlaps with it), they conclude that A, B and C are sequential, thus missing the observed concurrency.

This paper addresses this gap by presenting a refined version of an automated process discovery algorithm, namely Split Miner [6], capable of discovering true concurrency relations from event logs that record both the start and end timestamps of activity executions. The proposed technique, namely Split Miner 2.0, is also able to differentiate between exclusive and inclusive choices. The paper reports on an empirical evaluation that compares Split Miner 2.0 against existing baselines in terms of accuracy and model complexity measures.

The rest of the paper is structured as follows. Section 2 briefly reviews existing automated process discovery techniques. Section 3 introduces the approach to exploit true concurrency for automated process discovery. Section 4 presents the empirical evaluation while Section 5 summarizes the findings and further possible extensions.

2 Background and Related Work

An *event log* records information about a set of executions of a business processes (a.k.a. cases). Concretely, an event log is a chronological sequence of events, each one capturing a state change in the execution of an activity. As a minimum, each event in a log has three attributes: the identifier of the process execution (a.k.a. *case ID*); the label (i.e. the process activity the event refers to); and the timestamp (e.g. 10/07/2020 10.43). Optionally, an event may have other attributes such as the resource who triggered the event, their department, etc. In this paper, we require that at least one fourth attribute is attached to each event, namely the *life-cycle transition*. For a given event, this attribute indicates what state-change the referenced activity has undergone. The life-cycle of an activity captures all the states in an activity execution and their possible transitions. In general, one could observe very complex life-cycles, including states such as created, assigned, started, suspended, etc. In this paper, we adopt a simple life-cycle model wherein an activity execution can be in one of two states: *start* (i.e. the activity execution started); and *end* (i.e the activity execution ended).

Event logs can be exploited for different types of analysis including conformance checking, process performance mining, and automated process discovery[16]. In this paper, we focus on the latter. The goal of automated process discovery is to discover a process model (such as the one in Figure 1) by analysing an event log such as the one in Table 1 (the latter is just an extract and not a full log).

The quality of an automatically discovered process model is traditionally assessed over four dimensions: fitness – the amount of process behaviour recorded in the event log that can be replayed by the process model; precision – the amount of behaviour captured by the process model that can be found in the event log; generalization – the

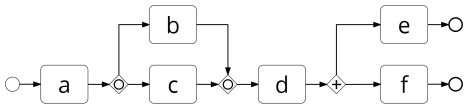


Fig. 1: Process model example.

Case-ID	Activity	Life-cycle	Timestamp
1	a	start	2020-07-08 10.03
2	a	start	2020-07-08 10.42
1	a	end	2020-07-08 10.57
2	a	end	2020-07-08 11.21
1	b	start	2020-07-08 13.29
1	c	start	2020-07-08 14.13
2	b	start	2020-07-08 15.22
2	b	end	2020-07-09 10.24
1	b	end	2020-07-09 10.37
2	d	start	2020-07-09 11.13
2	d	end	2020-07-09 12.28
1	c	end	2020-07-09 12.53

Table 1: Event log example.

amount of behaviour captured by the process model that even not being observed in the event log is likely to belong to the original process; and simplicity – quantifying how difficult is to understand the process model. Furthermore, a process model should be sound. The notion of *soundness* has been defined on Workflow nets [17] as a correctness criterion, and is also applicable to BPMN models. Formulated on BPMN models, soundness encompasses three properties: i) every process instance eventually reaches the end event (no deadlocks); ii) no end event is reached more than once during a process execution (proper completion); iii) each process activity is triggered in at least one process execution (no dead activities).

A recent literature review of automated process discovery algorithms [5] showed that only few algorithms stand out for accuracy and performance among those outputting procedural process models. Specifically, Inductive Miner (IM) [10], Evolutionary Tree Miner (ETM) [7], and Split Miner (SM) [6]. IM and ETM are known to discover process models that are either highly fitting or precise, discovering simple, block-structured and sound process models, while SM focuses on maximizing both fitness and precision at the cost of simplicity, structuredness, and in rare cases compromising the soundness of the process models [5, 6]. However, of these three automated process discovery algorithms, only IM provides a variant that takes into account the activities' life-cycle when discovering a process model. IM life-cycle variant [11] analyses the activities' life-cycles to distinguish between concurrency and interleaving relations.

Past studies that investigated the problem of discovering control-flow relations between activities by leveraging life-cycle information or execution times include: (1) a simple algorithm [13] for discovering block-structured process models from complete and noise-free event logs; (2) an extension of the α-algorithm, i.e. the β algorithm [19]; (3) an extension of Heuristics Miner [8]; and (4) the work of Senderovich et al. which explores process performance modelling via temporal network representation [14]. The first one is limited to noise-free log. The second and third are based on underlying algorithms that produce unsound and inaccurate models when applied to real-life event logs, as shown in [5]. The fourth approach is not geared to discovering process models but rather targets the problem of performance mining.

