Discovering Unseen Behaviour from Event Logs

Abel Armas Cervantes(✉) and Farbod Taymouri

The University of Melbourne, Melbourne, Australia
abel.armas@unimelb.edu.au

Abstract. Process mining techniques aim to discover insights into the performance of a business process by analysing its event logs. These logs capture historical process executions as sequences of activity occurrences (events). Often, event logs capture only part of the possible process behaviour because the number of executions can be very large, particularly when many activities are executed concurrently. A highly incomplete event log is problematic because process mining techniques use the event log as a starting point. This paper proposes a technique to discover behaviour from an incomplete log. In order to do so, the presented technique builds distributive lattices from the executions captured in the log, which have well-defined notions of completeness and can be used to discover behaviour from few observations. The paper tests the presented approach in a set of real-life event logs and measures the amount of behaviour that can be discovered.

Keywords: Process mining · Distributive lattices · Partial orders · Concurrency detection

1 Introduction

Process mining analyses historical business process executions to help discover fact-based opportunities for process improvements [17]. These historical executions are captured as event logs (or simply *logs*), where process executions are recorded as sequences (*traces*) of activity instances (*events*). These traces of events describe the order in which the activities were executed, thus the concurrent execution of activities are captured as interleavings. These logs are used as the starting point for various process mining operations, but there are three main ones: automated process model discovery, conformance checking (checking the conformance between a model describing expected behaviour and the log describing the observed behaviour) and process enhancement.

Event logs can capture only a handful of possible executions of the underlying process [20]. As the complexity of the process increases, it becomes more difficult to observe all possible traces that a process can generate, which is infinite in cases when there is looping behaviour. However, even in the finite case, when there is a large amount of activities that can be executed concurrently, it will be nearly impossible to capture all possible interleavings. In the worst case, it is

© Springer Nature Switzerland AG 2022
L. Bernardinello and L. Petrucci (Eds.): PETRI NETS 2022, LNCS 13288, pp. 23–42, 2022.
https://doi.org/10.1007/978-3-031-06653-5_2

necessary to have $n!$ traces to represent all possible interleavings of n concurrent activities. The completeness of a log is a critical issue because the great majority of process mining operations use the log as a starting point, and if the log is highly incomplete, then the analysis results can be highly inaccurate. In practice, the process model describing the actual behavior of the process is not available, hence it is impossible to assess the completeness of the log. But by making some assumptions about the underlying process, it is possible to gain a rough idea of the behavior the process generates.

The seminal work of Winskel et al. [12] shows the relationship between a family of Petri nets (conflict-free 1-safe nets), event structures and distributive lattices. While Petri nets and event structures explicitly represent concurrency between events, lattices represent concurrency as interleavings, where pairs of concurrent events form squares in the lattices. Figure 1 shows an example of the transformations defined in [12], where (a) shows a 1-safe Petri net, (b) its event structure and (c) the distributive lattice [5,16] representing each of the execution states of the Petri net. Note that in (c), every node in the lattice represents an execution state where a set of events have taken place and are ordered by subset inclusion. This lattice represents the evolution of a process execution by means of the edges between nodes; for instance, the edge from $\{A\}$ to $\{A, B\}$ represents the occurrence of event B. In the lattices, the traces that the process can generate are represented as paths (chain of consecutive nodes in the lattice) from the empty execution state to the final execution state. In the example displayed in Fig. 1, any path from $\{\}$ to $\{A, B, C, D, E\}$ represents a possible trace of the process.

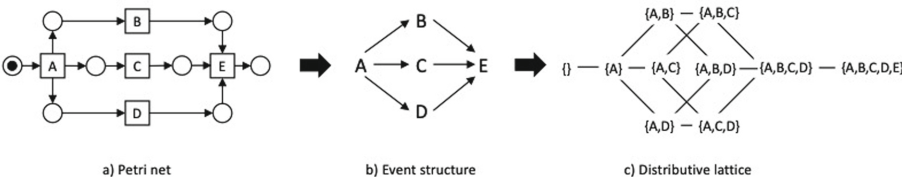


Fig. 1. From model to domain of configurations.

The aim of this paper is twofold. First, it aims to build distributive lattices from an event log. Each of these distributive lattices represents a concurrent execution that is derived from some traces in the log. If the lattices constructed from the log do not meet the distributivity property because not all interleavings were observed, then they are completed to be distributive, which will discover unseen behaviour. Second, by detecting missing behavior, a notion of event log completeness is defined, which can be used as a reference to assess the quality of a log. For the use of distributive lattices as a valid representation of an event log, two main assumptions are made: 1) the event log represents the behaviour of a 1-safe Petri net, so that each trace in the log represents the execution of the Petri net, and 2) there is no auto-concurrency in the underlying process (i.e., two activities with the same name cannot be concurrent).

The paper is structured as follows. Section 2 introduces the relevant definitions and notation for partial orders, lattices and event logs. Then, Sect. 3 presents the reconstruction of the distributive lattices from an event log, defines a notion of completeness based on the discovered behaviour, and presents a set of experiments using a set of real-life logs. Discussion and future work is presented in Sect. 4 and related work in Sect. 5. Finally, Sect. 6 concludes the paper.

2 Preliminaries

This section establishes the foundations for the rest of the paper. The first part introduces partial orders and lattices, and the second part introduces traces and event logs. Given that Petri nets are not a central element of this paper, we assume the reader is familiar with the basic notions of 1-safe Petri nets, see for example [15].

2.1 Partial Orders and Lattices

Let R be a binary relation over a set X, R is an ordering relation in X if it is reflexive ($(x, x) \in R$ for all $x \in X$), antisymmetric ($(x, y), (y, x) \in R$ implies $x = y$) and transitive ($(x, y), (y, z) \in R$ implies $(x, z) \in R$). The following definition presents reflexive partially ordered set, *poset* for short.

Definition 1 (Poset). *A* partially ordered set*, or simply* poset*, is a pair $\langle X, \leq \rangle$, where X is a set and $\leq$ is a reflexive, antisymmetric and transitive relation.*

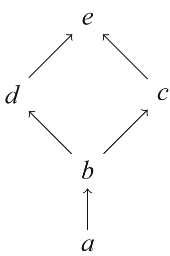


Fig. 2. Hasse diagram

Posets can be graphically represented as Hasse diagrams. These diagrams represent the *"cover" relation*, which is the transitive reduction of the graph representing the relation $\leq$ over the elements X. Given a pair of elements $x, y \in X$, y *covers* x, denoted as $x \prec y$, if $x < y$ and $\forall z \in X \ : \ x \leq z < y$ implies $x = z$. The Hasse diagram of a poset $\mathcal{P} = \langle X, \leq \rangle$ is shorthanded as $\mathcal{H}(\mathcal{P}) = \langle X, \prec \rangle$. Figure 2 shows a Hasse diagram where the cover relation is represented as arrows and the elements are represented by letters; e.g., b covers a in the diagram.

Meet and *join* are two common operators in a poset. Given a poset $\langle X, \leq \rangle$, $x \in X$ is the meet of a set $Y \subseteq X$, denoted $x = \bigsqcap Y$, iff (1) $\forall y \in Y \ : \ x \leq y$ and (2) $\forall z \in X : (\forall y \in Y : z \leq y) \Rightarrow z \leq x$. If Y contains only two elements, $X = \{a, b\}$, then the meet operation is written as $a \sqcap b$. Analogously, the join of $Y \subseteq X$ is an element x, denoted as $x = \bigsqcup Y$, iff (1) $\forall y \in Y \ : \ y \leq x$ and (2) $\forall z \in X : (\forall y \in Y : y \leq z) \Rightarrow x \leq z$. If $Y = \{a, b\}$, the join operation is written as $a \sqcup b$. The meet and join are also known as greatest lower bound (glb) and least upper bound (lub), respectively. For example, in the Hasse diagram in Fig. 2, $d \sqcup c = e$ and $d \sqcap c = b$.

