Local Concurrency Detection in Business Process Event Logs

Abel Armas Cervantes*[1], Marlon Dumas[2], Marcello La Rosa*[3], and Abderrahmane Maaradji*[4]

[1,3]The University of Melbourne, Australia
[2]University of Tartu, Estonia
[4]University of Algiers 1, Algeria

October 25, 2018

Abstract

Process mining techniques aim at analysing records generated during the execution of a business process in order to provide insights on the actual performance of the process. Detecting concurrency relations between events is a fundamental primitive underpinning a range of process mining techniques. Existing approaches to this problem identify concurrency relations at the level of event types under a global interpretation. If two event types are declared to be concurrent, every occurrence of one event type is deemed to be concurrent to one occurrence of the other. In practice, this interpretation is too coarse-grained and leads to over-generalization. This paper proposes a finer-grained approach, whereby two event types may be deemed to be in a concurrency relation relative to one state of the process, but not relative to other states. In other words, the detected concurrency relation holds locally, relative to a set of states. Experimental results both with artificial and real-life logs show that the proposed local concurrency detection approach improves the accuracy of existing concurrency detection techniques.

1 Introduction

Process mining is a body of techniques that help analysts understand business processes based on their event logs. In this context, an event log is a set of traces, each consisting of a sequence of events with associated attributes. Each event is an instance of an event type, corresponding to an activity in the business process. For example, an event log of an order-to-cash process may include event types such as "Goods shipped" and "Payment collected". An event of type

*This work was partly conducted when the author was at Queensland University of Technology

"Payment collected" may include additional attributes such as the confirmation number and the amount. Given an event log, process mining tools can extract a process model (*automated process discovery*), check the conformance of a given process model against the log (*conformance checking*), compare two event logs (*log delta analysis*), or detect changes in the execution of a process over time (*drift detection*), among other analysis operations.

A number of process mining techniques abstract the observed behavior as descriptive models reflecting how the actual work is being carried out. Some of those models focus on the description of the dependencies between events as behavioral relations, most notably causality relations (the occurrence of an event entails the subsequent occurrence of another one), conflict relations (the occurrence of an event excludes the occurrence of another) and interleaved concurrency relations (two events co-occur in any order). A key challenge in this context is how to accurately distinguish between events that always occur in a given order (the execution of an event depends on the execution of another) and events that can occur in any order (independent events) – causality vs. concurrency. This challenge arises in particular in the context of knowledge-driven processes, which are characterized by high levels of variation in the ordering of activities. In these processes, it is common to find states where multiple activities can be executed in any order and any number of times [6, 17]. At the same time, there are constraints and norms that result in some activities being performed (almost) always in a given order under certain conditions. For example, in a process for handling permitting applications, a number of verifications and approvals need to take place. When the application is of a certain type and is lodged in a given jurisdiction, some of these verifications always occur in a given order, while for other types of applications and jurisdictions, the order of these activities is unconstrained. Similarly, in a clinical process, when a patient is diagnosed with a common condition, a number of tests and treatments take place in a relatively fixed order. But for other diagnoses or in the absence of a definitive diagnosis, these tests and treatments are applied in any order. Existing process mining techniques are not designed to take into account these variations in causality and concurrency relations. Instead, when such variations exist, the activities in question are declared as parallel activities without further refinement.

Indeed, existing approaches to discovering concurrency – e.g. those embedded in the α process discovery algorithm and its variants [3, 23, 33, 20] – detect *global concurrency relations* at the level of pairs of event types. The semantics of a global concurrency relation between two event types is that an instance of the first type must be either followed or preceded by an instance of the second type regardless of where in the log these instances occur. In practice, this property does not always hold. For example, consider a log recording the executions of the process for plan lodgement and document registration in two different Australian states, South and Western Australia, whose model is shown in Fig. 1.[1] A global concurrency discovery approach – herein called a *global concurrency oracle* – would assert that event types "Update register" and "Update DCDB",

[1] This model forms part of a collection of real-life process models for handling land development applications in Australia.

and event types "Approve plan" and "Update register", among others, are concurrent. However, "Approve plan" and "Update register" are concurrent only in the case of Western Australia (i.e. when the WA path after the decision point is taken), while "Update register" and "Update DCDB" are never concurrent. This approximation then affects the precision of the process mining techniques. For example in the context of automated process discovery, the result is likely to be a model where activities "Update register" and "Update DCDB" can always occur in any order regardless of the state.

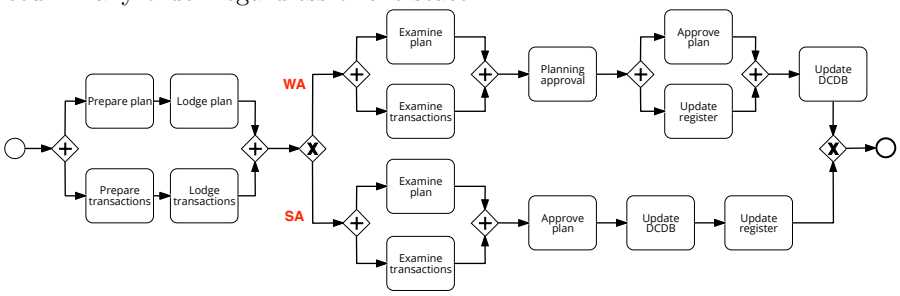


Figure 1: Process model for plan lodgement and document registration in Western Australia (WA) and South Australia (SA).

Two event types may erroneously be considered as concurrent, not only if they occur in conflicting traces as in the example of Fig. 1, but also if they occur in different execution states within traces. For instance, consider the traces $\langle C, D, E, F, G, H, L, A, B\rangle$ and $\langle B, A, C, D, E, F, G, H, L\rangle$. These traces have the same event types and even the same order between C, D, F, G, H, L, but the order between A and B depends on where these two events occur. Specifically, A occurs before B only if they are executed after L, and vice-versa B occurs before A if they both occur at the beginning of the trace. While A and B are clearly not concurrent in this example, they will still be considered as such under the semantics of a global concurrency relation. This situation arises even within a single trace, for instance if the trace contains two pairs of events of the same type, occurring in a different order in two different locations of the trace (e.g. any trace of the form $\langle \dots A\ B \dots B\ A \dots \rangle$).

As shown in the above examples, a global concurrency oracle tends to *over-generalize* the behavior captured in a log by allowing the execution of sets of event types (and so process activities) in any order throughout the log, even when such activities are in fact not concurrent, or are such only in certain scopes of the log. This leads to a more permissive representation of the log behavior.

This paper advocates an alternative *local concurrency* detection approach whereby a concurrency relation between two event types is scoped to certain execution states of the process. The main contribution is an approach that turns any global concurrency oracle into a scoped (local) one. The key idea is to construct a state transition graph from the event log and to traverse this graph in order to discover concurrency relations (using an existing concurrency oracle) in-between pairs of states. The accuracy of the proposed local concurrency detection approach is compared against the α global concurrency oracle

and inductive miner [18], a well-known process discovery algorithm, based on a synthetic testbed comprising a range of combinations of control-flow structures, as well as seven real-life logs. We also show on a real-life process model that the proposed concurrency oracle can improve the accuracy of an existing business process drift detection technique.

The rest of the paper is structured as follows. Section 2 discusses existing approaches to construct concurrency oracles and their limitations. Section 3 introduces the proposed approach, while Sections 4-6 present its experimental evaluation and Section 7 discusses threats to validity. Finally, Section 8 summarizes the results and outlines future work directions.

2 Related work

Several techniques have approached the problem of discovering behavioral relations (in particular concurrency relations) between pairs of event types. For instance, [8] outlines a technique based on statistical measures to discover event-based models that capture the concurrent execution of events types. This latter technique is however highly dependent on the quality of the log. In particular, it assumes that concurrency is embedded in blocks that have a single split and a single join event, and that the order of occurrence of the concurrent event types is uniformly distributed. The α-algorithm [3] and its variants [23, 33, 20] detect concurrency relations (among other behavioral relations) between event types. The α-algorithm itself declares a pair of event types to be concurrent if one immediately precedes the other and vice-versa. Both mentioned approaches, α-algorithm and that presented in [8], lead to global concurrency oracles, which as stated before, disregard the context where the events occur, thus leading to false positives.

Extensions of the α-algorithm such as $\alpha+$ [23] are designed to prevent the α-algorithm from confusing concurrency with (short) loops and other limitations, however, they still suffer from the limitation of being global.

In [10], the relations computed by the α-algorithm (referred to as α relations hereinafter) are used as a concurrency oracle to construct a partially ordered run (therein called an *instance graph*) from each trace in an event log. The resulting set of runs can be used to synthesize a process model (e.g. a Workflow net) [11]. This latter approach however inherits the limitations of the α-algorithm as a method for constructing concurrency oracles. A more recent approach to construct partially ordered runs [9] from traces addresses the issue of discovering concurrency in the presence of infrequent event types. The starting point is still a process discovery algorithm that incorporates a global concurrency oracle. Traces are turned into partially ordered runs based on the global oracle and then adjusted to take into account infrequent event types.

In the context of process model synthesis, some techniques require additional data for computing concurrency relations. For example, the approach in [19] requires that a log contains the start and end timestamps of every event, and then a pair of events are concurrent if they overlap in time. However, the data about the start and end timestamps of an event is not always available in the event logs.

A technique to discover scoped concurrency relations between events is proposed in [25]. Given an event log, this technique produces a conditional partial order graph [26]. In this graph, a concurrency relation is scoped by means of (data) conditions, i.e. the concurrency relation only holds when the condition evaluates to true at a given point in the process. A condition is determined by the execution of an event, e.g. a and b are concurrent if c is executed, noting that c does not necessarily need to occur before a and b. However, this technique makes the highly restrictive assumption that there are no two events of the same type in the same trace, since when a duplicate event is found in a trace, the trace is split and the two sub-traces are treated as two different traces, leading to the possibility of identifying concurrency across these traces.