In this paper, we extend the SM algorithm, which has been shown to produce accurate and (generally) sound process models over real-life logs. Figure 2 shows an overview of how SM discovers a process model from an event log. Given an input event log, SM operates over five steps: i) discover the directly-follows graph (DFG) and loops from the event log; ii) analyse the DFG for discovering concurrency rela-

tions; iii) filter the DFG by removing the infrequent behaviour; iv) discover the split gateways; v) discover the join gateways. Each step is a standalone operation based on tailored algorithms [6], such a modular approach allows the replacement of any step with alternative methods. In this paper, we show how we updated the first, second, and fifth steps to discover true concurrency and inclusive choices, and reduce the chances of producing unsound process models via heuristics.

Fig. 2: Overview of the Split Miner approach [6].

3 Approach

In this section, we describe how we redesigned the first two steps of the Split Miner original approach [6] and integrated in the last step two heuristics to repair models that are unsound due to improper completion and identify inclusive relations between activities, enabling the discovery of OR-splits.

3.1 Refined Directly-follows Graph Discovery

Given an event log, the first step performed by Split Miner is to sequentially read the events and build the directly-follows graph (DFG). Although this operation is straightforward, its output strictly depends on how the event log and the DFG are defined. Definitions 1, 2, and 3 capture the notion of DFG used in the original Split Miner.

Definition 1. *[Event Log as in [6]] Given a set of process activity labels $\mathscr{A}$, an* event log $\mathscr{L}$ *is a multiset of traces, where a trace $t \in \mathscr{L}$ is a sequence of activity labels $t = \langle a_1, a_2, \ldots, a_k \rangle$, with $a_i \in \mathscr{A}, 1 \leq i \leq k$. In addition, we use the notation $a \in \mathscr{A}$ to refer an activity a that belongs to a generic trace $t \in \mathscr{L}$.* [1]

Definition 2. *[Directly-Follows Relation as in [6]] Given an event log $\mathscr{L}$ and two process activities $a_x, a_y \in \mathscr{A}$, we say that activity a_y directly-follows activity a_x, with notation $a_x \rightsquigarrow a_y$, if and only if (iff)* $\exists \langle a_1, a_2, \ldots, a_k \rangle \in \mathscr{L} \mid a_i = a_x \wedge a_j = a_y \wedge j = i+1 \wedge 0 < i < n$.

Definition 3. *[Directly-Follows Graph as in [6]] Given an event log $\mathscr{L}$, its* Directly-Follows Graph (DFG) *is a directed graph $\mathscr{G} = (N, E)$, where N is the non-empty set of nodes, where each node represents a unique activity $a \in \mathscr{L}$ and there exists a bijective function $\lambda : N \mapsto \mathscr{A}$ such that $\lambda(n)$ retrieves the activity n refers to; and E is the set of edges capturing the directly-follows relations of the activities observed in $\mathscr{L}$, $E = \{(n,m) \in N \times N \mid \lambda(n) \rightsquigarrow \lambda(m)\}$.*

To capture the activities' lifecycle information, we refine the concept of event log.

Definition 4. *[Refined Event Log] Given a set of events $\mathscr{E}$, a* refined event log $\mathscr{L}_\rho$ *is a multiset of traces, where a trace $t \in \mathscr{L}_\rho$ is a sequence of events $t = \langle e_1, e_2, \ldots, e_k \rangle$, with $e_i \in \mathscr{E}, 1 \leq i \leq k$. Each event $e \in \mathscr{E}_\rho$ is a tuple $e = (l, p, t)$, where $l \in \mathscr{A}$ is the process activity the event refers to, retrieved with the notation e^l; $p \in \{start, end\}$ is the state of the life-cycle of activity l, retrieved with the notation e^p; and t is the timestamp of the event, retrieved with the notation e^t.*

[1] For simplicity, we use the term activity to refer to its label.