A lattice is a special type of poset that contains a join and a meet for every pair of elements in the set X. One can easily check that the Hasse diagram in

Fig. 2 represents a lattice. Lattices are *distributive* [5] if they are distributive over $\sqcup$ and $\sqcap$. The following definition formalises lattices and the distributive property.

Definition 2 (Lattices and distributive lattices). *A* lattice *is a poset* $\langle X, \leq \rangle$ *where* $\forall x, y \in X : x \sqcup y$ *and* $x \sqcap y$ *exist. A lattice is* distributive *if* $\forall x, y, z \in X \; : \; x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$.

Some elements in the lattice can be of one of three types: top element, bottom element and prime. An element x is the bottom element if $\forall y \in X : x \leq y$, and x is the top element if $\forall y \in X : y \leq x$. An element in a poset is *prime* if it is neither the bottom element nor the join of two other elements, see the formal definition below.

Definition 3 (Prime elements). *Let* $\mathcal{P} = \langle X, \leq \rangle$ *be a lattice. An element* $x \in X$ *is a complete prime (*prime *for short) iff for every* $Y \subseteq X$ *iff* $\bigsqcup Y$ *exists and* $x \leq \bigsqcup Y$*, then there exists* $y \in Y$ *such that* $x \leq y$*. The set of complete primes for* $\mathcal{P}$ *is denoted as* $\mathcal{C}_{\mathcal{P}}$*.*

In a distributive lattice, prime elements are those covering exactly one element [10]. The set of primes below an element $x \in X$ (w.r.t. $\leq$) are denoted as $\phi(x) = \{x' \in \mathcal{C}_{\mathcal{P}} \mid x' \leq x\}$. For example, in Fig. 2, the elements b, c and d are primes, and $\phi(d) = \{d, b, a\}$. All the elements between a pair of elements x and y define an *interval*, and it is called a *prime interval* if it contains only x and y.

Definition 4 (Prime interval). *Let* $\langle X, \leq \rangle$ *be a poset, an* interval *between* $x_1, x_2 \in X$ *is* $[x_1, x_2] = \{x_3 \in X \mid x_1 \leq x_3 \leq x_2\}$ *and it is* prime *iff* $x_1 \neq x_2$ *and* $[x_1, x_2] = \{x_1, x_2\}$*.*

Let $pr([x, y]) = \phi(y) \backslash \phi(x)$ be the set difference between the primes below x and the primes below y. In the case of a prime interval $[x, y]$, $pr([x, y])$ is a singleton. A pair of prime intervals $[x_1, x_2]$ and $[x_3, x_4]$ are said to be *equivalent*, denoted as $[x_1, x_2] \equiv [x_3, x_4]$ iff their difference is the same prime element, i.e., $pr([x_1, x_2]) = pr([x_3, x_4])$.

2.2 Distributive Lattices and Concurrency

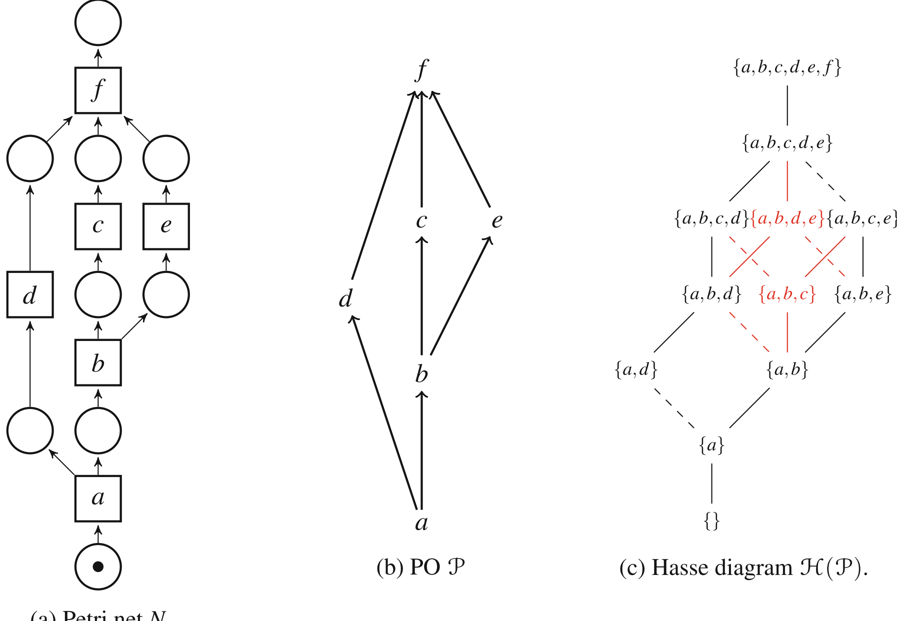
Winskel et al. [12] shows the connection between conflict-free 1-safe Petri nets, elementary event structures and distributive lattices[1]. Elementary event structures are posets describing the execution of events by means of causality $\leq$, such that an event a has to occur before an event b when $a < b$. The authors showed that the execution states of the unfolding (elementary event structure) of a conflict-free 1-safe Petri net form a distributive lattice when ordered by set

[1] The connection defined in [12] considers a family of Petri nets called *causal nets* where there is not conflict – every place has at most one transition connected from and to it – and F^+ is irreflexive, where $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation of the net with places P and transitions T.

inclusion; furthermore, the prime elements of the distributive lattice correspond to the events in the elementary event structure. The distributive lattice captures all possible interleavings of concurrent events, which can occur at different executions states, and the order $\leq = \subseteq$. Additionally, the meet ($\sqcap$) and join ($\sqcup$) operators correspond to the set intersection ($\cap$) and union ($\cup$) set operations, respectively. As shown in [5,16], a distributive lattice is a ring of sets where the union and intersection of every pair of elements are present in the lattice.

Figure 3 shows a conflict-free 1-safe Petri net N (Fig. 3a), the corresponding elementary event structure $\mathcal{P}$ (Fig. 3b) and the distributive lattice $\mathcal{H}(\mathcal{P})$ of its execution states ordered by $\subseteq$ (Fig. 3c). In the *elementary event structure*, the behavior is described by means of causality ($\leq$) and concurrency between events, such that events $x, y \in X$ are concurrent iff $\neg(x \leq y \ \vee \ y \leq x)$. The execution states, *a.k.a. configurations*, of an elementary event structure are left-closed subsets of events (i.e., $Y \subseteq X$ is left closed iff $x \in Y \wedge x' \leq x \Rightarrow x' \in Y$). In $\mathcal{H}(\mathcal{P})$, the nodes are the execution states, and the cover relation is denoted by a line, please disregard the different colors and line formats as they are explained later. The cover relation describes the evolution of a configuration and represents the occurrence of an event. For instance, in Fig. 3c, the cover relation between $\{a\}$ and $\{a, d\}$ represents the execution of the event d after a. The bottom and top element in this distributive lattice represents the state where no event has been executed $\{\}$ and the final state $\{a, b, c, d, e, f\}$ where all events have occurred.