In principle, an alternative approach to detecting local concurrency from a log could be to use trace clustering techniques to group traces into clusters, and apply a global concurrency oracle within each cluster. In the context of process mining, there exist several techniques that have approached the problem of clustering traces in an event log according to different notions of similarity, e.g. [5, 4, 28, 29, 24, 15, 32]. However, there are two issues with this approach. First, trace clustering is very heuristic in nature (e.g. [28, 29, 24] implement k-means and self-organizing maps), thus the discovered clusters can contain traces of different executions, which may lead to compute non-existent concurrency relations (false positives). As an example, one of the latest trace clustering techniques, ActiTraC [32], applied over the log generated by the land development model of Fig. 1, cannot distinguish all the traces related to Western Australia from those related to Southern Australia (specifically, three traces of Western Australia are put in the same cluster with all the traces of South Australia), leading to the detection of spurious concurrency relations.[2]

The second issue is that trace clustering disregards the scopes where event types occur. Consider the log $\{\langle C, D, E, F, G, A, B, H, L\rangle, \langle C, D\ , E, F, G, H, L, A, B\rangle, \langle B, A, C, D, E, F, G, H, L\rangle, \langle V, W, T, S, H, L, A, B\rangle, \langle V, W, T, S, A, B, H, L\rangle\}$, which is a more elaborated example of the one shown in the Introduction. In this log, the first three traces contain the same event types and are part of the same clusters according to [32],[28, 29] and [16]. If we used a global oracle like the α, this would regard A and B as concurrent, since both interleavings are present in the first three traces. However, the occurrence of B followed by A only occurs at the beginning of a trace while that of A followed by B occurs after G in the rest of the traces. Thus, the scope where each of the interleavings occur is different, and the event types should not be diagnosed as concurrent. This is because trace clustering techniques and our approach are fundamentally different. Trace clustering is *slicing* operation over a log [1], which divides the log into groups of traces that can be analyzed as a whole, while our approach can be seen as a *dicing* operation over the log [1], where the log is divided vertically into different scopes across groups of traces or within the same trace, within which local concurrency is identified.

[2]The settings used for this ActiTrac in ProM were Distance metric = Euclidian and Clustering Algorithm = Quality Threshold Clustering, while the rest of the parameters were left with their default values.

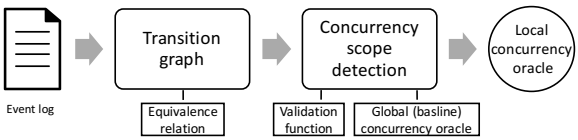


Figure 2: Proposed approach.

3 Discovering local concurrency

An overview of the proposed approach is given in Fig. 2. The first step consists in creating a transition graph that represents the information captured in an event log. This graph has one initial state and at least one final state, and every path from the initial to a final state is a trace in the log. The step requires an equivalence relation ($\equiv$) to find similar execution states in the process. Given the transition graph, the second step identifies concurrency relations that hold between two states. This step takes as input a global concurrency oracle ($\mathcal{C}$) and a validation function ($\mathcal{F}$). $\mathcal{C}$ can be for example the α-algorithm or other oracles mentioned in the previous section. Since a given concurrency relation may hold to various degrees in multiple scopes, including pairs of scopes such that one contains the other, it is necessary to select its most suitable scope. To this end, $\mathcal{F}$ is used to assess the likelihood of the computed concurrency relations. Note that the equivalence relation, the global concurrency oracle and the validation function can be customized; however, in this paper we present a single configuration of these three elements.

This section starts by presenting the construction of the transition graph and then the computation of the scopes.

3.1 Transition graph of an event log

The first step of our approach consists in constructing a transition graph representing the behavior captured in the event log, such that every transition in the graph represents the execution of an event type (event) and every state represents the occurrence of some events. This event log abstraction has been widely used in the context of process mining. For instance, [2] presents several strategies for the construction of a transition graph that can be modified to vary the degree of generalization. However, the approach presented in [2] aims at constructing a transition graph from which a model is synthesized using the theory of regions, and considers steps where transitions are added to or removed from the transition graph. Different from [2], our approach neither adds nor removes transitions in such graph.

Before presenting the construction of a transition graph, we define some notations on sequences, traces and event logs.

Definition 1 (Sequences, subsequences and extension)**.** *Let X be a set of elements. A sequence of length $n \in \mathbb{N}$ over X is denoted as $\sigma = \langle a_1\ a_2\ a_3\ \ldots\ a_n \rangle \in X^*$, whereas an* empty *sequence is denoted as ϵ. The length of a sequence σ is represented as $|\sigma|$.*

Given a sequence $\sigma = \langle a_1\ a_2\ a_3\ \ldots\ a_n \rangle$, the prefix subsequence *of length m, shorthanded as $\sigma_{[1\ldots m]}$, is another sequence composed by the first m elements,*

i.e., $\sigma_{[1...m]} = \langle a_1\ a_2\ a_3\ \ldots\ a_m \rangle$ *such that* $1 \leq m \leq n$*; whereas the prefix of an empty sequence* ϵ *is another empty sequence* ϵ'*. The set of all prefix subsequences of* $\sigma = \langle a_1\ a_2\ a_3\ \ldots\ a_n \rangle$ *is represented as* $\phi(\sigma) = \{\sigma_{[1...k]} \mid 1 \leq k \leq n\}$*. A prefix subsequence* $\sigma' = \langle a_1\ \ldots a_l \rangle$ *of a sequence* σ *can be extended with an element* e_x*, denoted as* $\sigma' \oplus e_x$*, if* $\sigma' \oplus e_x = \langle a_1\ \ldots a_l\ e_x \rangle$ *is another prefix subsequence in* $\phi(\sigma)$*.*

The suffix subsequence *of length* m *is the sequence composed by the last* m *elements of* σ*, and it is denoted as* $\sigma_{[m,n]} = \langle a_{(n-m)+1}\ \ldots a_{n-1}\ a_n \rangle$*,* $1 \leq m \leq n$*.* ⌟

An event log is a set of sequences of event types occurrences, a.k.a. traces.[3]

Definition 2 (Trace, Event log)**.** *Let* E *be a set of events,* Λ *be a set of event types, and* $\lambda : E \to \Lambda$ *be a labelling function. A* trace *is a sequence of events* $\sigma = \langle \lambda(e_1), \lambda(e_2), \ldots, \lambda(e_n) \rangle$ *for* $e_i \in E$*,* $1 \leq i \leq n$*, while an* event log $\mathcal{L}$ *is a set of traces.* ⌟

A trace σ can contain several events of the same type and, in order to distinguish one another, we adopt the following convention: every event is unique within a trace and identified by a label and an index. The type of the event is the label, and the occurrence number of this event type within the trace is the index. For instance, the trace $\langle a\ b\ b \rangle$ is composed by one occurrence of activity a and two occurrences of activity b, thus the events in the trace would be $\langle a_1\ b_1\ b_2 \rangle$. By the abuse of notation, we refer to any generic event as e_i where $i \in \mathbb{N}$. If the order amongst the events is of no interest, then a trace can be represented as a set, thus we use $\widehat{\sigma}$ to denote the set representation of σ.

Definition 3 (Event and trace equivalence)**.** *Let* $\sigma = \langle e_1\ e_2 \ldots e_m \rangle$ *and* $\sigma' = \langle e'_1\ e'_2 \ldots e'_n \rangle$ *be a pair of traces,* e_i *be an event in* σ*, for* $1 \leq i \leq m$*, and* e' *be an event in* σ'*, for* $1 \leq j \leq n$*. The events* e_i *and* e'_j *are* equivalent*, denoted as* $e_i \sim e_j$*, iff they represent the same occurrence of the same event type, i.e.,* $i = j$ *and* $\lambda(e_i) = \lambda(e_j)$*. Two traces are* equivalent*, denoted as* $\sigma \sim_{order} \sigma'$*, iff* $m = n$ *and for every* $1 \leq i \leq n$ *then* $e_i \sim e'_i$*. Finally, two traces are* set equivalent*, denoted as* $\sigma \sim_{set} \sigma'$*, if their two set representations contain equivalent events, i.e.,* $\forall e_i \in \widehat{\sigma}\ \exists e'_j \in \widehat{\sigma'} : e_i \sim e'_j$ *and* $\forall e'_j \in \widehat{\sigma'}\ \exists e_i \in \widehat{\sigma} : e_i \sim e'_j$*.*

Intuitively, a pair of traces are order equivalent if they have equivalent events and the order among them is the same; whereas they are set equivalent if they contain equivalent events but not necessarily in the same order.

Every event in a trace has a past and a future. Consider a trace $\sigma = \langle e_1\ e_2 \ldots e_n \rangle$ and an event e_i where $1 < i \leq n$, $\lfloor e_i \rfloor$ signifies the past of e_i and is the prefix subsequence of σ up to i, i.e., $\lfloor e_i \rfloor = \sigma_{[1...i-1]}$, while $\lceil e_j \rceil$ signifies the future of an event e_j, where $1 \leq j < n$, and is the suffix subsequence after j, i.e., $\lceil e_j \rceil = \sigma_{[j+1,n]}$. The prefix (suffix) subsequence of the event e_1 (resp. e_n) is the empty sequence ϵ. For instance, given the trace $\langle i_1\ b_1\ c_1\ d_1\ o_1 \rangle$, the past of b_1 is $\lfloor b_1 \rfloor = \langle i_1 \rangle$ and its future is $\lceil b_1 \rceil = \langle c_1\ d_1\ o_1 \rangle$.

[3]Generally speaking, an event log is a multiset of traces, however we focus simply on the ordering of events and disregard the information about the number of times each trace occurs in the log. Furthermore, for simplicity, we assume that every trace is a complete execution and the log is noise free.

A trace describes the evolution of an execution of a system by means of its prefix subsequences and their extensions. Thus, a trace can be represented as a transition graph every pair of execution states (set representation of a prefix subsequence) are linked by an extension, which transforms one state into the other. Formally, a transition graph is the tuple $\langle V, v_i, W, E, T\rangle$, where V is the set of states, v_i is the initial state, W is the set of final states, E is the set of events, and T is a transition relation. The next definition suggests how to construct a transition graph from a trace.