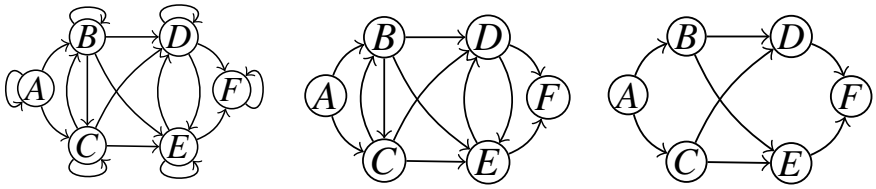
While redefining the event log to capture the activities' life-cycle information is intuitive and follows from its original definition [16], the same does not apply for the DFG. Indeed, more than one approach could be used to generate a DFG from a refined event log. The simplest approach would be to disregard all the events of a specific state of an activity life-cycle, for example, we could remove from $\mathscr{L}_\rho$ all the events $e \in \mathscr{L}_\rho \mid e^p = start$ or all the events $e \in \mathscr{L}_\rho \mid e^p = end$. Then, the refined event log would turn into an event log (Definition 1) and the DFG would be constructed according to Definition 3, but this would be equivalent to discarding the activities' lifecycle information.

An alternative approach was proposed by Leemans et al. [11] and incorporated into a variant of the Inductive Miner that takes into account lifecycle transitions, herein called Inductive Miner Lifecycle (IM-lc). According to [11], an activity a_y directly-follows an activity a_x if any of the life-cycle states of activity a_y is observed after any of the life-cycle states of activity a_x in the same trace and between the two observations no activity completes the execution of its full life-cycle (see Definition 5).

Definition 5. *[Directly-Follows Relation as in [11]] Given a refined event log $\mathscr{L}_\rho$ and two process activities $a_x, a_y \in \mathscr{A}$, the relation $a_x \rightsquigarrow a_y$ holds iff $\exists\ \langle e_1, e_2, \ldots, e_k \rangle \in \mathscr{L}_\rho \mid {e^l}_i = a_x \wedge {e^l}_j = a_y \wedge i < j \wedge \nexists n, m \in\ ]\, i, j\, [\ \mid\ n < m\ \wedge {e^p}_n = start \wedge {e^p}_m = end \wedge {e^l}_n = {e^l}_m$.*

According to Definition 5, a directly-follows relation would hold between two activities whose life-cycles overlap (i.e. the start-state of an activity is observed between the start-state and the end-state of another activity). While this is important and useful for IM-lc to discover concurrency relations [10], it would not be beneficial for Split Miner, since Split Miner requires to remove the directly-follows relations between activities that are considered concurrent [6]. Consequently, we are interested in discarding directly-follows relations of activities whose life-cycles overlap. We redefine the directly-follows relation of activities observed in a refined event log as follows. An activity a_y directly-follows an activity a_x if the start-state of the life-cycle of activity a_y is observed after the end-state of the life-cycle of activity a_x and no end-state of other activities are observed in between (see Definition 6).

Definition 6. *[Directly-Follows Relation] Given a refined event log $\mathscr{L}_\rho$ and two process activities $a_x, a_y \in \mathscr{A}$, the relation $a_x \rightsquigarrow_r a_y$ holds iff $\exists\ \langle e_1, e_2, \ldots, e_k \rangle \in \mathscr{L}_\rho \mid {e^l}_i = a_x \wedge {e^l}_j = a_y \wedge {e^p}_i = end \wedge {e^l}_j = start \wedge 1 \leq i < j \leq k \wedge \nexists n \in\ ]\, i, j\, [\ \mid {e^p}_n = end$.*



(a) Split Miner [6]. (b) Inductive Miner LC [11]. (c) Our approach.

Fig. 3: Examples of discovered DFGs by applying Definition 2, 5, and 6 (left to right).

The new version of Split Miner we propose in this paper relies on Definition 6. Depending on the definition of directly-follows relation that one adopts when generating the DFG, one may discover very different DFGs. As an example, let us consider the following refined event log (captured as a collection of traces, where each event is represented as the activity it refers to – including its life-cycle state as subscript, *s*

standing for *start* and *e* standing for *end*): $\mathscr{L}_{\rho_x} =$
$\{\langle A_s,A_e,B_s,C_s,C_e,B_e,E_s,D_s,D_e,E_e,F_s,F_e\rangle, \langle A_s,A_e,B_s,C_s,B_e,C_e,E_s,D_s,E_e,D_e,F_s,F_e\rangle,$
$\langle A_s,A_e,C_s,B_s,B_e,C_e,D_s,E_s,D_e,E_e,F_s,F_e\rangle, \langle A_s,A_e,C_s,B_s,C_e,B_e,D_s,E_s,E_e,D_e,F_s,F_e\rangle\}$;
Figure 3 shows the DFGs discovered from the $\mathscr{L}_{\rho_x}$ by applying Definition 2 (original Split Miner approach), Definition 5 (Inductive Miner life-cycle approach), and Definition 6 (this paper approach).