In the distributive lattice, the concurrent executions of pairs of events form diamond-like shapes representing interleavings. For example, in Fig. 3c, the states



(a) Petri net N

(b) PO $\mathcal{P}$

(c) Hasse diagram $\mathcal{H}(\mathcal{P})$.

Fig. 3. Petri net, partial order and Hasse diagram

$\{a\}, \{a, d\}, \{a, b\}$ and $\{a, b, d\}$ represent the concurrent execution of events b and d. Note, for instance, both intervals $[\{a\}, \{a, d\}]$ and $[\{a, b\}, \{a, b, d\}]$ represent the execution of the event d (i.e., $pr([\{a\}, \{a, d\}]) = pr([\{a, b\}, \{a, b, d\}])$ and thus $[\{a\}, \{a, d\}] \equiv [\{a, b\}, \{a, b, d\}]$). In fact, all dotted lines in Fig. 3c are an equivalence class because they represent occurrences of the event d at different execution states.

2.3 Traces and Event Logs

An *event log* (or simply *log*) captures historical executions of a process, where every execution of a process activity produces an event in the log. Thus, several events in the log may stem from the same activity. Hereinafter the activities of a process are represented as Σ and the events as E. An event can have different attributes, such as the name of the corresponding activity, resources or execution time. In this paper, we assume that the only available attribute of an event is the name of the corresponding activity. In order to relate activities to events, $\lambda : E \rightarrow \Sigma$ is a labeling function, such that the activity of an event e is denoted as $\lambda(e) = l$, where $l \in \Sigma$.

Process executions are captured in the log by means of traces. These traces are sequences of events ordered by their order of observation. A pair of traces are considered the same if they have the same number of events and those events are instances of the same activities executed in the same order. A log L can contain several occurrences of the same traces, thus a log is defined as a multiset of traces. In some occasions, the set representation of the log – containing only distinct traces – will be used and represented as $Set(L)$.

Definition 5 (Trace and event log). *Given a finite set of events E, a* trace $\sigma = \langle e_1, e_2, \ldots, e_n \rangle \in E^*$ *is a sequence of events and an* event log $L \subseteq E^*$ *is a multiset of traces. The set of distinct traces in the log is represented as $Set(L)$. The number of occurrences of the same trace in the log (multiplicity) is denoted as $\gamma(\sigma)$.*

In the presence of concurrency, a single process execution can be captured in the log by different traces. These traces contain the same activity occurrences but vary in order. For example, $\langle a, b, e, c, d, f \rangle$ and $\langle a, d, b, c, e, f \rangle$ are two possible traces generated by a process where d is concurrent with b, c and e (see Fig. 3b).

Let us define some notation for traces. The length of a trace $\sigma = \langle e_1, e_2, \ldots, e_n \rangle$ is shorthanded as $|\sigma| = n$ and it is the number of events in σ. The event at the ith position is accessed as $\sigma[i]$. The prefix $\sigma[1, k]$ of a trace σ contains the first k elements of the trace, i.e., $\sigma[1, k] = \langle e_1, e_1, \ldots, e_k \rangle$ for $1 \leq k \leq |\sigma|$.

The labelling function previously defined for events can be extended to prefixes of traces where $\lambda(\sigma[1, k]) = \langle \lambda(e_1), \lambda(e_2), \ldots, \lambda(e_k) \rangle$. A trace or prefix of a trace can be represented as a set or as a multiset of labels. These representations of a (prefix of a) trace σ are denoted as $Set(\sigma)$ and $MultiSet(\sigma)$, respectively.

The cardinality of a set and a multiset Z is denoted as $|Z|$ and refers to the number of elements in Z; note that in the case of multisets, the cardinality considers the multiplicities of the elements in Z.

The following section presents the main contribution of the paper, where traces are used as "seeds" to construct distributive lattices.

3 Distributive Lattices of an Event Log

This section presents the main contributions of the paper. The central idea is the reconstruction of distributive lattices representing execution states of a process execution. This reconstruction starts by merging groups of event log traces into lattices, such that prefixes of a trace represent an execution state, and then missing elements are added until the lattices are distributive.

The completion operation over the lattice can discover new behavior by introducing new elements (unseen execution states or event executions) until the distributivity property is met. However, in the presence of noise, this operation can introduce a large amount of new behavior that may be undesirable. Thus, Subsect. 3.1 presents the steps for the construction of the distributive lattice from a set of traces, and Subsect. 3.2 describes a way to tame the possibly large amount of behavior introduced during the completion operation.

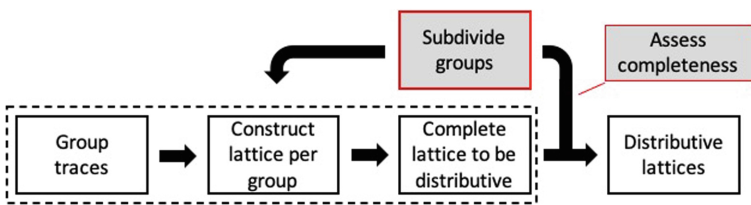


Fig. 4. Overview of the proposed approach.

3.1 Reconstruction of Distributive Lattice

The reconstruction of the distributive lattice is inspired by two main existing results: the behavior of a conflict-free 1-safe Petri net forms a distributive lattice [12] and a distributive lattice is a ring of sets [5,16]. Thus, the proposed approach consists of three central steps (dotted area in Fig. 4): 1. group traces representing the same process executions (interleavings of concurrent executions), 2. construct a lattice for each group of traces, and 3. complete the lattice until it is distributive (missing elements can be computed by using two set operations: union and intersection).

Before presenting the three steps of the reconstruction, let us define a special labelling for handling traces with events with the same name. In a trace, every event is unique but, when the process originating the log contains repeatable behavior, several events within a trace can stem from the same activity. In order to differentiate the events with the same label within a trace σ, $\bar{\lambda}$ is a special labelling function, such that $\bar{\lambda}(e) = \lambda(e)_{w(e,i)}$, where $w(e,i) = |\{e' = \sigma[j] \mid 1 \leq j \leq i \ \wedge \ \lambda(e') = \lambda(e)\}|$. The special labelling can be applied to traces, where $\bar{\lambda}(\sigma) = \langle \bar{\lambda}(e_1), \bar{\lambda}(e_2), \ldots \bar{\lambda}(e_{|\sigma|}) \rangle$. Intuitively, the special label of an event has a sub-index representing its number of occurrence within the trace. For instance, given a trace $\sigma = \langle a, b, c, c \ldots \rangle$, then $\bar{\lambda}(\sigma) = \langle a_1, b_1, c_1, c_2 \ldots \rangle$.

Grouping Traces. The first step is to define a notion of equivalence over the log traces, such that a pair of traces are equivalent if they represent the same process execution. A simple notion of equivalence is multiset equivalence, where traces are equivalent if they have events of the same activities that were executed the same number of times. For example, the traces $\langle a, b, e, c, d, f \rangle$ and $\langle a, d, b, c, e, f \rangle$ would be considered as multiset equivalent because they represent a single execution of the same activities. Note that if the special labelling is used, the multiset equivalence of a trace can be defined as a set equivalence as shown next.