Definition 4 (Transition graph of a trace)**.** *Let $\sigma = \langle e_1\ e_2\ \ldots\ e_n\rangle$ be a trace. The transition graph representing σ is defined as $\langle V, \varnothing, \{\widehat{\sigma}\}, E, T\rangle$, where*

$$\begin{array}{lcl} V & = & \{\widehat{\sigma'} \mid \sigma' \in \phi(\sigma)\} \\ E & = & \widehat{\sigma} \\ T & = & \{(\widehat{\sigma_1}, e_i, \widehat{\sigma_2}) \mid e_i \in E \ \wedge\ \sigma_1 = \lfloor e_i \rfloor \ \wedge\ \sigma_2 = \lfloor e_i \rfloor \oplus e_i\} \end{array} \qquad \lrcorner$$

Consider the log $\mathcal{L} = \{\langle i_1\ b_1\ c_1\ d_1\ o_1\rangle, \langle i_1\ a_1\ c_1\ d_1\ f_1\ o_1\rangle, \langle i_1\ a_1\ d_1\ c_1\ f_1\ o_1\rangle\}$, the transition graphs representing the traces in $\mathcal{L}$ are shown in Fig. 3. Observe that some traces can represent interleavings of the same execution and thus similar execution states. For instance, the two transition graphs at the bottom of Fig. 3 can be seen as interleavings of an execution where c_1 and d_1 are concurrent, which would imply that the states $\{i_1\}, \{i_1, a_1\}, \{i_1, a_1, c_1, d_1\}, \{i_1, a_1, c_1, d_1, f_1\}$ and $\{i_1, a_1, c_1, d_1, f_1, o_1\}$ are somehow similar. Indeed by treating these states as similar executions, the concurrency relation between c_1 and d_1 would appear as a diamond in the corresponding transition graph – see grayed out states in the transition graph depicted in Fig. 4 – denoting the possible (interleaved) concurrent execution of such events.

$$\varnothing \xrightarrow{i_1} \{i_1\} \xrightarrow{b_1} \{i_1, b_1\} \xrightarrow{c_1} \{i_1, b_1, c_1\} \xrightarrow{d_1} \{i_1, b_1, c_1, d_1\} \xrightarrow{o_1} \{i_1, b_1, c_1, d_1, o_1\}$$

$$\varnothing \xrightarrow{i_1} \{i_1\} \xrightarrow{a_1} \{i_1, a_1\} \xrightarrow{c_1} \{i_1, a_1, c_1\} \xrightarrow{d_1} \{i_1, a_1, c_1, d_1\} \xrightarrow{f_1} \{i_1, a_1, c_1, d_1, f_1\} \xrightarrow{o_1} \{i_1, a_1, c_1, d_1, f_1, o_1\}$$

$$\varnothing \xrightarrow{i_1} \{i_1\} \xrightarrow{a_1} \{i_1, a_1\} \xrightarrow{d_1} \{i_1, a_1, d_1\} \xrightarrow{c_1} \{i_1, a_1, d_1, c_1\} \xrightarrow{f_1} \{i_1, a_1, d_1, c_1, f_1\} \xrightarrow{o_1} \{i_1, a_1, d_1, c_1, f_1, o_1\}$$

Figure 3: Transition graphs representing three different traces.

We do not require the transition graph to meet any specific property, nor do we assume the log to be complete with respect to the α relations (i.e. we do not expect all concurrency interleavings to be present). Instead, we extract all the behavior represented in the log and detect patterns of concurrency from within this behavior.

We now turn our attention to the definition of an equivalence relation between states of transition graphs. As hinted previously, the equivalence relation collapses sets of "similar" states, which can introduce some generalization and, by the same token, discover patterns reflecting the concurrent execution of events. The equivalence relation between states defined in this section is grounded on the most primitive interpretation of the (interleaved) concurrent execution of a pair of events. Specifically, starting from an execution state, a pair of concurrent events can occur in any order and lead to the same execution state. The latter is the essence of the results presented in [27], where the

authors show that a transition graph-like representation (*domain of configurations*) can represent the true-concurrency semantics of a system, where the concurrent execution of a pair of events is manifested as diamond-like shapes.

The main idea of the equivalence relation presented below is to construct a transition graph where every state is associated to a unique set of events: the events that have occurred before the state is reached. Each transition is labeled by an event and connects a pair of states, such that the set of events in the *source* state is a strict subset of the set of events of the *target* state. We distinguish two special types of nodes in a transition graph, a unique initial state $\varnothing$, and at least one final state (state with no outgoing transitions) representing an execution of a process.

Figure 4 shows an example of the type of transition graph that we seek to extract from a log. The graph represents two executions: $\{i_1, a_1, c_1, d_1, f_1, o_1\}$ and $\{i_1, b_1, c_1, d_1, o_1\}$. In the graphical representation, the sets of events denote states and the events associated to the transitions are displayed aside. Note that in the case of $\{i_1, a_1, c_1, d_1, f_1, o_1\}$, there is a diamond (cf. gray nodes in Fig. 4) representing the possible concurrent execution of events c_1 and d_1. These diamonds define the *scopes* where pairs of events are executed concurrently, and thus give place to a notion of local concurrency relations. The bottom and top states of a diamond are referred to as start and end of the scope, and they represent the states where no concurrent event has been executed and the state where all the concurrent events have taken place, respectively. For example, in Fig. 4, the states $\{i_1, a_1\}$ and $\{i_1, a_1, c_1, d_1\}$ define the start and end of the scope, respectively, where c_1 and d_1 are concurrent. Note that the concurrency relation between c_1 and d_1 does not hold for the other occurrences on the right hand side.

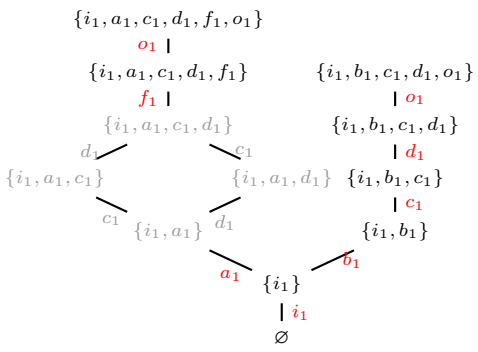


Figure 4: Transition graph of the log $\mathcal{L} = \{\langle i_1\ b_1\ c_1\ d_1\ o_1\rangle, \langle i_1\ a_1\ c_1\ d_1\ f_1\ o_1\rangle, \langle i_1\ a_1\ d_1\ c_1\ f_1\ o_1\rangle\}$

The equivalence relation defined below deems a pair of states as equivalent if they represent the occurrence of equivalent events and, either they were executed in the same order (they are essentially the same event type occurrences) or the same set of events (in the same order) can occur from both states.

Definition 5 (Transition graph equivalence)**.** *Given a pair of traces σ and σ' and their corresponding transitions graphs $G = \langle V, v_i, \{\widehat{\sigma}\}, E, T\rangle$ and $G' = \langle V', v_i', \{\widehat{\sigma'}\}, E', T'\rangle$. Let $\sigma_1 = \sigma_{[1\ldots m]}$ and $\sigma_1' = \sigma'_{[1\ldots m]}$ be a pair of subtraces, such that $v = \widehat{\sigma_1}$ and $v' = \widehat{\sigma_1'}$ are the corresponding states in the graphs. The* states in the transition graph are equivalent *— shorthanded as $v \equiv v'$ — if $\sigma_1 \sim_{set} \sigma_1'$ and either of the following hold: (i) $\sigma_1 = \sigma_1' = \epsilon$, (ii) $\sigma_1 \sim_{order} \sigma_1'$, or (iii) $n = |\sigma| = |\sigma'| \ \wedge\ (m = n \ \vee\ \sigma_{[m+1,n]} \sim_{order} \sigma'_{[m+1,n]})$.*

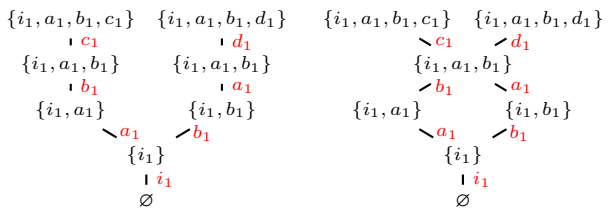
The equivalence relation between states gives place to an equivalence rela-

tion between events in the transition graph. *Consider a pair of transitions* $(v'_1, e_i, v''_1) \in T$ *and* $(v'_2, e_j, v''_2) \in T'$. *Then,* $e_i \equiv e_j$ *iff* $v'_1 \equiv v'_2$ *and* $v''_1 \equiv v''_2$.

The equivalence class of a state v *is defined as* $\langle v \rangle_{\equiv} = \{v' \mid v \equiv v'\}$ *and, similarly for events, the equivalence class of an event* e_i *is* $\langle e_i \rangle_{\equiv} = \{e_j \mid e_i \equiv e_j\}$. ⌟

For example, consider the transition graphs at the bottom of Fig. 3, by Definition 5, the states $\varnothing$ are equivalent by rule (i); states $\{i_1\}$ and $\{i_1, a_1\}$ are equivalent by rule (ii); and states $\{i_1, a_1, c_1, d_1\}, \{i_1, a_1, c_1, d_1, f_1\}$ and $\{i_1, a_1, c_1, d_1, f_1, o_1\}$ are equivalent by rule (iii). Then, the equivalent events are i_1, a_1, f_1 and o_1.

The conditions (ii) and (iii) in the equivalence relation between states (Definition 5) allow us to cope with possible over-generalization during the construction of the transition graph of a log, while condition (i) aims only at merging the initial states of the graphs. On the one hand, condition (ii) merges states representing the execution of events that occurred in the same order, even if those states stem from different traces. On the other hand, condition (iii) merges all states that lead to the same final state. The latter condition targets the cases when the order of a set of events is defined by the execution where they occur. For example, consider the traces $\langle i_1, a_1, b_1, c_1 \rangle$ and $\langle i_1, b_1, a_1, d_1 \rangle$: the order between a_1 and b_1 could depend on the occurrence of either c_1 or d_1. The transition graph constructed out of these two traces is displayed in Fig. 5a. Observe that in order to merge the states $\{i_1, a_1, b_1\}$, it is necessary to have the trace $\langle i_1, b_1, a_1, c_1 \rangle$, in which case the transition graph of the three traces would be the one shown in Fig. 5b.



(a) Transition graph for $\langle i_1, a_1, b_1, c_1 \rangle$ and $\langle i_1, a_1, b_1, d_1 \rangle$

(b) Transition graph for $\langle i_1, a_1, b_1, c_1 \rangle$, $\langle i_1, a_1, b_1, d_1 \rangle$ and $\langle i_1, b_1, a_1, c_1 \rangle$

Figure 5: Transitions graphs produced by Def. 5

Given an equivalence notion $\equiv$, e.g., the one presented in Def. 5, the construction of a transition graph from an event log is presented next.