3.2 Refined Concurrency Discovery

The second step of the original Split Miner that we redesigned is the concurrency discovery. Split Miner relies on a simple heuristic to discover concurrency, precisely, given a DFG and two activities $A,B \in \mathscr{A}$ such that neither A nor B is a self-loop, A and B are assumed concurrent iff three conditions are true: A directly-follows B and B directly-follows A (Relation 1); A and B do not form a short-loop (Relations 2 and 3); the frequency of the two directly-follows relations $A \rightsquigarrow B$ and $B \rightsquigarrow A$ is similar (Relation 2).[2]

$$A \rightsquigarrow B \wedge B \rightsquigarrow A \tag{1}$$

$$\nexists \langle a_1,a_2,\ldots,a_k\rangle \in \mathscr{L} \mid a_i = A \wedge a_{i+1} = B \wedge a_{i+2} = A \ \wedge \ i \in [1,k-2] \tag{2}$$

$$\nexists \langle a_1,a_2,\ldots,a_k\rangle \in \mathscr{L} \mid a_i = B \wedge a_{i+1} = A \wedge a_{i+2} = B \ \wedge \ i \in [1,k-2] \tag{3}$$

$$\frac{||A \rightsquigarrow B| - |B \rightsquigarrow A||}{|A \rightsquigarrow B| + |B \rightsquigarrow A|} < \varepsilon \quad (\varepsilon \in [0,1]) \tag{4}$$

The simplicity of the concurrency oracle of Split Miner derives from the simplicity of the input event log (see Definition 1). However, when receiving as input a refined event log (Definition 4), it is possible to identify true concurrency by focusing on activities whose life-cycles overlap and are hence truly executed concurrently (e.g. by different process resources). Consequently, we redefine the concurrency discovery oracle as follows. Given two activities $A,B \in \mathscr{A}$ and a refined event log $\mathscr{L}_\rho$, we say A and B are concurrent if the following relation holds:

$$2 \cdot \frac{|A \asymp B|}{|A| + |B|} \geq \varepsilon \quad (\varepsilon \in [0,1]) \tag{5}$$

where $|A \asymp B|$ is the total number of observations of overlapping life-cycles of A and B in $\mathscr{L}_\rho$; $|A|$ and $|B|$ are respectively the total number of complete life-cycle[3] observations of activity A and activity B in $\mathscr{L}_\rho$; and ε is an arbitrary variable (given as input parameter) defining the minimum percentage of times that the two activities' life-cycles are required to overlap to assume the two activities concurrent. In particular, when $\varepsilon = 1$ our notion of concurrency is equivalent to the notion of *strong simultaneousness* defined by Van der Werf et al. [18] as well as Allen's interval relations [3] of *overlaps*, *contains*, *starts*, and *is finished by*. While for any other value of $\varepsilon > 0$ it is equivalent to a parametrized notion of *weak simultaneousness* [18]. Given that real-life event logs often contain noise and infrequent process behaviour, requiring $\varepsilon = 1$ would be very restrictive and may lead to the discovery of no concurrent activities.

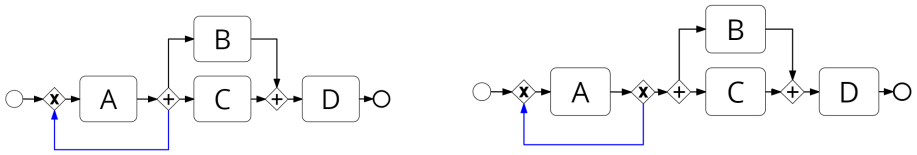
Although both our approach and IM-lc infer concurrency relations between activities from the observation of overlapping life-cycles, we rely on an heuristic before

[2] The frequency of a directly-follows relation is the number of times the relation is observed.
[3] E.g. including start and end states.

validating the concurrency relations (i.e. Equation 5) – in-line with the original Split Miner; while IM-lc assumes the information contained in the log to be valid a priori (this is mitigated by another extension of IM-lc that embeds a filtering technique [11]).

3.3 Heuristic Improvement

Although Split Miner guarantees to discover sound acyclic process models and *deadlock-free* cyclic process models with *no dead activities*, for cyclic process models it does not guarantee *proper completion*. However, it is possible to reduce the chances to discover process models exhibiting improper completion by applying the following heuristic: for each AND-split gateway in a process model with an outgoing edge that is a loop-edge (leading to a topologically deeper node of the process model), we create a preceding XOR-split gateway and set this latter as source of the loop-edge. Figure 4 intuitively show how the heuristic operates, the loop-edge is highlighted in blue and, in general, activities could be present in the loop-edge.



(a) Model with improper completion. (b) Model after applying heuristic.

Fig. 4: Heuristic removal of improper completion generated by loops.