Definition 6 (Set equivalent traces). *A pair of traces σ, σ' are set equivalent, denoted as $\sigma \sim_{set} \sigma'$, iff $Set(\bar{\lambda}(\sigma)) = Set(\bar{\lambda}(\sigma'))$. I.e., $|\sigma| = |\sigma'|$ and $\bar{\lambda}(\sigma[i]) \in Set(\bar{\lambda}(\sigma'))$ for all $1 \leq i \leq |\sigma|$.*

Given that the aim of the presented approach is to build a distributive lattice, where the top element represents the final process execution, the minimum condition to consider a pair of traces as equivalent is multiset equivalence.

Constructing the Lattice. The second step is to build a lattice from a set of $\sim_{set}$-equivalent traces. A trace σ represents execution states, where a prefix $\sigma[1,k]$ is the state where events $\langle e_1, e_1, \ldots, e_k \rangle$ have occurred. Indeed, each element in the lattice will represent a trace prefix. The following definition formally defines a prefix equivalence between traces.

Definition 7 (Equivalent prefixes). *Let σ_1, σ_2 be two traces. The prefixes $\sigma_1[1,k]$ and $\sigma_2[1,k]$ are equivalent iff $\sigma_1[1,k] \sim_{set} \sigma_2[1,k]$.*

Given a group of equivalent traces G, let $\sigma[1,k]_{\equiv} = \{\sigma'[1,k] \mid \sigma' \in G \ \wedge \ \sigma[1,k] \sim_{set} \sigma'[1,k]\}$ be the equivalence class for the trace prefix $\sigma[1,k]$. The lattice representing G is the pair $\langle X, \subseteq \rangle$, where X represents the equivalence classes for the prefixes of traces in G and the order between the elements in X is the subset containment between a pair of elements in the equivalence classes.

Definition 8 (Lattice of traces). *Given a log L, the lattice of a set of traces $T \subseteq L$ is a pair $\langle X, \subseteq \rangle$, where $X = \{\sigma[1,k]_{\equiv} \mid \sigma \in L \ \wedge \ 1 \leq k \leq |\sigma|\} \cup \epsilon$ and ϵ is a special state representing the bottom element, and $x \subseteq y$ for $x, y \in X$, if $\bar{\lambda}(\sigma') \subseteq \bar{\lambda}(\sigma'')$, such that $\sigma' \in x$ and $\sigma'' \in y$.*

Observe that the above definition constructs a valid lattice because there is a unique bottom element ϵ and a unique top element $Set(\sigma)$ that can be the meet and join elements, respectively, for any pair of elements. Please, observe that in the Hasse diagram we denote ϵ as the empty set $\{\}$.

Reconstructing Distributive Lattices. The completion of the lattice consists of introducing the relations and elements needed to transform any lattice into a distributive one. This completion operation is based on the fact that a distributive lattice is isomorphic to a ring of sets [5], where the union and intersection of every pair of elements is also an element in the lattice.

Definition 9 (Lattice completion). *Let $\mathcal{D} = \langle X, \leq \rangle$ be a lattice. The completion of $\mathcal{D}$ for a distributive lattice is a lattice $\mathcal{D}^* = \langle Y, \leq \rangle$ where $X \subseteq Y$ and $(x \cup y) \in Y$ and $(x \cap y) \in Y$ for all $x, y \in Y$.*

Consider the conflict-free 1-safe Petri nets shown in Fig. 5. The possible execution states of these nets can be represented as the distributive lattices displayed in Fig. 5. These lattices can be reconstructed from only two traces (represented by the black elements and relations). For example, traces $\langle a, b, c, d \rangle$ and $\langle b, c, d, a \rangle$ are the only traces necessary to reconstruct the lattice in Fig. 6a, where activity a can occur after $\{b\}$ and $\{b, c\}$; after completion, the lattice represents four traces. In Fig. 6b, traces $\langle a, b, c, d \rangle$ and $\langle c, d, a, b \rangle$ can be used to generate the distributive lattice where $\{a, c\}$ is the union (intersection) of $\{a\}$ and $\{c\}$ (resp. $\{a, b, c\}$ and $\{a, c, d\}$), which represents six traces. Finally, for the reconstruction of the lattice in Fig. 6c only traces $\langle a, b, c \rangle$ and $\langle c, b, a \rangle$ are necessary to obtain the lattice representing six traces.

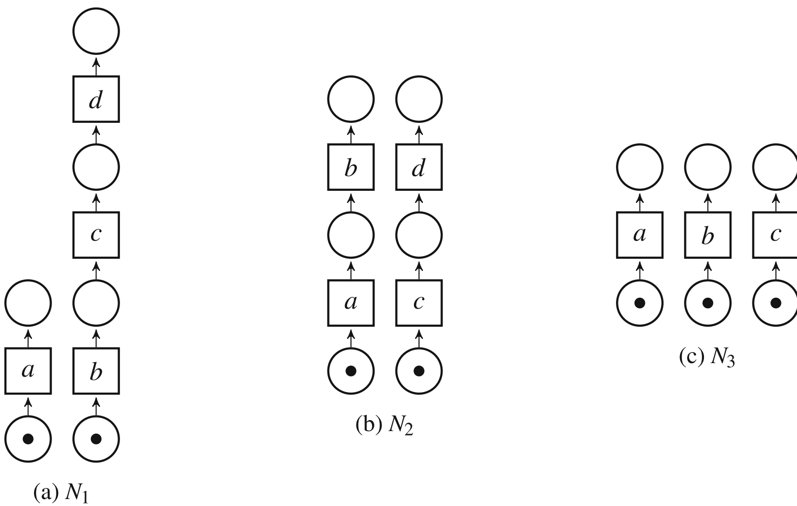


Fig. 5. Petri nets with computations represented in the distributive lattices in Fig. 6

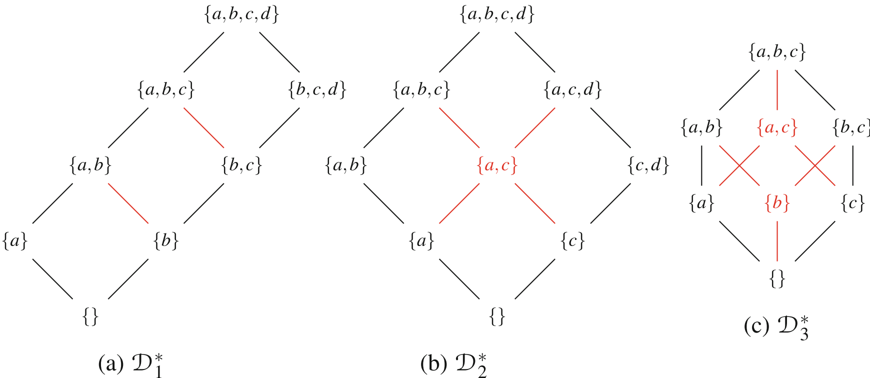


Fig. 6. Completed distributive lattices representing the execution states of the nets in Fig. 5

Note that other equivalences have been proposed in the context of process mining (see [19]) for constructing transition system-based representations of an event log, which are not necessarily lattices (e.g., some of the transitions systems can represent loops). [19] puts forward the idea of closing the diamonds (called *extend strategy*) in a transition system. This operation is able to handle cases as that in Fig. 6a, but it is limited to discover relations between events, and fails to discover missing elements (i.e., execution states). Thus, [19] would fail to reconstruct the lattices shown in Figs. 6b and 6c. This discussion is expanded in the related work in Sect. 5.