Definition 6. *Let* $\mathcal{L}$ *be an event log. The transition graph of* $\mathcal{L}$ *is defined as* $G = \langle V, v_i, W, E, T \rangle$, *where*

$$
\begin{array}{lcl}
V & = & \{\langle v \rangle_{\equiv} \mid v = \widehat{\sigma'} \ \wedge \ \sigma' \in \phi(\sigma)\} \\
v_i & = & \varnothing \\
W & = & \{\langle \widehat{\sigma} \rangle_{\equiv}\} \\
E & = & \{\langle e_i \rangle_{\equiv} \mid 1 \leq i \leq |\sigma|\} \\
T & = & \{(\langle v_1 \rangle_{\equiv}, \langle e_i \rangle_{\equiv}, \langle v_2 \rangle_{\equiv}) \mid \exists \sigma', \sigma'' \in \phi(\sigma), e' \in \langle e_i \rangle_{\equiv} \ : \ v_1 = \widehat{\sigma'} \ \wedge \ v_2 = \widehat{\sigma''} \ \wedge \\
 & & v_2 = v_1 \cup \{e'\}\}
\end{array}
$$

for all traces σ *in* $\mathcal{L}$. ⌟

The transition graph representing the event log $\mathcal{L} = \{\langle i_1\ b_1\ c_1\ d_1\ o_1\rangle, \langle i_1\ a_1\ c_1\ d_1\ f_1\ o_1\rangle, \langle i_1\ a_1\ d_1\ c_1\ f_1\ o_1\rangle\}$ constructed with the equivalence in Def. 5 is that of Fig. 4.

3.2 Discovering scopes of concurrency

Once the transition graph is constructed, the second step consists of turning an existing global concurrency oracle into a scoped (local) one. Specifically, the aim is to discover parts of a transition graph where concurrency relations between events are likely to hold. We refer to this parts of the transition graph as scopes and denote them as $\mathcal{S}$. The approach requires a (global) concurrency oracle $\mathcal{C}$, which computes a set of relations from a given set of traces, and a validation function $\mathcal{F} : \mathcal{S} \times (E \times E) \to \{true, false\}$ that, given a concurrency relation in a scope, retrieves a boolean value representing the outcome of the validation.

In a transition graph there is a path between a pair of states v_1 and v_2 iff there is a sequence of transitions $\langle (v_1, e_1, v_a)\ (v_a, e_2, v_b) \dots (v_x, e_x, v_2)\rangle$. Let $\pi(v_1, v_2) = \langle (v_1, e_1, v_a)\ (v_a, e_2, v_b) \dots (v_x, e_x, v_2)\rangle$ represent a path between v_1 and v_2, $\pi(v_1, v_2)|_E = \langle e_1\ e_2 \dots e_x\rangle$ represent the sequence of events in the path, and $\pi(v_1, v_2)|_{\lambda(E)} = \langle \lambda(e_1)\ \lambda(e_2) \dots \lambda(e_x)\rangle$ represent the sequence of event types. The set of all distinct paths, distinct sequences of events and distinct sequences of event types are represented as $\Pi(v_1, v_2)$, $\Pi(v_1, v_2)|_E$ and $\Pi(v_1, v_2)|_{\lambda(E)}$, respectively. A state v_x is in $\pi(v_1, v_2)$, denoted as $v_x \in \pi(v_1, v_2)$, if there is a transition (v_i, e_k, v_j) in $\pi(v_1, v_2)$, such that $v_x = v_i$ or $v_x = v_j$. By the abuse of notation, let $e_x \in \Pi(v_1, v_2)$ denote the existence of a path between v_1 and v_2 that includes e_x and, similarly for transitions, let $(v_a, e_i, v_b) \in \Pi(v_1, v_2)$ denote the existence of a path including the transition (v_a, e_i, v_b).

A *scope* $\mathcal{S}$ is a pair of start v_s and end v_e states, such that there is at least one path from v_s to v_e, i.e., $|\Pi(v_s, v_e)| >= 1$. A scope $\mathcal{S}$ is called a *concurrency scope* for a pair of events e_1 and e_2, if the given global concurrency oracle $\mathcal{C}$ and validation function $\mathcal{F}$ recognize a valid concurrency relation between such events.

Definition 7 (Concurrency scope)**.** *Let $G = \langle V, i, W, E, T\rangle$ be a transition graph, $\lambda : E \to \Lambda$ be a labelling function, $\mathcal{C}$ be a concurrency oracle, and $\mathcal{F}$ be a validation function. A concurrency scope $\mathcal{S}$ is the tuple $\langle v_s, v_e, (e_1, e_2)\rangle$, where $v_s \in V$ is the start state, $v_e \in V$ is the end state and $(e_1, e_2) \in E \times E$ is a pair of events, such that $(\lambda(e_1), \lambda(e_2)) \in \mathcal{C}(\Pi(v_s, v_e)|_{\lambda(E)})$, $e_1, e_2 \in \Pi(v_s, v_e)$ and $\mathcal{F}(\mathcal{S}, (\lambda(e_1), \lambda(e_2))) = true$.* ⌟

Intuitively, a concurrency scope for a pair of events (e_1, e_2) is valid if their types are deemed concurrent by the global concurrency oracle, they appear at least in one path in the scope, and the validation function asserts such relation. We turn our attention to the computation of the scopes where we rely on well known concepts of graph theory, dominator and post-dominator relations. Intuitively, in any directed graph, a vertex a is the dominator of a vertex b if every path from an initial node i to b contains a. For directed graphs with a final vertex f, we say that a vertex z is the post-dominator of a vertex y if all paths from y to f contain z. Both, dominator and post-dominator relations are

reflexive and transitive, and their transitive reduction are rooted trees referred to as dominator tree $\mathcal{T}_{dom}$ and post-dominator tree $\mathcal{T}_{post}$, respectively.

As shown in the example displayed in Fig. 4, the concurrency relations between a pair of events can hold only in certain executions. Thus, our approach will decompose a transition graph with many final states (each of them representing an execution) into subgraphs from the initial state $\varnothing$ to each of the final states, and then compute the concurrency scopes for each of those subgraphs.

The algorithm for the computation of the concurrency scopes is displayed in Algorithm 1. Consider the transition graph G, a global concurrency oracle and a validation function. Then it starts by iterating over the subgraphs G' from the initial state to one of the final states v in G (Line 3). For each subgraph G', the dominator and post-dominator tree are constructed (Lines 4 and 5). In a postorder manner in the dominator tree, the start of a scope is selected v_e (Line 6), while the end of the scope is the parent of v_e in the post-dominator tree (Line 8). In Line 10, each of the concurrency relations retrieved by the local concurrency oracle is checked in the scope (v_s, v_e). If the validation function asserts the concurrency relation between a pair of event types (a, b) in a scope (v_s, v_e), then it is added as a concurrency scope associated to a final state v (Line 19). The algorithm considers two operations for expanding a concurrency scope whenever a concurrency relation holds (Line 20-22), or restricting the scope when the validation function fails and smaller concurrency scopes could be detected (Line 25-28). The output of the algorithm is a set of tuples $(v, \mathcal{S})$, where v is a set of events representing a final state, and $\mathcal{S}$ is a concurrency scope. Observe that a single final state v can be associated to many concurrency scopes. Thus, given an event log, the concurrency scopes for a trace σ are those in $\{(\widehat{\sigma}, \mathcal{S})\}$.

We observe that the data structures in the Algorithm 1 are all finite. In particular, the log used to construct G contains a finite number of traces and events, thus there is a finite number of final states and the loop in line 3 terminates. Every subgraph from the initial to a final state is finite, since every trace in the log contains a finite number of events, and as a result the dominator and post-dominator trees are finite as well, which implies a finite number of iterations over the loop in line 6. The loop in line 9 iterates over the set of concurrency relations computed by the oracle which, by the finiteness of the events, is finite. As for the procedure *computeScope*, if a concurrency scope is valid (line 17), then a recursive call of the method occurs and v_e becomes its parent in the post-dominator tree. Given that this is a tree, every node has a unique parent with exception of the root, which has none, hence for each node except for the root, there is only one option to move up. Finally, if v_e is not a leaf (line 27), then v_e becomes the unique child of the node. In this case, if the scope has not been added to $\mathcal{O}$ and a concurrency relation exists between the analyzed events (tuple a, b in the algorithm), a recursive call to procedure *computeScope* is done. The algorithm terminates since a valid concurrency scope is only checked once, and the value for v_e to expand or restrict a scope is, at most, equivalent to the total number of nodes in the tree.

Next, we present an example of a concurrency oracle and a validation function that could be plugged into the algorithm.

Algorithm 1: Computing concurrency scopes

Algorithm
  Input: Transition graph $G = \langle V, \varnothing, W, E, T \rangle$, concurrency oracle $\mathcal{C}$ and validation function $\mathcal{F}$
  Output: Concurrency scopes $\mathcal{O} = \{(v, \mathcal{S}) \mid \mathcal{S}$ is a concurrency scope, and $v \in W\}$
  $\mathcal{O} \leftarrow \varnothing$
  for $G' = \langle V', \varnothing, v, E', T' \rangle$ *such that* $v \in W$ **do**
    $\mathcal{T}_{dom} \leftarrow$ dominator tree of G'
    $\mathcal{T}_{post} \leftarrow$ post-dominator of G'
    for v_s *in* $\mathcal{T}_{dom}$ *in postorder* **do**
      if $v_s \neq v$ *and* v_s *has a parent in* $\mathcal{T}_{post}$ **then**
        $v_e \leftarrow$ parent of v_s in $\mathcal{T}_{post}$
        foreach $(a, b) \in \mathcal{C}(\Pi(v_s, v_e)|_{\lambda(E)})$ **do**
          `computeScope`$(v, (v_s, v_e), (a, b))$
        end
      end
    end
  end
  return $\mathcal{O}$
Procedure `computeScope`$(v, (v_s, v_e), (a, b))$
  if $\mathcal{F}((v_s, v_e), (a, b))$ **then**
    $\mathcal{O} \cup \{(v, \mathcal{S})\}$, such that
      $\mathcal{S} = \langle v_s, v_e, (a_i, b_j) \rangle$ and $a_i, b_j \in \Pi(v_s, v_e)$ ▷ Add as concurrency scope
    if v_e *has a parent in* $\mathcal{T}_{post}$ **then**
      $v_e \leftarrow$ parent of v_e in $\mathcal{T}_{post}$ ▷ Expand
      `computeScope`$((v_s, v_e), (a, b))$
    end
  else
    if v_e *has a child in* $\mathcal{T}_{post}$ **then**
      $v_e \leftarrow$ child of v_e in $\mathcal{T}_{post}$ ▷ Restrict
      if $\{(v, \mathcal{S}')\} \notin \mathcal{O}$ *and* (v_s, v_e) *is a scope and* $(a, b) \in \mathcal{C}(\Pi(v_s, v_e)|_{\lambda(E)})$ **then**
        ▷ where $\mathcal{S}' = \langle v_s, v_e, (a_i, b_j) \rangle$
        `computeScope`$((v_s, v_e), (a, b))$
      end
    end
  end

3.2.1 Global (baseline) concurrency

We rely on existing concurrency oracles for the computation of the concurrency relations between pairs of events, e.g. [3, 25, 8], and assume that no two events with the same label can be executed concurrently. As a baseline we use the concurrency relation introduced in [3], herein referred to as α concurrency. Intuitively, a pair of labels a, b are concurrent if a is sometimes observed immediately after b and vice-versa. The definition of α concurrency is given next.