Lastly, we integrated an heuristic to discern between concurrency and inclusive relations, in other words identifying when an AND-split gateway is a candidate OR-split gateway. This second heuristic operates as follows. For each AND-split gateway in a process model, we consider all the successor activities and we check pairwise whether there exist traces where the pair of activities are mutually exclusive (i.e. one of the two activities is executed but not the other). Then, if the majority of the pairs of activities are both mutually exclusive and concurrent in different traces,[4] we turn the AND-split gateway into an OR-split gateway and we update accordingly the OR-join gateway. As an example, let us consider the model in Figure 5a and the event log $\mathscr{L}_{\rho_y} = \{\langle A_s,A_e,B_s,C_s,D_s,B_e,D_e,C_e,E_s,E_e\rangle^3, \langle A_s,A_e,C_s,D_s,C_e,D_e,E_s,E_e\rangle^2,$ $\langle A_s,A_e,B_s,D_s,D_e,B_e,E_s,E_e\rangle, \}$; B and C are observed three times concurrently and three times are mutually exclusive, B and D are observed four times concurrently and two times mutually exclusive, C and D are observed five times concurrently and one mutually exclusive. Given that two pairs of activities out of three (B,D and B,C) are eligible for inclusiveness, we turn the AND gateways into OR gateways (Figure 5b).

4 Evaluation

In this section, we present an empirical evaluation that compares Split Miner 2.0 ($SM_{2.0}$) with three state-of-the-art automated process discovery algorithms: the original Split Miner [6] (SM), the Inductive Miner Lifecycle (IM-lc) [10] including its infrequent behaviour filter [11], and the most recent version of IM, namely IMfa [9].

[4] With at least one observation of mutual exclusiveness every two observations of concurrency or vice-versa.

(a) Before. (b) After.

Fig. 5: Heuristic identification of OR-split gateways.

4.1 Dataset and Setup

As testing dataset, we selected eleven real-life event logs (L1-L11) containing activity lifecycle information. The logs were sourced from companies operating in different fields (e.g. insurance, manufacturing, banking) and geographic areas (i.e. Europe and Australia). Given that these logs are not publicly available, we added a publicly available simulated event log known as the "Repair example" (R-Log),[5] which also contains activity lifecycle information. We did not include the BPIC12 and BPIC17 logs simply because the former does not have any overlapping lifecycle, and for the latter both SM and $SM_{2.0}$ produced the same model, which was analysed in previous studies [4, 6, 5].

Table 2 displays the characteristics of the event logs, highlighting their variety, with logs containing short to long traces (length 2 to 1,230), a wide range of distinct traces (0.28% to 96.55%) and distinct events (6 to 80), as well as notable differences in the total number of traces (37 to 70,512) and events (1,156 to 830,522). The lifecycle information for each activity recorded in these event logs was complete, i.e. the start and end events were recorded for each activity.

From each log, we discovered a process model with $SM_{2.0}$, SM, and IM-lc, and compared the quality of the discovered models over three quality measures: fitness, precision, and simplicity. Several methods have been proposed for measuring fitness and precision of an automatically discovered process model [15]. In this paper we use two methods, the one proposed by Adriansyah et al. [1, 2] (alignment-based accuracy) and the one proposed by Augusto et al. [4] (Markovian accuracy). As proxy for simplicity we use the following three metrics [12]: *Size* – the total number of nodes of a process model; *Control-flow complexity* (CFC) – the amount of branching induced by the split gateways in the process model; *Structuredness* – the percentage of nodes located inside a single-entry single-exit fragment of the process model.

We implemented $SM_{2.0}$ as a Java command-line application,[6] and we ran the experiments on an Intel Core i7-8565U@1.80GHz with 32GB RAM running Windows 10 Pro (64-bit) and JVM 8 with 14GB of allocated RAM (10GB Stack and 4GB Heap). All the discovery algorithms (SM, $SM_{2.0}$, and IM-lc) were executed using their default input parameters, and we set a timeout of 30 minutes for each algorithm execution and for each measurement.

[5] `http://www.promtools.org/prom6/downloads/example-logs.zip`
[6] Available as "Split Miner 2.0" at `http://apromore.org/platform/tools`

Event Log	Total Traces	Distinct Traces	Total Events	Distinct Events	Trace Length		
					MIN	AVG	MAX
L1	28,504	2.64%	443,862	23	4	15	1230
L2	3,885	9.11%	15,096	6	2	3	60
L3	954	10.80%	13,740	18	6	14	46
L4	37	86.49%	1,156	18	22	31	36
L5	146	78.08%	3,764	18	2	25	84
L6	551	96.55%	19,174	80	2	34	126
L7	70,512	0.28%	830,522	8	4	11	40
L8	9,906	2.19%	9,906	26	6	44	354
L9	1,182	92.81%	46,282	9	12	39	276
L10	608	11.51%	18,238	21	4	2	88
L11	1,214	20.18%	11,226	12	4	9	58
R-Log	1,104	5.53%	15,468	8	6	14	30

Table 2: Descriptive statistics of the logs.