Another example of the possible lattices that can be reconstructed is presented in Fig. 7. The displayed lattice is built from the traces $\langle a, b, e, c, d, f\rangle$ and $\langle a, d, b, c, e, f\rangle$. This is not a distributive lattice because neither the union of the states $\{a, b, d\}$ and $\{a, b, e\}$, nor the intersection of $\{a, b, c, d\}$ and $\{a, b, c, e\}$ are present. The completion of the lattice in Fig. 7 is that shown in Fig. 3c, where all added elements and relations are in red.

As noted previously, by construction, a completed lattice is distributive because it is a ring of sets and hence distributive. The following proposition simply states the fact that the result of the completion operation is a distributive lattice.

$\{a,b,c,d,e,f\}$
$\{a,b,c,d,e\}$
$\{a,b,c,d\}$ $\{a,b,c,e\}$
$\{a,b,d\}$ $\{a,b,e\}$
$\{a,d\}$ $\{a,b\}$
$\{a\}$
$\{\}$

Fig. 7. Lattice

Proposition 1. *The completion $\mathcal{D}^*$ of a lattice $\mathcal{D}$ is distributive.*

Before moving to the next subsection, let us discuss a possible encoding of the lattice. Using the special labelling function, each execution state can be encoded as a bit-vector, where every position in the vector represents an event's special label. A value of 1 at a given position represents the occurrence of such an event. The bit-vector encoding will be particularly

useful during the completion of the lattice when many union and intersection operations will be performed.

3.2 Containing the Amount of Behaviour Discovered During Completion

The completion operation, while simple, can be a double-edged sword. In the case of processes capable of generating large amounts of distinct traces (for example when there are many concurrent activities), even the majority of the behaviour can be reconstructed from a few traces acting as seeds. For instance, the lattices in Fig. 6 can be fully reconstructed from only two traces. However, this makes the technique too sensitive to noise or exceptional behavior that is not part of the "usual" process behaviour. In this case, a single (noisy) trace can lead to the insertion of a large amount of behaviour when computing the missing elements of a lattice. Note that, the filtering of noise in an event log is an orthogonal problem that has been studied independently (see for example the works in [6,8,22]), thus a noise-filtering technique can be applied to the log as a pre-processing step before the reconstruction of the lattices.

In order to control large numbers of behaviours that might be introduced during the completion operation, this subsection presents a strategy to, first, compute the completeness of a log, and then to control the amount of behavior introduced in the lattice.

Measuring Completeness. The completeness of a lattice is defined with respect of the number of traces represented after completion. Then, a Hasse diagram of a distributive lattice is seen as a graph where every path (sequence of contiguous edges) from the bottom to the top element represents a trace. Formally, a path p in a poset $\langle X, \subseteq\rangle$ is $p = \langle s_1, s_2, s_3, \ldots, s_n\rangle$, where $s_i \in X$, for $1 \leq i \leq n$, and $s_j \prec s_{j+1}$ for $1 \leq j < n$. Note that for extracting the traces from the paths, it is necessary to look at the events represented by the prime intervals of two consecutive nodes. Thus, the trace represented by a path p is $t(p) = \lambda(\langle pr([s_1, s_2]), pr([s_2, s_3]), \ldots,\ pr([s_{n-1}, s_n])\rangle)$. For instance, Fig. 7 has a path $p = (\{\}, \{a\}, \{a, b\}, \{a, b, e\}, \{a, b, c, e\},\ \{a, b, c, d, e\}, \{a, b, c, d, e, f\})$, and the corresponding trace of p is $t(p) = \langle a, b, e, c, d, f\rangle$.

The measure of completeness Θ for a lattice is computed as follows. $\Theta(\mathcal{D})$ is defined as the ratio between the total number of paths in the completed lattice $\mathcal{D}^*$ and the number of paths in the lattice $\mathcal{D}$ prior completion, where $\mathcal{D}_{\#P}$ and ${\mathcal{D}^*}_{\#P}$ are the set of paths in the lattice and its completed version, respectively.

$$\Theta(\mathcal{D}) = \frac{|\mathcal{D}_{\#P}|}{|{\mathcal{D}^*}_{\#P}|} \tag{1}$$

A Hasse diagram is a directed acyclic graph, thus using dynamic programming, the number of paths can be computed in $\mathcal{O}(V + E)$ where V is the number of set elements and E is the number of cover relations.

The completeness measure can be used to control the amount of behavior introduced in the lattice during completion. For instance, the lattice of a set of traces G can be deemed as *valid* if $\Theta(\mathcal{D}) \geq \beta$, where β is a given threshold. Then,

if $\Theta(\mathcal{D}) < \beta$ then G has to be refined into subgroups and the lattices for such subgroups have to be constructed independently. Intuitively, if the completion operation introduced more behavior than desired according to β, i.e. $\Theta(\mathcal{D}) < \beta$, it is necessary to subdivide the group of traces where the completion operation will introduce less new elements, relations and, as a consequence, fewer paths. The way to subdivide the groups of traces is left for future work. In the current tool implementation, a hierarchical clustering was used as a black box, such that the distance between σ and σ' is $|\{\sigma[i] \mid \lambda(\sigma[i]) \neq \lambda(\sigma'[i]) \text{ for } 1 \leq i \leq |\sigma|\}|$. In words, the distance between a pair of traces is the number of events that are not the same at a given position in both traces.

Consider the lattice in Fig. 6c. In this lattice, the black lines represent two traces $\langle a, b, c\rangle$ and $\langle c, b, a\rangle$, while the red elements are inserted. In this example, the completed lattice represents six traces, while only two were given as seeds, thus $\Theta(\mathcal{D}_3) = \frac{2}{6} = 0.333$. Then, if $\beta > 0.333$, then it would be necessary to build one lattice for the trace $\langle d, c, e\rangle$ and one for $\langle e, c, d\rangle$.

3.3 Experiments

In order to test how much behaviour we can discover in real-life event logs, the approach was implemented and tested using a set of publicly available logs. This section presents the results of the presented approach with a series of real-life event logs. For reproducibility purposes, the library, benchmark and results (lattices and elementary event structures were included for completeness) can be found in this link: *Latticer* at https://blogs.unimelb.edu.au/bpm/tools/.

Datasets. The experiments were conducted using 11 publicly available real-life logs obtained from the 4TU Centre for Research Data.[2] Table 1 shows the

Table 1. Event logs

Log name	#Events	#Distinct events	#Traces	#Distinct Traces	Trace length	
					Min.	Max.
$BPIC12$	262200	36	13087	4366	3	175
$BPIC13_{cp}$	6660	4	1487	183	1	35
$BPIC13_{inc}$	65533	4	7554	1511	1	123
$BPIC14_{f}$	369485	9	41353	14948	3	167
$BPIC15_{1f}$	21656	70	902	295	5	50
$BPIC15_{2f}$	24678	82	681	420	4	63
$BPIC15_{3f}$	43786	62	1369	826	4	54
$BPIC15_{4f}$	29403	65	860	451	5	54
$BPIC15_{5f}$	30030	74	975	446	4	61
$RTFMP$	561470	11	150370	231	2	20
$SEPSIPS$	15214	16	1050	846	3	185

[2] https://data.4tu.nl/Eindhoven_University_of_Technology/categories/Commerce_Management_Tourism_and_Services/13500.

characteristics of these logs including the number of (unique) events, (unique) traces, and the minimum and maximum length of traces for each event log.

Results. The distributive lattices for each of the event logs were computed with different thresholds β ranging from 0.0 to 1.0. Table 2 shows the number of lattices generated with different thresholds β. Intuitively, the lower β, the more behaviour is accepted when completing the lattice and potentially the fewer lattices.