Definition 8 (α concurrency [3]). *Let σ be an event trace. A pair of tasks with labels $a, b \in \mathcal{L}$ are said to be in α* directly precedes relation*, denoted $a \prec b$, iff there exists a trace $\sigma = \langle e_1\ e_2 \dots e_n \rangle$ in L, such that $a = \lambda(e_i)$ and $b = \lambda(e_{i+1})$, $1 \leq i < n$. A pair of tasks a, b are α* concurrent*, denoted $a \parallel b$, iff $a \prec b \wedge b \prec a$.* ⌟

Oftentimes the concurrency relations computed by the concurrency oracle, and in particular by the α concurrency, can be spurious. For instance, consider the trace $\langle a_1\ b_1\ c_1\ d_1\ b_2\ a_2 \rangle$, where the α concurrency would deem events with labels a, b as concurrent. Thus, the validation function is used to refine the results of the concurrency oracle and filter out spurious concurrency relations.

3.2.2 Validation of concurrency relations

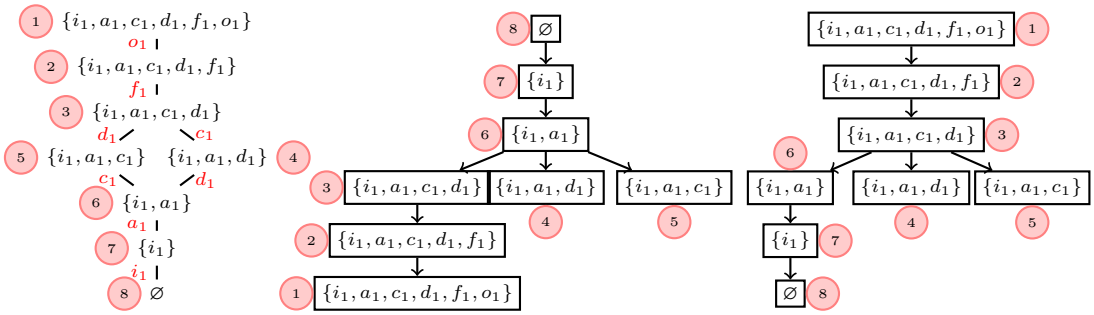
As an example, we define a validation function based on the proportion of events deemed as concurrent that can be executed from the same states in the scope.

Definition 9 (Validation function). *Let $co(a,b)|_{(v_s,v_e)} = \{v \in \Pi(v_s, v_e) \mid (v, a_i, v'),\ (v, b_j, v'') \in \Pi(v_s, v_e)\}$ be the set of states within the paths $\Pi(v_s, v_e)$ where events with labels a and b can occur. Additionally, let $\#(a) = \{(v, a_i, v') \in \Pi(v_s, v_e)\}$ be the set of transitions associated to the occurrence of an event with label a in $\Pi(v_s, v_e)$. Then, the occurrence of a, b w.r.t. to a is $f(a) = \frac{|co(a,b)|_{(v_s,v_e)}|}{|\#(a)|}$ and, similarly w.r.t. b, $f(b) = \frac{|co(a,b)|_{(v_s,v_e)}|}{|\#(b)|}$. The event types a, b are concurrent in (v_s, v_e) iff the following three conditions hold 1. $f(a) > tOccurrence$, 2. $f(b) > tOccurrence$, and 3. $abs(f(a) - f(b)) < tBalance$, for some thresholds $tOccurrence$ and $tBalance$.* ⌟

Intuitively, the validation function checks that the number of events with labels a, b can often be executed from the same state w.r.t. to their total number of occurrences (i.e., the proportion is higher than a given threshold $tOccurrence$) and that the proportions between those events are similar enough (i.e., not bigger than $tBalance$).

As an example, let G, $\mathcal{C}$ and $\mathcal{F}$ be the transition graph displayed in Fig. 4, the α concurrency oracle (Def. 8) and the validation function presented above (Def. 9), respectively. Algorithm 1 starts by computing a subgraph G' for each of the final states. Assume G' is the subgraph displayed in Fig. 6a and $v = \{i_1, a_1, c_1, d_1, f_1, o_1\}$ is the final state. The dominator and post-dominator tree of G' are displayed in Figs. 6b and 6c, respectively. The numbers next to the nodes in Fig. 6 correspond to the order in which

they will be processed (postorder computed over the dominator tree). The first processed node is $v_s = \{i_1, a_1, c_1, d_1, f_1, o_1\}$ (observe that the *if* condition (line 7 in the algorithm) will evaluate to false, since v_s has no parents in the post-dominator tree). Then, in the next iteration $v_s = \{i_1, a_1, c_1, d_1, f_1\}$ and $v_e = \{i_1, a_1, c_1, d_1, f_1, o_1\}$, the latter since $\{i_1, a_1, c_1, d_1, f_1, o_1\}$ is the parent of v_s in the post-dominator tree; however, $\mathcal{C}$ would not detect any concurrency relation in the paths from v_s to v_e. Consider the case when $v_s = \{i_1, a_1\}$ and thus $v_e = \{i_1, a_1, c_1, d_1\}$, where (c, d) are concurrent according to $\mathcal{C}$, and this pair is passed to $computeScope(v, (v_s, v_e), (c, d))$ to verify if it is a concurrency scope. Then, for the validation function, $co(c, d)|_{(v_s, v_e)} = \{\{i_1, a_1\}\}$, $\#(c) = \{(\{i_1, a_1\}, c_1, \{i_1, a_1, c_1\}), (\{i_1, a_1, d_1\}, c_1, \{i_1, a_1, c_1, d_1\})\}$ and $\#(d) = \{(\{i_1, a_1\}, d_1, \{i_1, a_1, d_1\}), (\{i_1, a_1, c_1\}, d_1, \{i_1, a_1, c_1, d_1\})\}$, so $f(a) = \frac{1}{2}$, $f(b) = \frac{1}{2}$ and $abs(f(a) - f(b)) = 0$. Therefore, for any pair of thresholds $tOccurrence < 0.5$ and $tBalance > 0.0$, the scope $(v, (v_s, v_e), (c, d))$ will be detected as a concurrency scope by $\mathcal{F}$, in which case the algorithm will attempt to expand it to the parents of v_e.



(a) Subgraph (b) Dominator tree (c) Post-dominator tree

Figure 6: Computation of example concurrency scopes for the transition graph in Fig. 4

3.2.3 Complexity analysis

The worst-case time complexity of our algorithm is dominated by the complexity of constructing the transition graph of an event log, and the complexity of computing the scopes and the traces within them. The complexity of these steps is polynomial. Given a log with n number of events, the construction of the transition graph is done in $O(\frac{n(n+1)}{2})$. This is because for every event e_i in the log, $1 \leq i \leq n$, it is necessary to spot the state in the transition graph that reflects the execution of e_i with respect to the given equivalence relation. If there is no equivalent state then a new state is added. Thus, the number of comparison operations for finding equivalent states increases, at most, together with the number of events analysed.

The complexity of the computation of scopes and traces within a scope is as follows. Given a transition graph with V states, T transitions and a unique final state, the dominator and post-dominator trees can be computed in $O((|T| + |V|)log(|T| + |V|))$ with the Lengauer-Tarjan algorithm. Independently of the traversals of the dominator and post-dominator trees, there can be at most $|V|^2$ scopes (all possible combinations of start-end states). Then, given a

scope $\mathcal{S}$ with E' events and V' states, there can be up to a factorial number of paths (i.e., all interleavings of the concurrent execution of the events E') in $\mathcal{S}$. However, using dynamic programming techniques and given that the transition graphs are directed and acyclic, the computation of the paths can be done in $O(|V'| + |T'|)$, where T' is the number of transitions in $\mathcal{S}$.

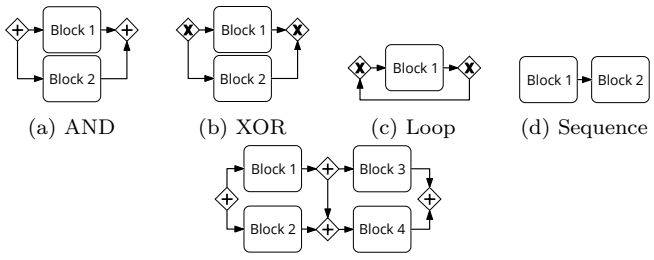
The next section presents the evaluation with a set of synthetic logs and uses the concurrency oracle and validation function presented above.

4 Evaluation of accuracy and time performance

We implemented our local concurrency oracle in a Java tool called *ProLoCon*[4] and used it to evaluate the approach's accuracy and time performance. The tool takes as input a log in MXML or XES format, a concurrency oracle (currently the α oracle [3] and the oracle in [25] are supported), as well as the values of the thresholds *tOccurrence* and *tBalance* for the validation function.

4.1 Datasets generation

In order to evaluate the accuracy of our approach, we generated a gold standard consisting of a set of synthetic process models, which are single-entry single-exist (SESE) and capture a wide combination of control-flow constructs. These models were obtained by randomly composing the following SESE fragments: AND, XOR, Loops, Sequences and Z-blocks (see Fig. 7 for the BPMN notation of each fragment). The models were generated as trees of height two, where every internal node is a fragment either containing nested fragments or atomic activities, and every leaf is an activity. The leaves of the trees are activities randomly chosen and duplicate activities are allowed as long as they do not introduce auto-concurrency, i.e., pairs of activities with the same label cannot belong to the same parallel block. This led to a total of 82 models, ranging from a minimum size of 10 nodes to a maximum size of 20 nodes (avg. = 15.5 nodes). Out of these models, ten are cyclic and all include at least one pair of concurrent activities.