4.2 Results

Table 3 reports the fitness, precision, and simplicity measurements. Due to space limits, the table does not show the measurements for IMfa because they were either equal or worse than those for IM-lc, with the exception of those obtained on L9 (which reported a slight improvement).

We can observe that $SM_{2.0}$ is less prone to discovering unsound models than SM, with the latter discovering an unsound model every three and the former only discovering sound models. This achievement reflects the effectiveness of our heuristics for removing improper completion.

In terms of accuracy, the results obtained with the alignment-based accuracy and the Markovian accuracy are partially consistent in line with previous findings [4]. In fact, the two measuring methods agree on the best models in terms of fitness, precision, and F-score, respectively 100%, 66%, and 75% of the times.

As for fitness, IM-lc outperforms both SM and $SM_{2.0}$ as expected [5]. In terms of precision and F-score $SM_{2.0}$ and SM achieve the highest scores, with $SM_{2.0}$ performing better than SM, most of the times discovering more precise and fitting process models ultimately achieving a higher F-score. In fact, $SM_{2.0}$ accuracy scores are either higher than or equal to those of SM, the latter outperforming the former in fitness or precision only two times according to the alignment-based accuracy, and only three times according to the Markovian accuracy. Compared to IM-lc, $SM_{2.0}$ discovers eleven times more precise process models, indipendently of the measurement method.

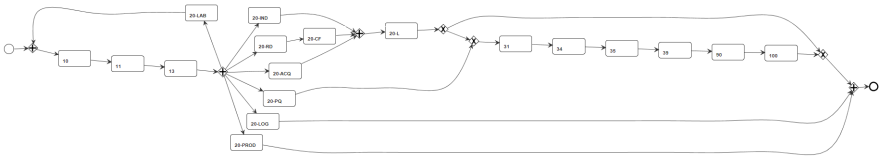
As for simplicity, $SM_{2.0}$ stands out by producing smaller models than those discovered by both SM and IM-lc (9 times out of 12) and with a lower CFC (10 times out of 12). However, $SM_{2.0}$ and SM cannot systematically produce fully-structured process models as opposed to IM-lc which achieves this by design. Lastly, the execution times of IM-lc, SM, and $SM_{2.0}$ are negligible: all the process models were discovered within a minute (except for log L7, where IM-lc timed out).

Figure 6 shows two qualitative examples of the improvements yielded by $SM_{2.0}$. Considering the models from the L6 log (Figures 6a and 6b), $SM_{2.0}$ discovered the inclusive-OR relations between several activities of the process and removed the improper completion, while SM produced an unsound model. In the specific case of the L6 log, we also had the chance to validate the discovered model with the process analysts of the organization this log was extracted from, who confirmed that the activities were indeed in an inclusive-OR relation. Considering the models from the R-log (Fig-