The best result was obtained in the case of $BPIC13_i$, where the number of lattices decreased from 1394 to 535 for $\beta = 1.0$ and $\beta = 0.0$, respectively; whereas the smallest reduction in number of lattices was observed in the case of $BPIC15_5f$, where the number decreased from 295 to 264 for $\beta = 1.0$ and $\beta = 0.0$, respectively.

Table 2. Number of lattices for different β

	Threshold β										
Dataset	**1.0**	**0.9**	**0.8**	**0.7**	**0.6**	**0.5**	**0.4**	**0.3**	**0.2**	**0.1**	**0.0**
$BPIC12$	3921	3921	3914	3832	3778	3632	3606	3479	3390	3310	3159
$BPIC13_{cp}$	152	152	151	139	134	131	125	120	113	110	106
$BPIC13_i$	1394	1394	1382	1341	1292	1236	1204	1143	1040	949	535
$BPIC14_f$	13670	13656	13582	13398	13086	12559	12428	11843	11297	10511	7225
$BPIC15_{1f}$	295	295	295	295	295	295	295	288	278	277	264
$BPIC15_{2f}$	416	416	416	416	412	410	410	409	402	398	368
$BPIC15_{3f}$	785	785	785	775	750	738	736	708	702	682	598
$BPIC15_{4f}$	451	451	451	451	442	437	437	425	423	419	370
$BPIC15_{5f}$	446	446	446	446	436	436	436	436	434	434	424
$RTFMP$	152	152	151	137	123	108	106	94	91	86	85
$SEPSIS$	791	791	784	777	751	736	734	709	673	621	434

Even though the reduction of the number of traces can be considerable, the biggest impact of the technique is in the amount of behaviour introduced in the lattices. Table 3 shows the total number of paths represented by the constructed lattices. As shown in the last column, $\beta = 0.0$ can lead to a huge amount of paths. For example, in the case of $BPIC15_{4f}$, the number of paths increases from 451 when $\beta = 1.0$, which is the number of distinct traces, to more than 2 billion when $\beta = 0.0$. In this latter example, the great majority of the paths is extracted from a lattice that represents 2,087,976,600 paths and that is computed from 14 distinct traces. A reason for this can be that there is noise in the logs, or traces that are multiset equivalent but do not represent executions of a concurrent activities (e.g., the process can have activities with the same name that are simply performed in a specific order).

Table 3. Number of paths in the lattices for different β

	Threshold β										
Dataset	**1.0**	**0.9**	**0.8**	**0.7**	**0.6**	**0.5**	**0.4**	**0.3**	**0.2**	**0.1**	**0.0**
$BPIC12$	4366	4366	4368	4446	4507	4797	4877	5338	6003	7028	1128721
$BPIC13_{cp}$	183	183	184	191	197	202	213	234	286	319	461
$BPIC13_{i}$	1511	1511	1515	1540	1590	1670	1747	1967	2543	3565	95633528
$BPIC14_{f}$	14948	14951	14979	15132	15513	16596	16984	19269	22870	34609	628335832
$BPIC15_{1f}$	295	295	295	295	295	295	295	318	369	378	1454
$BPIC15_{2f}$	420	420	420	420	424	428	428	433	475	507	4308820
$BPIC15_{3f}$	826	826	826	836	863	889	899	1007	1052	1347	5416969
$BPIC15_{4f}$	451	451	451	451	460	470	470	506	517	571	2092799796
$BPIC15_{5f}$	446	446	446	446	456	456	456	456	470	470	5032
$RTFMP$	231	231	232	245	262	292	302	402	440	628	753
$SEPSIS$	846	846	848	855	882	912	928	1029	1254	1986	208796

Figures 8 and 9 show the lattices for each dataset, where every dot represents a lattice. Each graph shows the threshold which was used to create the lattice, the lattice's completeness $\Theta(\mathcal{D})$ and the number of paths it represents after completion. Please note that the dot colors are according to the thresholds β. In these graphs it is possible to observe that there are few lattices that contribute with the largest amount of paths when the threshold $\beta = 0.0$.

4 Discussion and Future Work

The main purpose of this paper is to present distributive lattices as a representation that can guide the discovery of unseen behavior from event logs. However, the advantages of using distributive lattices as a representation of the information of the event logs goes beyond the approach presented in this paper. Distributive lattices can be used as the bridge to go from traces representing interleavings of a concurrent process to models with true concurrency semantics, such as the elementary event structures (see [12]). This is particularly relevant because event structures have been proposed as suitable representations for the behaviour of process models and event logs [7].

The construction of distributive lattices from event logs has many potential uses and directions for our future work, we list some of them below.

Detection of duplicates: As a future work, we will explore the possibility to detect duplicates (activities which carry the same name but are executed in different contexts). In particular, we want to explore if the detection of these duplicates can improve the quality of the models that can be generated using automated discovery of process models.
Detection of undesirable behaviour: While this paper was focused on deriving behavior from observed traces, it may also represent behaviour that should be forbidden if the log is complete. In this case, any other trace inserted during