(a) AND (b) XOR (c) Loop (d) Sequence

(e) Z block

Figure 7: Fragments used for the generation of synthetic process models.

The logs were generated from the synthetic models using the ProM plugin

[4]Available at `http://apromore.org/platform/tools`

"Generate Event Log from Petri Net" [31].[5] The obtained logs vary from a minimum of 4 to a maximum of 300 traces (avg. = 24 traces), with a total number of events ranging from 24 to 2,400 (avg. = 173 events).

4.2 Setup

Using the model-log pairs in our dataset, we computed the *F-score* between the concurrency relations identified in the log and those extracted from the respective model (gold standard). The computation of the F-score is divided into three steps: 1. create the local concurrency oracle from the log, 2. encode the behavior a model and, using the oracle, of a log as a model of true concurrency, and 3. compare the expected concurrency (the one occurring in the model) with the one observed in the log. The evaluation framework is shown in Fig. 8. For the encoding of the behavior of models and logs, we reuse the transformation proposed in [14]. Generally speaking, [14] puts forward a technique for conformance checking which considers the encoding of the behavior of both models and logs as Prime Event Structures (PESs) [27], a well-known model of true concurrency. In fact, the mentioned formalism, PES, has been suggested as a suitable formalism for a unified behavioral representation of a log and a process model in the context of process mining [12].

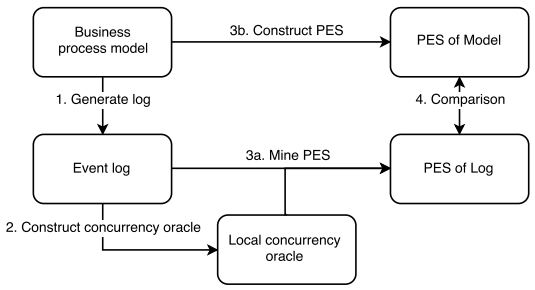


Figure 8: Evaluation framework (artificial logs).

A PES captures the set of traces of a log or process model by means of events which are related via three different binary relations: conflict, causality and concurrency. Events in a conflict relation cannot occur in the same execution, hence, all events in a trace are conflict-free. The order between events is defined via the causality relation, i.e., an event e is causal to e' if e' can only occur after e. Finally, concurrency is the absence of an order between events, thus a pair of events in a concurrency relation can occur in

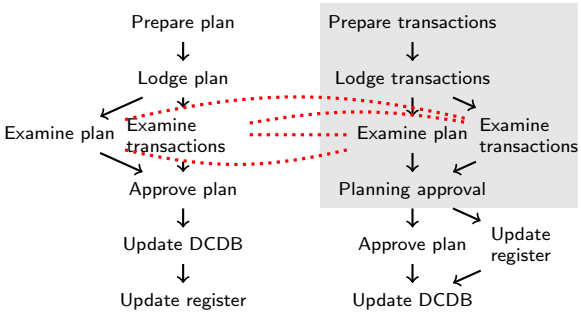


Figure 9: PES of model in Fig. 1

[5] Parameters of the simulation: complete generation; min./max. traces to add for each generated sequence: 1; max. times marking seen: 2; only include traces that reach end state; only include traces without remaining tokens.

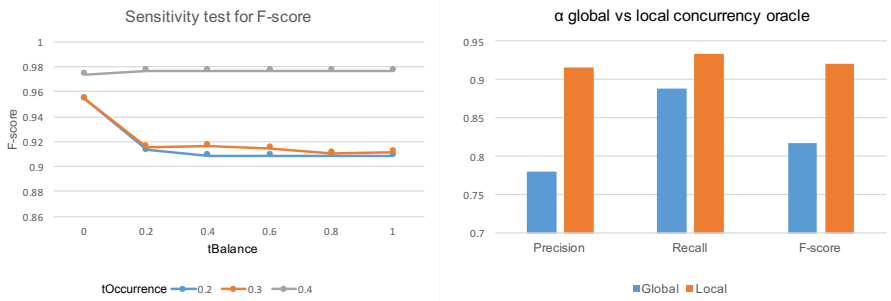
any other. For example, Fig. 9 shows the PES for the land development model of Fig. 1: the arrows represent causality relations, while the red dotted lines represent conflicts. In a PES, if two events are neither in conflict nor in causal relation, they are concurrent. A *configuration* is a set of events that are conflict-free and partially-ordered. An event, or set of events, that can occur at a given configuration is an *extension* of a configuration; thus, if two events are concurrent, they are extensions of at least one configuration. In Fig. 9, the events in the shaded area are an example configuration, and the two events *Approve plan* and *Update register* are its two possible extensions. These two notions, configurations and extensions, describe the execution semantics of a PES.

[14] extends the classical notion of PES to represent the behavior of both acyclic and cyclic process models, as well as the behavior encoded in a log. In the latter case, it requires as input any concurrency oracle for transforming the traces in the log into partially-ordered sets of events which are then used to build the PES. According to [14], given a pair of PESs P_{log}, P_{model} from a log and a model, respectively, a pair of configurations s_1 of P_{log} and s_2 of P_{model} are *equivalent* if they represent equivalent events (i.e. with abuse of notation the two configurations are set equivalent as per Def. 3). Then, let TP (true positives) be the set of equivalent configurations s_1, s_2 where for any pair of concurrent events a, b which are extensions of s_1 in P_{log}, there is a pair of concurrent events a', b' extending s_2 in P_{model}, such that $a \sim a'$ and $b \sim b'$. Let FP (false positives) be the set of equivalent configurations s_1, s_2, where there is a pair of concurrent events a, b that are extensions of s_1 in P_{log} and for which there is no equivalent events extending s_2. Finally, let FN (false negatives) be the set of equivalent configurations s_1, s_2, such that there is a pair of concurrent events a', b' extending s_2 in P_{model} and for which there is no equivalent events extending s_1.

Having defined the sets of true positives, false positives and false negatives, we can compute Precision and Recall, and their F-score.

4.3 Results

Before measuring the accuracy of our approach, we did a sensitivity test on the thresholds *tOccurrence* and *tBalance*. This test was performed using the synthetic dataset and consisted in trying different configurations for both thresholds; the values used for *tOccurrence* were between $[0.0, 0.5]$, with increments of 0.1, and for *tBalance* were between $[0.0, 1.0]$ with increments of 0.2. In the case of *tOccurrence* > 0.5, the validation function resulted too strict and no concurrency relation could be detected, leading to a very low F-score. Each combination of the thresholds was used to measure the F-score over each of the model-log pairs in the dataset and, finally, all values for the F-score were averaged for each configuration of thresholds. Figure 10a shows how the F-score varies according to three different combinations of *tOccurrence* and *tBalance*. The rest of the combinations gave similar values to either of the trends presented in the chart and thus were omitted. Observe that the F-score plateaus at a value of 0.977 with *tOccurrence*=0.4 and *tBalance*=0.2. The same result was obtained with a *tOccurrence*=0.5. Therefore, we chose *tOccurrence*=0.4 and *tBalance*=0.2 for our experiments.



(a) Sensitivity test for *tOccurrence* and *tBalance*

(b) Avg. precision, recall and F-score for the α oracles

Figure 10: Sensitivity test and accuracy for the synthetic dataset.

The other parameters used for the evaluation are the state equivalence of Def. 5, and the α concurrency as the global baseline concurrency oracle (Def. 8). The concurrency oracle derived from [25] was also used as a global baseline oracle; however, the identified concurrency relations were exactly the same as those retrieved by the α concurrency.

First, we computed the precision, recall and F-score for each of the model-log pairs using both the global and the local concurrency oracle. Then, for each of the oracles, we averaged the values for precision, recall and F-score across all model-log pairs. Figure 10b reports the average results for all 82 model-log pairs. The bar diagram shows a sensible increment in F-score (from 0.82 to 0.92), mostly determined by the increase in precision when using our local concurrency oracle instead of the global one (from 0.78 to 0.92). The lower precision of the α oracle is due to its over-generalization, given that local concurrency relations are identified as global relations. The lower recall of the α oracle is due to the assumption that the log is complete w.r.t. the direct follows relation, meaning that all possible such relations are expected to be present in the log, for the oracle to accurately detect the α-relations. However, this assumption hardly holds in real-life datasets, hence we did not enforce log completeness when generating the logs for our experiments.

There were two problematic constructs where the local concurrency oracle failed to accurately determine the scope of a pair of concurrency relations. One construct is when there are two AND blocks containing the same tasks and following each other. For example, given a log $\{\langle a\ b\ a\ b\rangle, \langle b\ a\ a\ b\rangle, \langle a\ b\ a\ b\rangle, \langle a\ b\ b\ a\rangle\}$, the local concurrency oracle identifies as concurrent every pair of a and b from the beginning to the end of every trace, and thus fails to identify that the second occurrence of a and b depends on the first occurrence. The other problematic construct is a sequence of a loop of an activity a, followed by a concurrent block of activities a and b. In this case, the concurrency oracle identifies the event in the loop as concurrent with b. These cases are however also misclassified by the

	Global (ms)	**Local (ms)**
Max	0.788	172.064
Min	0.006	0.392
Avg	0.060	6.867

Table 1: Execution times of our oracle against the α oracle.

global baseline oracle.

Table 1 reports the statistics on the execution time (in milliseconds) for the computation of the concurrency relations using the global α oracle and its local counterpart. Even though the execution times for the computation of our local concurrency oracle are sensibly higher than those of the global one, the overall time taken is still quite low, in the order of milliseconds (average of 6.9 ms).

5 Evaluation of generalization effects

As a second experiment, we evaluated the effects of using a local concurrency oracle on the generalization of the process behavior captured in the event log. First, we compared the generalization introduced by the local concurrency oracle with that introduced by the global concurrency oracle. Second, in order to investigate the possible over-generalization inherent to existing process discovery algorithms, we compared the generalization introduced by the local concurrency oracle with that of the Inductive Miner [18], which is a state-of-the-art automated discovery technique. Specifically, given that the Inductive Miner can generate a Petri net out of an event log, which is able to replay every trace in the log, we used such model as an oracle (referred to as *"inductive"* oracle hereinafter) that transforms a trace into a partial order (process induced by replaying the trace on the net).