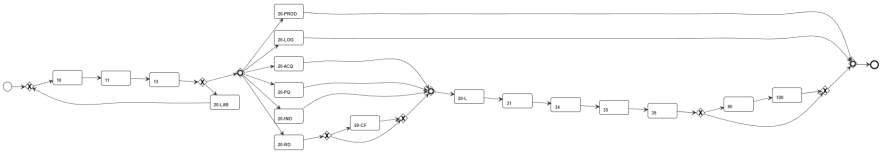
Event Log	Discovery Approach	Alignment Accuracy [1, 2]			Markovian Accuracy [4]			Simplicity		
		Fitness	Precision	F-score	Fitness	Precision	F-score	Size	CFC	Struct.
L1	IM-lc	0.88	0.78	0.83	0.82	0.15	0.25	**40**	26	**1.00**
	SM	**0.98**	0.94	**0.96**	**0.96**	**0.44**	**0.60**	47	32	0.47
	$SM_{2.0}$	0.83	**0.97**	0.90	0.44	0.34	0.38	45	**25**	0.56
L2	IM-lc	0.87	0.44	0.59	0.53	0.14	0.22	20	11	**1.00**
	SM	**0.92**	**1.00**	**0.96**	**0.69**	**0.88**	**0.77**	**14**	**6**	**1.00**
	$SM_{2.0}$	**0.92**	**1.00**	**0.96**	**0.69**	**0.88**	**0.77**	**14**	**6**	**1.00**
L3	IM-lc	**0.98**	0.71	0.82	**0.88**	0.08	0.14	49	27	**1.00**
	SM	0.96	0.97	**0.96**	0.72	**0.41**	**0.52**	36	16	0.58
	$SM_{2.0}$	0.93	**0.99**	**0.96**	0.40	0.07	0.12	**31**	**10**	0.77
L4	IM-lc	**0.98**	0.41	0.57	**1.00**	0.06	0.12	35	12	**1.00**
	SM	0.84	**1.00**	**0.91**	0.45	**0.79**	**0.57**	26	6	0.46
	$SM_{2.0}$	0.94	0.66	0.78	0.93	0.08	0.15	**25**	**3**	**1.00**
L5	IM-lc	**0.83**	0.53	0.65	**0.90**	0.17	0.29	33	12	**1.00**
	SM	*unsound*			*unsound*			31	11	0.45
	$SM_{2.0}$	0.76	**0.79**	**0.78**	0.86	**0.19**	**0.31**	**27**	**3**	0.59
L6	IM-lc	*measurements timeout*			0.10	0.00	0.01	**126**	**78**	**1.00**
	SM	*unsound*			*unsound*			161	98	0.14
	$SM_{2.0}$	**0.70**	**0.66**	**0.68**	**0.31**	**0.23**	**0.26**	138	80	0.50
L7	IM-lc	*discovery timeout*			*discovery timeout*			*discovery timeout*		
	SM	**0.88**	**1.00**	**0.94**	**0.73**	**0.90**	**0.81**	**12**	**2**	**1.00**
	$SM_{2.0}$	**0.88**	**1.00**	**0.94**	**0.73**	**0.90**	**0.81**	**12**	**2**	**1.00**
L8	IM-lc	**0.85**	0.40	0.55	**0.87**	0.03	0.06	61	39	**1.00**
	SM	*unsound*			*unsound*			160	118	0.02
	$SM_{2.0}$	0.77	**0.76**	**0.77**	0.38	**0.33**	**0.35**	**46**	**26**	0.70
L9	IM-lc	**0.94**	0.26	0.41	**0.92**	0.43	**0.58**	23	11	**1.00**
	SM	*unsound*			*unsound*			**17**	**5**	0.53
	$SM_{2.0}$	0.57	**0.91**	**0.70**	0.28	**0.45**	0.35	**17**	**5**	0.47
L10	IM-lc	**0.95**	0.75	0.84	**0.98**	0.15	0.26	31	8	**1.00**
	SM	0.77	**1.00**	**0.87**	0.95	**0.93**	**0.94**	**29**	**6**	**1.00**
	$SM_{2.0}$	0.77	**1.00**	**0.87**	0.95	**0.93**	**0.94**	**29**	**6**	**1.00**
L11	IM-lc	**0.91**	0.75	0.82	**0.45**	0.14	0.22	36	21	**1.00**
	SM	0.83	**0.90**	**0.87**	0.29	0.26	**0.28**	44	28	0.16
	$SM_{2.0}$	0.84	**0.90**	**0.87**	0.06	**0.33**	0.10	**22**	**11**	0.59
R-Log	IM-lc	**0.99**	0.98	**0.99**	**1.00**	0.96	**0.98**	16	5	**1.00**
	SM	0.91	**0.99**	0.95	0.45	0.83	0.59	**14**	**4**	0.36
	$SM_{2.0}$	0.98	0.97	0.98	0.94	**0.98**	0.96	16	5	0.50

Table 3: Experiment results.

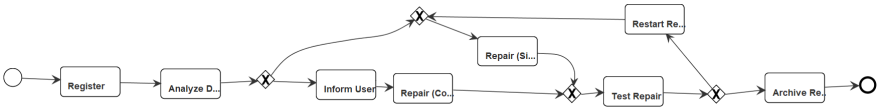
ures 6c and 6d), only $SM_{2.0}$ discovers the concurrency relations between its activities, while SM mixes us the concurrency relations with loops.



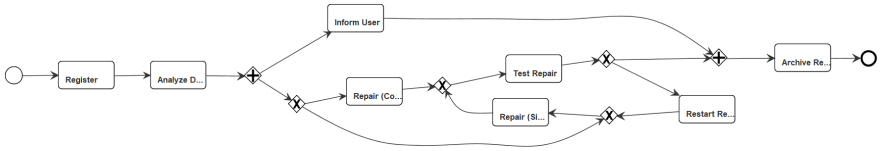
(a) SM model discovered from L6.



(b) $SM_{2.0}$ model discovered from L6.



(c) SM model discovered from R-Log.



(d) $SM_{2.0}$ model discovered from R-Log.

Fig. 6: Models discovered by SM and $SM_{2.0}$ from the L6 and R-Log.