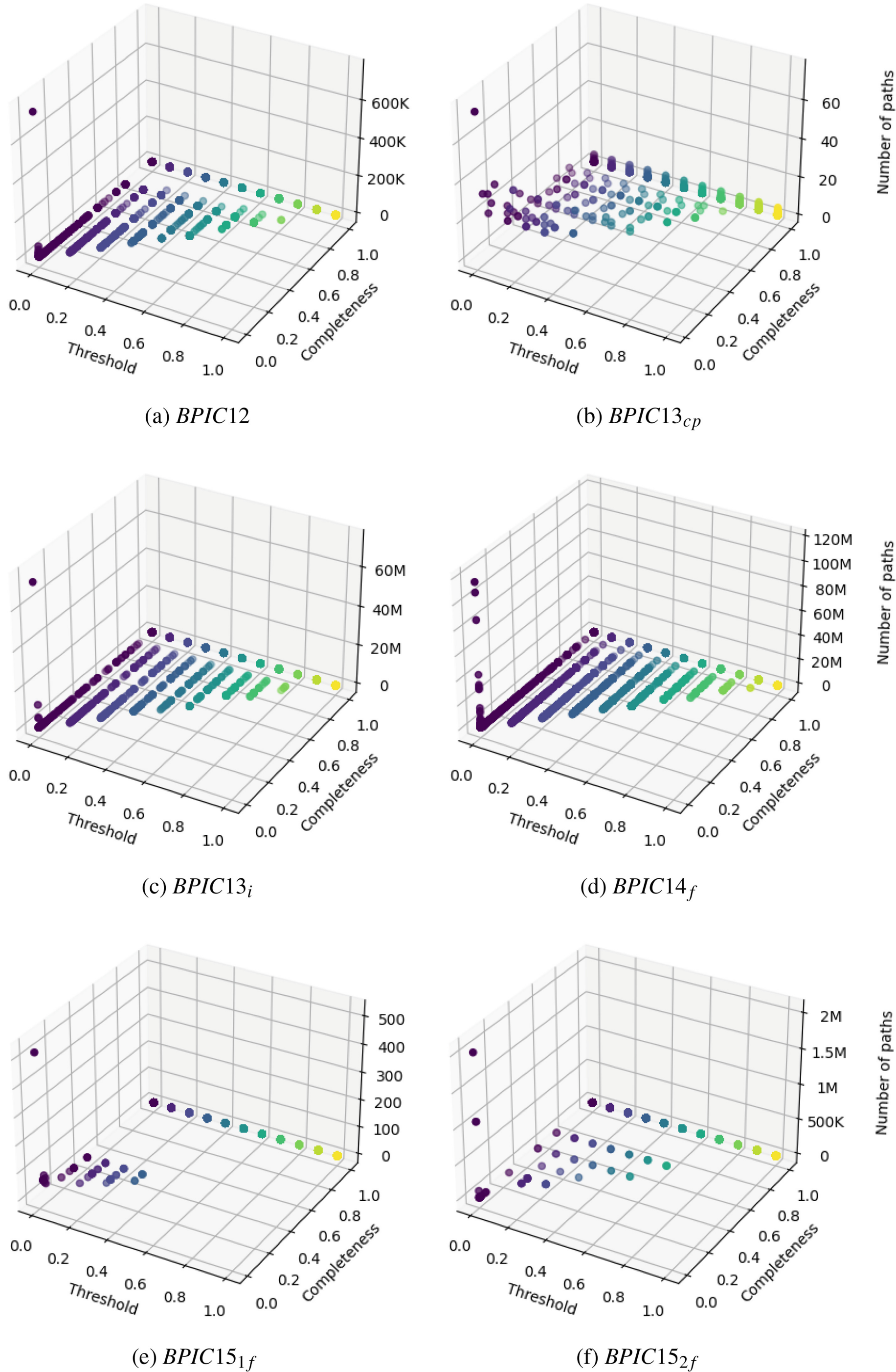
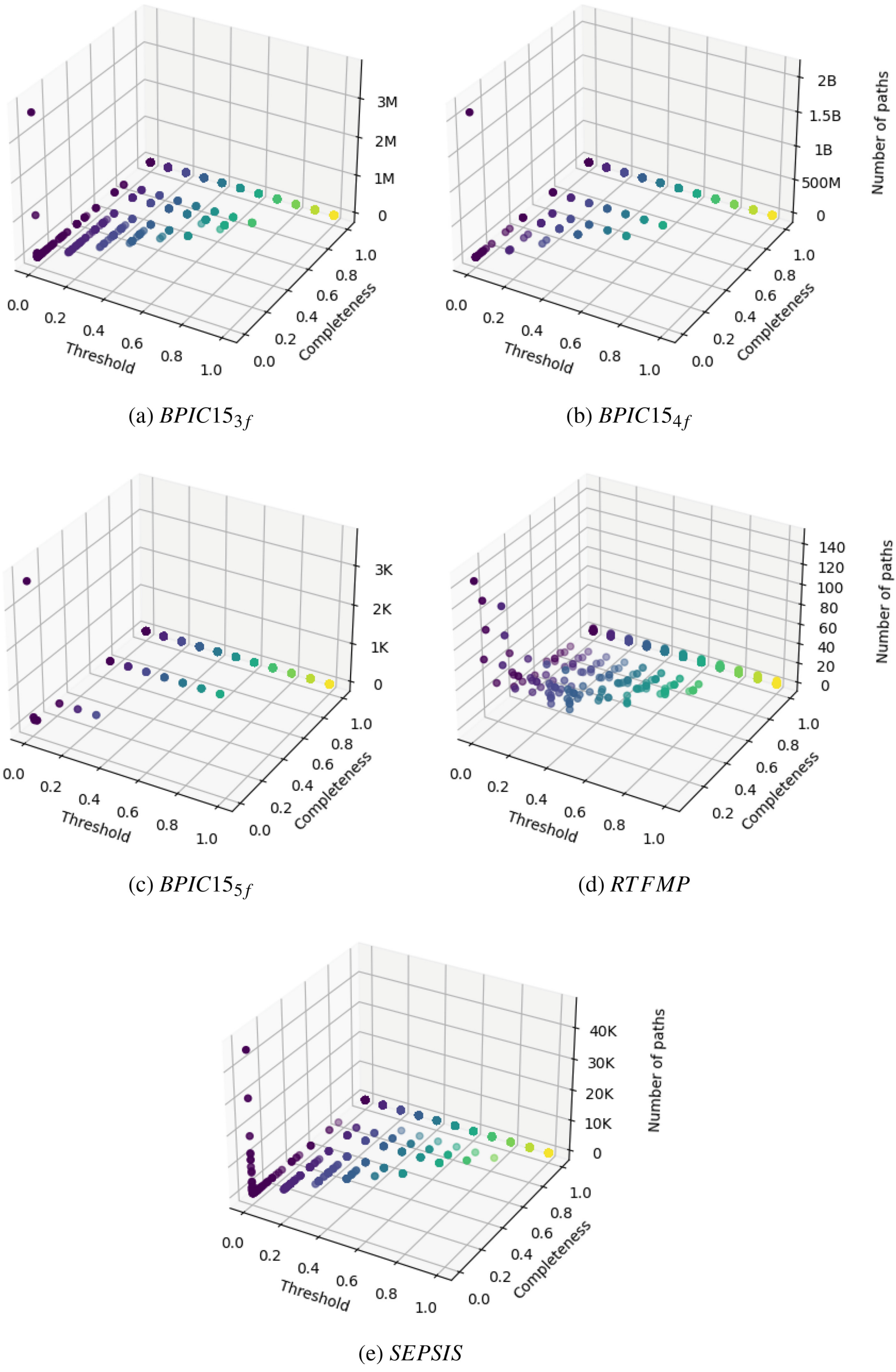


Fig. 8. Thresholds, completeness and number of paths for the lattices per dataset



(a) $BPIC15_{3f}$

(b) $BPIC15_{4f}$

(c) $BPIC15_{5f}$

(d) $RTFMP$

(e) $SEPSIS$

Fig. 9. (2) Thresholds, completeness and number of paths for the lattices per dataset

the construction and completion of the distributive lattices can be a trace that should be forbidden by the process.
Process model discovery from partial orders: A promising direction for future work is to define automated model discovery techniques that take partial orders as input and generate process models with guarantees (e.g., free-choice sound workflow nets or models in BPMN notation). Some works moving in this direction are [3,4,14]. Currently, the majority of automated process model discovery techniques take event logs as input and derive concurrency relations over the activities, rather than over the events (e.g., [2,11]).
Generate unseen behavior as event log traces: Once the distributive lattices have been constructed, it is possible to compute all traces they represent. While the generation of the traces is trivial, in this paper we only consider control flow information and do not take into account other possible event attributes. As a future work, we will explore the possible attributes that can be derived/extrapolated during the generation of the traces from the distributive lattices.

Finally, another promising direction for future work is to consider other types of lattices (e.g., semi-modular lattices) and the Petri net classes to which they correspond.

5 Related Work

The closest related work is that in van der Aalst et al. [19], where different strategies for constructing transitions systems from event logs are presented. Once the transition systems are constructed, the authors put forward the idea of adding new edges between states in the transition systems (called "extend" strategy) as a way to discover behaviour that was not observed in the log but was likely to be present in the process. Our work differs from such approach in two ways. First, we adopt a well-known formalism, distributive lattices, with a well-defined notion of completeness. Thus, when introducing new behaviour in the distributive lattices, it is possible to determine when all missing behaviour has been added, which is not the case when using transition systems. Furthermore, the extend strategy in [19] can only add missing edges between pairs of states, but in the case of distributive lattices, it is possible to discover missing states as well as edges.