For this second experiment we used seven real-life logs. The first log is that distributed with the BPI Challenge (BPIC) 2012; it captures executions of a personal loan origination process at a Dutch financial institute.[6] The second log captures executions of an IT service desk process at an Italian IT Vendor, for handling both service requests and incidents. The third log captures executions of a business process for plan lodgement and document registration in two Australian states, as recorded by a land development company, whose model is depicted in Figure 1. The fourth, fifth and sixth logs capture executions of a process for handling motor glass claims at an Australian insurance company. Finally, the last event log is extracted from an information system for managing road Traffic fines in Italy.[7] The characteristics of these logs are reported in the first part of Table 2.

In this experiment the gold standard (normative model of the process) is unknown, thus it is not possible to perform a sensitivity test to obtain the best values for *tOccurrence* and *tBalance*, which are used by the local concurrency oracle. Therefore, we reuse the thresholds chosen for the experiments in Section 4, i.e., *tOccurrence*=0.4 and *tBalance*=0.2, assuming that the level of completeness of the log is comparable to that of the synthetically generated logs. The state equivalence and the concurrency oracle are those of Def. 5 and Def. 8, respectively.

To measure the effects of generalization, we first transformed each trace of each real-life log into a partial order run using the local, global and inductive oracles. Next, we measured the number of *true global* (TG) concurrency relations, i.e. a concurrency relation between events a_1, b_1 is considered true global if there

[6] doi:10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f

[7] doi:10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5

is a scope computed by the local oracle for every occurrence of the event types a, b; the number of *true local* (TL) relations (or *false global*), i.e. those relations that were identified by the global (inductive) oracle, for which our oracle found a local scope; and the number of *false concurrent* (FC) relations, i.e. those pairs of events which the global (inductive) oracle identified as being concurrent, but were not found to be concurrent at all by our local oracle. As an example, with reference to Fig. 1, the concurrency relation between "Update register" and "Update DCDB" is a case of false concurrency, since these two event types are actually never concurrent, while the concurrency relation between "Approve plan" and "Update register" is an example of true local concurrency.

Using these measures, we then computed the *concurrency over-generalization ratio* as the ratio between the number of false global and false concurrent relations, and the total number of concurrency relations found by the global (inductive) oracle, i.e. $c_{og} = \frac{TL+FC}{TG+TL+FC}$. In essence, this formula measures how much the global oracle over-generalizes the behavior captured in the log, either by identifying a local concurrency relation as being global, or by identifying a pair of events as being (globally) concurrent when they are not. The results are reported in the second part of Table 2 for both the global and the inductive oracles.

Observe that the global and inductive oracles have a concurrency over-generalization ratio ranging from 25% in the case of one of the insurance company logs, to 100% in the case of the BPIC 2012, IT vendor and Traffic fines logs. The high value obtained in these latter logs suggests that many such concurrent relations are indeed authentically local. This observation is supported by the average number of repeated events per trace in these three logs, i.e. the ratio between the number of events in a trace and the number of event types in that trace, averaged across all traces. This ratio is 1.3 in the Traffic fines log (30% of events are repeats of other events) and as high as 4.3 in the IT Vendor log and 4.4 in the BPIC log (i.e. an event type appears on average four times in a trace). However, some of these local relations may actually be due to the incompleteness of the log. Thus, the validation function of our oracle may turn out to be too strict because not all the interleavings of concurrent tasks are captured in the log, leading to a reduced scope of the relation, hence to local concurrency.

Log	**Events**	**Event types**	**Traces**	**Distinct traces**	**Avg trace length**	**TG**	**Alpha Conc. relations**			**Inductive Conc. relations**		
							TL	FC	c_{og}	TL	FC	c_{og}
BPIC 2012	262,200	23	13,087	4,336	42	0	44	35	1	15	0	1
IT Vendor	75,353	9	12,720	1,026	13	0	8	1	1	7	0	1
Land dev.	18,240	14	1,440	36	12.6	5	1	1	0.28	1	1	0.29
Insurance 1	19,544	12	4,954	74	4.39	6	11	1	0.67	11	0	0.65
Insurance 2	7,606	11	1,853	38	4.63	6	2	0	0.25	2	0	0.25
Insurance 3	52,361	12	13,302	102	4.50	5	15	1	0.76	15	1	0.76
Traffic fines	561,470	11	150,370	231	8.18	0	26	2	1	26	0	1

Table 2: Statistics on real-life logs and their concurrency relations.

Figure 11 shows an example of a trace extracted from the Traffic fines log, that resulted in two non-isomorphic partial order runs, after relaxing the order relations of the events in the trace according to the local oracle and to the global oracle. As we can see, the run obtained with the global oracle identifies the "Payment" event as being concurrent with all other events. This is a clear case of concurrency over-generalization because logically the payment for a fine can only be done after the fine has actually been sent to the citizen. Similarly, event "Receive result appeal from prefecture" is concurrent to "Insert date appeal to prefecture", which again is logically not possible. This is due to the repetition of such events within the same traces in the log. On the other hand, our local oracle identifies the correct order of these events, by placing "Payment" at the end of the run, and "Insert fine notification" and "Add penalty" in a local concurrency relation with "Inser date appeal to prefecture", with the latter event always preceding "Receive result appeal from prefecture".

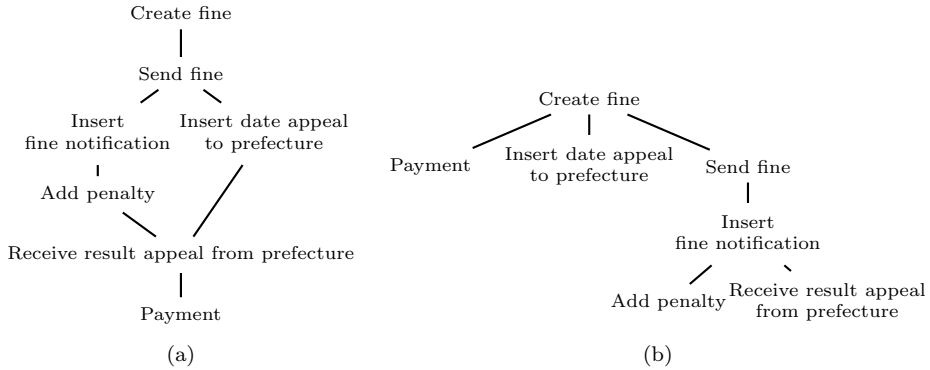


Figure 11: Traffic fines log: partial order runs derived from the same trace using the local oracle (a) and the global oracle (b).

6 Application to process drift detection

This section presents the results of an experiment conducted to study the impact of using our local concurrency oracle instead of the global one for process drift detection.

Early detection of business process changes based on event logs, also known as *process drift detection*, enables analysts to identify and act upon changes that may otherwise affect process performance. A drift is a statistically significant change observed in an event log or in a stream of traces. In [21, 22], we introduced a drift detection technique where the basic idea is to perform a statistical test over the distributions of partially-ordered runs observed in two consecutive time windows sliding over a log or a stream of traces. A run represents a set of traces that are equivalent to each other modulo a concurrency relation between event types. First, a pair of sets of completed process traces, which are extracted from two juxtaposed time windows, are transformed into runs with the use of a concurrency oracle. Then, in order to detect a process drift, we evaluate the hypothesis of whether the statistical distributions of the distinct runs extracted

from the two juxtaposed windows are similar, using the Chi-square test on the contingency matrix built from the frequencies of the distinct runs in the two windows. A drift is detected when the P-value outputted by the statistical test is less than the significance level (the test threshold is typically set to 0.05).

In [21, 22], we used the global α concurrency oracle to build runs from traces. In this experiment we replace this oracle with our local oracle and re-assess the drift detection accuracy, to see if there is any noticeable improvement. To do so, we applied both versions of the technique on an event log recording 2,259 execution traces of a commercial claims handling process at an Australian insurer. This log spans over a period of one year and contains 13,454 events of which twelve are distinct. Similar to the experiments presented in Section 5, the normative model of the log is unknown, so we use the same settings for the local concurrency oracle as those used in Section 5 ($tOccurrence$=0.4 and $tBalance$=0.2, with the state equivalence as defined in Def. 5 and the concurrency oracle as defined in Def. 8).

Figure 12 reports the *P–value* of consecutive statistical tests based on the two oracles. Overall, the two plots reveal similar behavior. Indeed, two drifts were detected at the same position by the detection method when using either oracle. Nevertheless, the use of the global oracle led to detecting a further drift at trace 386 which was not detected when using the local concurrency oracle.

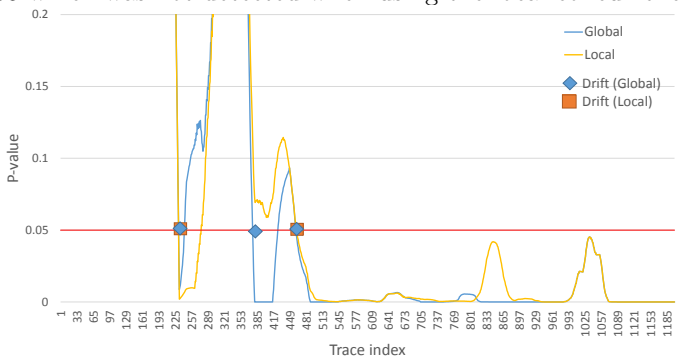


Figure 12: Plot of the *P–value* of the statistical test for drift detection, using both the global and local concurrency oracles.

Upon inspection of the runs underpinning the only drift not detected by the local oracle, we found that the global oracle discovered a concurrent relation in the window before the drift, but not in the window after the drift. This false concurrency relation was discovered even if one of the interleavings between the two events involved in the relation (from "ReviewApprovePayment" to "ReviewInvoice") was much less frequent than the other interleaving. Figure 13 plots the relative frequency of the two interleavings over the log. The use of a local concurrency oracle can then detect process drifts more accurately, for instance, by filtering out cases of spurious concurrency relations, like in our experiment above, or by detecting finer-grained changes in concurrency between time windows, hence avoiding generalization.

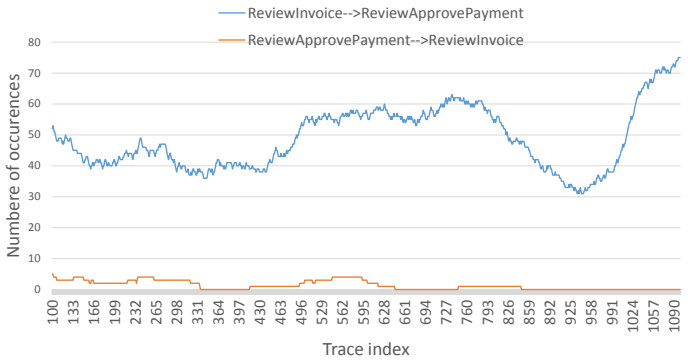


Figure 13: Plot of the number of occurrences of the interleaved event labels over a sliding window (initialized with 100 traces).