5 Conclusion

In this paper, we presented Split Miner 2.0 ($SM_{2.0}$), an extension of Split Miner (SM) [6] that exploits the activities' start and end timestamps recorded in an event log to discover true concurrency and inclusive choice relations between activities. This is achieved by redesigning the discovery of a directly-follows graph from an event log, adapting the concurrency notion of Van der Werf et al. [18], and introducing an intuitive heuristic to identify inclusive relations. Furthermore, given that SM cannot guarantee sound process models, we designed an heuristic that reduces the chances of discovering process models exhibiting improper completion. The empirical evaluation shows that $SM_{2.0}$ can discover more concurrent relations than SM, remove improper completion, and identify OR-splits, while preserving SM's model accuracy and reducing the complexity.

Although several studies have investigated the problem of automated process discovery from event logs [5], most of them operate on simple event logs with only three attributes: case id, timestamp, and activity label. Future research work in this area may

focus on designing more sophisticated automated process discovery algorithms that can discover more complex BPMN process models by leveraging additional information that may be available in real-life event logs. Another direction for future work is to design accuracy measures such as fitness and precision that go beyond simple control-flow relations and include support for inclusive gateways, including the OR-join.

Acknowledgments. Research funded by the Australian Research Council (grant DP180102839) and the Estonian Research Council (grant PRG887).

References

1. A. Adriansyah, J. Munoz-Gama, J. Carmona, B. van Dongen, and W.M.P. van der Aalst. Alignment based precision checking. In *BPM*. Springer, 2012.
2. A. Adriansyah, J. Munoz-Gama, J. Carmona, B. van Dongen, and W.M.P. van der Aalst. Measuring precision of modeled behavior. *ISeB*, 13(1), 2015.
3. J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.
4. A. Augusto, A. Armas Cervantes, R. Conforti, M. Dumas, and La Rosa. Measuring fitness and precision of automatically discovered process models: A principled and scalable approach. *IEEE TKDE (to appear)*, 2020.
5. A. Augusto, R. Conforti, M. Dumas, M. La Rosa, F.M. Maggi, A. Marrella, M. Mecella, and A. Soo. Automated discovery of process models from event logs: Review and benchmark. *IEEE TKDE*, 31(4), 2019.
6. A. Augusto, R. Conforti, M. Dumas, M. La Rosa, and A. Polyvyanyy. Split miner: automated discovery of accurate and simple business process models from event logs. *KAIS*, 2018.
7. J. Buijs, B. van Dongen, and W. van der Aalst. On the role of fitness, precision, generalization and simplicity in process discovery. In *CoopIS*. Springer, 2012.
8. A. Burattin and A. Sperduti. Heuristics miner for time intervals. In *ESANN*, 2010.
9. S.J.J. Leemans and D. Fahland. Information-preserving abstractions of event data in process mining. *Knowledge and Information Systems*, 62(3):1143–1197, 2020.
10. S.J.J. Leemans, D. Fahland, and W.M.P. van der Aalst. Discovering block-structured process models from event logs containing infrequent behaviour. In *BPM Workshops*. Springer, 2014.
11. S.J.J. Leemans, D. Fahland, and W.M.P van der Aalst. Using life cycle information in process discovery. In *BPM*, pages 204–217. Springer, 2016.
12. J. Mendling. *Metrics for process models: empirical foundations of verification, error prediction, and guidelines for correctness*, volume 6. Springer Science & Business Media, 2008.
13. G. Schimm. Mining exact models of concurrent workflows. *Comput Ind*, 53(3):265–281, 2004.
14. A. Senderovich, M. Weidlich, and A. Gal. Temporal network representation of event logs for improved performance modelling in business processes. In *International Conference on Business Process Management*, pages 3–21. Springer, 2017.
15. A.F. Syring, N. Tax, and W.M.P. van der Aalst. Evaluating conformance measures in process mining using conformance propositions. In *Transactions on Petri Nets and Other Models of Concurrency XIV*, pages 192–221. Springer, 2019.
16. W.M.P. van der Aalst. *Process Mining - Data Science in Action*. Springer, 2016.
17. W.M.P. van der Aalst, K. van Hee, A. ter Hofstede, N. Sidorova, E. Verbeek, M. Voorhoeve, and M. Wynn. Soundness of workflow nets: classification, decidability, and analysis. *Formal Asp. Comput.*, 23(3), 2011.
18. J.M.E.M. van der Werf, R. Mans, and W.M.P. van der Aalst. Mining declarative models using time intervals. In *PNSE+ ModPE*, pages 313–331. Citeseer, 2013.
19. L. Wen, J. Wang, W.M.P. van der Aalst, B. Huang, and J. Sun. A novel approach for process mining based on event types. *J Intell Inf Syst*, 32(2):163–190, 2009.