Another related work is that on concurrency oracles. [9] uses concurrency oracles to transform event log traces into elementary event structures. In such case, the amount of behaviour added when inserting concurrency will depend on the quality of the oracle. In particular, [9] uses the alpha relations [18] as oracle, which deems a pair of events as concurrent if the activities were ever observed in different orders in the log. For example, alpha relations deem (a, b) concurrent if a is executed before b in one trace, and b is executed before a in another trace. These relations can be spurious because it is possible that the order between a and b is particular to some executions (e.g., a is always executed before b in

the trace $\langle c, a, b\rangle$; whereas b is always executed before a in the trace $\langle f, b, a\rangle$). In order to address that issue, [1] put forward the idea of local concurrency oracles that find concurrency relations that only apply to particular areas of the traces. Such technique uses two threshold, arbitrarily defined by the user, to control the sensitivity of the oracle. Different from the concurrency oracle approaches, the use of distributive lattices gives a reference as to what is the missing behaviour without having to rely on arbitrary thresholds or inserting concurrency derived from distinct computations.

The problem of measuring the completeness of an event log is not new. Probabilistic approaches to measure log completeness can be found in [13,21,23]. Different from these approaches, our aim is to discover the behaviour that is not observed but likely to be present in the process (under some assumptions). Instead, the probabilistic approaches aim at computing a lower bound representing the completeness of a log. While informative, it does not allow us to obtain the missing behavior.

While there are few techniques that compare directly to the approach presented in the paper, process mining operations, and in particular automated process discovery techniques, implicitly discover behavior when abstracting the behavior in a log (e.g., when creating models [2,11]). In fact, a way to assess the quality of a discovered process model is by measuring its generalization, behavior that is not observed in the log but likely to be part of the process [17]. The approach presented in this paper can be seen as a pre-processing step that discovers concurrency from groups of equivalent traces, which can then be inserted during the construction of other more sophisticated models, such as models in BPMN notation.

6 Conclusion

This paper presented an approach to discover unseen behavior from an event log. The approach is based on the reconstruction of distributive lattices, which represent execution states of process instances. This approach was inspired by the results presented in [12], where the relationship between a family of Petri nets (conflict-free 1-safe nets), event structures and lattices was shown. There, the authors showed that the lattices representing the execution states of the Petri nets were distributive when ordered by subset inclusion. The approach presented in this paper starts by computing lattices from groups of traces representing the same computation. Then, these lattices are completed by inserting missing elements until the distributivity property is fulfilled. Using the discovered behavior, a measure of completeness is defined to assess the volume of traces discovered during the completion operation. Finally, it was shown how this measure of completeness can be used to control the amount of behavior discovered.

In order to test the effect of the proposed approach in real-life logs, a set of experiments were run to measure the amount of behavior discovered in a collection of publicly available event logs. It was observed that in some event logs, the number of traces that could be computed from the distributive lattice

could be very large. The latter may be due to noise in the log, which can lead to the discovery of too much behavior that may not be part of the "normal" process. Thus, while the completion operation is simple, it can be too sensitive to noise. Nonetheless, by using the measure of completeness as threshold, it is possible to control the amount of discovered behavior.

Acknowledgements. The authors would like to thank all the reviewers for their valuable comments. Special thanks to reviewer 2 for their insightful suggestions.

References

1. Armas-Cervantes, A., Dumas, M., Rosa, M.L., Maaradji, A.: Local concurrency detection in business process event logs. ACM Trans. Internet Technol. **19**(1), 1–23 (2019)
2. Augusto, A., Conforti, R., Rosa, M.L., Dumas, M.: Split miner: discovering accurate and simple business process models from event logs. In: 2017 IEEE International Conference on Data Mining (ICDM), pp. 1–10 (2017)
3. Bergenthum, R.: Prime miner - process discovery using prime event structures. In: 2019 International Conference on Process Mining (ICPM), pp. 41–48 (2019)
4. Bergenthum, R., Desel, J., Mauser, S., Lorenz, R.: Synthesis of petri nets from term based representations of infinite partial languages. Fundam. Inf. **95**(1), 187–217 (2009)
5. Birkhoff, G.: On the combination of subalgebras. Math. Proc. Cambridge Philos. Soc. **29**(4), 441–464 (1933)
6. Conforti, R., Rosa, M.L., ter Hofstede, A.H.M.: Filtering out infrequent behavior from business process event logs. IEEE Trans. Knowl. Data Eng. **29**(2), 300–314 (2017)
7. Dumas, M., García-Bañuelos, L.: Process mining reloaded: event structures as a unified representation of process models and event logs. In: Devillers, R., Valmari, A. (eds.) PETRI NETS 2015. LNCS, vol. 9115, pp. 33–48. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19488-2_2
8. Fani Sani, M., van Zelst, S.J., van der Aalst, W.M.P.: Repairing outlier behaviour in event logs. In: Abramowicz, W., Paschke, A. (eds.) BIS 2018. LNBIP, vol. 320, pp. 115–131. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93931-5_9
9. García-Bañuelos, L., van Beest, N.R.T.P., Dumas, M., Rosa, M.L., Mertens, W.: Complete and interpretable conformance checking of business processes. IEEE Trans. Softw. Eng. **44**(3), 262–290 (2018)
10. Habib, M., Nourine, L.: Tree structure for distributive lattices and its applications. Theoret. Comput. Sci. **165**(2), 391–405 (1996)
11. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs - a constructive approach. In: Colom, J.-M., Desel, J. (eds.) PETRI NETS 2013. LNCS, vol. 7927, pp. 311–329. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38697-8_17
12. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, Part I. Theor. Comput. Sci. **13**, 85–108 (1981)
13. Pei, J., Wen, L., Yang, H., Wang, J., Ye, X.: Estimating global completeness of event logs: a comparative study. IEEE Trans. Serv. Comput. **14**(2), 441–457 (2021)

14. Ponce-de-León, H., Rodríguez, C., Carmona, J., Heljanko, K., Haar, S.: Unfolding-based process discovery. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) ATVA 2015. LNCS, vol. 9364, pp. 31–47. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24953-7_4
15. Reisig, W.: Petri Nets: An Introduction. Springer, Heidelberg (1985)
16. Stone, M.H.: The theory of representation for boolean algebras. Trans. Am. Math. Soc. **40**(1), 37–111 (1936)
17. van der Aalst, W.: Process Mining: Data Science in Action, 2nd edn. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49851-4
18. van der Aalst, W.M.P., Weijters, A.J.M.M., Maruster, L.: Workflow mining: discovering process models from event logs. IEEE Trans. Knowl. Data Eng. **16**(9), 1128–1142 (2004)
19. Van der Aalst, W.M.P., Rubin, V., Verbeek, H.M.W., van Dongen, B.F., Kindler, E., Günther, C.W.: Process mining: a two-step approach to balance between underfitting and overfitting. Softw. Syst. Model. **9**(1), 87–111 (2010)
20. van der Aalst, W.M.P., Weijters, A.J.M.M.: Process mining: a research agenda. Comput. Ind. **53**(3), 231–244 (2004)
21. van Hee, K.M., Liu, Z., Sidorova, N.: Is my event log complete? - a probabilistic approach to process mining. In: International Conference on Research Challenges in Information Science, pp. 1–12 (2011)
22. Wang, J., Song, S., Lin, X., Zhu, X., Pei, J.: Cleaning structured event logs: a graph repair approach. In: 2015 IEEE 31st International Conference on Data Engineering, pp. 30–41 (2015)
23. Yang, H., Ter Hofstede, A.H., Van Dongen, B.F., Wynn, M.T., Wang, J.: On global completeness of event logs. BPM Center Report BPM-10-09 (2010)