7 Threats to validity

The evaluation of our approach is subject to some threats to validity. First, we assume that every trace in the log is a complete execution, and the log is noise-free. The computation of the local concurrency oracle assumes that the data received as input shall be faithfully preserved throughout the intermediate representations used by our technique. Thus, the pre-processing of the log – for instance to remove noise or incomplete behavior – is considered as an orthogonal problem. An example of a noise filtering technique which could be used before the computation of the local concurrency oracle, is presented in [7]. Second, we assume the real-life logs used in the experiments in Section 5 are somewhat complete. Specifically, we assume they contain enough observations of possible interleavings between concurrent event labels (tasks). Given that the normative process models for these real-life logs are non-existent, it was not possible to compute the accuracy of our technique with respect to a gold standard. Instead, we used a notion of *concurrency over-generalization ratio* to measure the level of potential over-generalization induced by a global concurrency oracle assuming that the log contains enough observations of the possible interleavings between concurrent event labels. The use of this ratio in the experimental evaluation constitutes a threat to construct validity. We mitigated this threat by considering sufficiently large event logs (1000+ traces) such that if multiple event labels are concurrent, we would expect to observe a significant percentage of their possible interleavings. If the latter assumption holds, the concurrency over-generalisation ratio provides a reliable measure of the potential over-generalization of the global concurrency oracle.

Another implication of the absence of the normative process models for the real-life logs is that it was not possible to make a sensitivity test to obtain the best values for the thresholds *tBalance* and *tOccurrence*. For the experiments in Sections 5 and 6, we reused the values computed in Section 4. Although these thresholds were computed over a synthetic dataset, the values are not too strict and allow some flexibility in the presence of incompleteness w.r.t. all possible interleavings of concurrent behavior.

Finally, the use of only seven real-life event logs in the evaluation, limits, to some extent, the generalizability of the conclusions. However, the event logs

included in the evaluation are of different sizes and characteristics, and originate from different application domains.

8 Conclusion and future work

This paper presented an approach to turn any algorithm for constructing a global concurrency oracle from an event log into one that constructs local oracles. By scoping the concurrency to a set of states in a state transition graph constructed from the event log, the approach effectively increases the accuracy of the detected set of concurrency relations, while avoiding over-generalization of the process behavior captured in the log. Experimental results have shown that the local concurrency oracles derived from the proposed approach outperform the corresponding global oracles when applied to the task of extracting partially ordered runs from an event log. An application in the context of business process drift detection has shown the ability of the derived local concurrency oracles to enhance the accuracy of existing process mining methods.

The experimental evaluation suggests that there is room for improvement in the proposed method, particularly the local concurrency oracle fails to find accurate scopes in cases where two blocks of concurrency — including the same event types — precede each other, which leads to detecting bigger scopes than the actual, and when loops and concurrency blocks with common event types precede each other, in which case the event in the loop is detected as concurrent with the event types of the concurrent block. A more extensive evaluation with other (global) concurrency oracles and other parameters for constructing the transition system could inform the development of more robust variants of the proposed method. Additionally, we would like to explore the effects of different values for the thresholds used in our local concurrency approach, given different levels of incompleteness of real-life logs.

Another direction for future work is to explore other applications of the proposed concurrency oracle, for example by combining it with techniques for conformance checking [14], log delta analysis [30] and automated process discovery [13], which are based on models of concurrency based on partial orders, and may thus potentially benefit from a finer-grained distinction between causality and concurrency relations. These techniques need to be adapted to take as input not only a pair of events for which concurrency needs to be determined, but also a state of execution for this pair.

In particular, in the context of automated discovery, it is also worth exploring the trade-off between model accuracy and simplicity when comparing a model discovered using a global concurrency oracle with one discovered using a local oracle. Our hypothesis is that the over-generalization of log behavior induced by a global concurrency oracle will result in models exhibiting more behavior than what is actually recorded in the log, hence less precise models. Meantime, a local concurrency oracle will lead to label duplication, and so to models of higher size than those discovered using a global concurrency oracle. However, higher size may be counterbalanced by simpler control-flow structures given that labels are duplicated. So ultimately the complexity of the models discovered with the two different concurrency oracles may be comparable.

Acknowledgments. This research is funded by the Australian Research Council (grant DP150103356) and the Estonian Research Council (grant IUT20-55).

References

[1] van der Aalst, W.M.P.: Process Cubes: Slicing, Dicing, Rolling Up and Drilling Down Event Data for Process Mining, pp. 1–22. Springer International Publishing (2013), https://Doi.org/10.1007/978-3-319-02922-1_1

[2] van der Aalst, W.M.P., Rubin, V., Verbeek, E., van Dongen, B.F., Kindler, E., Günther, C.: Process mining: a two-step approach to balance between underfitting and overfitting. SoSyM 9(1) (2008)

[3] van der Aalst, W.M.P., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE TKDE 16(9) (2004)

[4] Bose, R.P.J.C.: Process mining in the large: preprocessing, discovery, and diagnostics. PhD dissertation, Eindhoven University of Technology, Eindhoven (2012)

[5] Bose, R.P.J.C., van der Aalst, W.M.P.: Trace Clustering Based on Conserved Patterns: Towards Achieving Better Process Models, pp. 170–181. Springer, Berlin (2010)

[6] Brander, S., Hinkelmann, K., Martin, A., Th´onssen, B.: Mining of agile business processes. In: Artificial Intelligence for Business Agility – AAAI Spring Symposium. pp. 9–14. AAAI Press (2011)

[7] Conforti, R., La. Rosa, M., ter Hofstede, A.H.M.: Filtering out infrequent behavior from business process event logs. IEEE TKDE 29(2), 300–314 (Feb 2017)

[8] Cook, J.E., Wolf, A.L.: Event-based detection of concurrency. In: FSE. ACM, New York, NY, USA (1998)

[9] Diamantini, C., Genga, L., Potenaa, D., van der Aalst, W.: Building instance graphs for highly variable processes. Expert Syst. Appl. 59(15) (October 2016)

[10] van Dongen, B.F., van der Aalst, W.M.P.: Multi-phase process mining: Building instance graphs. In: Proc. of ER. Springer (2004)

[11] van Dongen, B.F., Desel, J., van der Aalst, W.M.P.: Aggregating causal runs into workflow nets. Trans. Petri Nets and Other Models of Concurrency 6 (2012)

[12] Dumas, M., García-Bañuelos, L.: Process mining reloaded: Event structures as a unified representation of process models and event logs. In: Proc. of PETRI NETS. Springer (2015)

[13] Fahland, D., van der Aalst, W.M.P.: Simplifying discovered process models in a controlled manner. Inf. Syst. 38(4), 585–605 (2013)

[14] García-Bañuelos, L., van Beest, N.R., Dumas, M., La Rosa, M.: Complete and interpretable conformance checking of business processes. IEEE TSE (2017)

[15] Greco, G., Guzzo, A., Pontieri, L., Sacca, D.: Discovering expressive process models by clustering log traces. IEEE TKDE 18(8), 1010–1027 (Aug 2006)

[16] Hompes, B.F., Buijs, J.C.M., van der Aalst, W.M., Dixit, P., Buurman, J.: Detecting change in processes using comparative trace clustering. In: Data-Driven Process Discovery and Analysis - 5th IFIP WG 2.6 Int. Symp., SIMPDA 2015, Vienna, Austria. pp. 54–75 (2015)

[17] Huang, Z., Lu, X., Duan, H., Fan, W.: Summarizing clinical pathways from event logs. Journal of Biomedical Informatics 46(1), 111–127 (2013)

[18] Leemans, S.J.J., Fahland, D., Aalst, W.M.P.: Proc. of PETRI NETS, chap. Discovering Block-Structured Process Models from Event Logs - A Constructive Approach. Springer, Berlin (2013), http://dx.Doi.org/10.1007/978-3-642-38697-8_17

[19] Leemans, S.J., Fahland, D., van der Aalst, W.M.: Using life cycle information in process discovery. In: in press) BPM Workshops (2015)

[20] Li, J., Liu, D., Yang, B.: Process mining: Extending α-algorithm to mine duplicate tasks in process logs. In: Proc. of APWeb/WAIM 2007 Workshops. Springer (2007)

[21] Maaradji, A., Dumas, M., Rosa, M.L., Ostovar, A.: Fast and accurate business process drift detection. In: Proc. of BPM. LNCS, vol. 9253, pp. 406–422. Springer (2015)

[22] Maaradji, A., Dumas, M., Rosa, M.L., Ostovar, A.: Detecting sudden and gradual drifts in business processes based on event logs. IEEE TKDE (2017)

[23] de Medeiros, A.K.A., van Dongen, B.F., van der Aalst, W.M.P., Weijters, T.: Process mining: Extending the α-algorithm to mine short loops. Tech. rep., Eindhoven University of Technology, Eindhoven (2004)

[24] de Medeiros, A.K.A., Guzzo, A., Greco, G., van der Aalst, W.M.P., Weijters, T., van Dongen, B.F., Saccá, D.: Process Mining Based on Clustering: A Quest for Precision, pp. 17–29. Springer, Berlin (2008), https://Doi.org/10.1007/978-3-540-78238-4_4

[25] Mokhov, A., Carmona, J.: Event log visualisation with conditional partial order graphs: from control flow to data. In: Proc. of ATAED (2015)

[26] Mokhov, A., Yakovlev, A.: Conditional partial order graphs: Model, synthesis, and application. IEEE Transactions on Computers 59(11) (Nov 2010)

[27] Nielsen, M., Plotkin, G., Winskel, G.: Petri nets, event structures and domains, part I. Theoretical Computer Science 13(1) (1981)

[28] Song, M., Günther, C.W., van der Aalst, W.M.P.: Improving process mining with trace clustering. J. Korean Inst. Ind. Eng. 34(4), 460–469 (2008)

[29] Song, M., Günther, C.W., van der Aalst, W.M.P.: Trace Clustering in Process Mining, pp. 109–120. Springer, Berlin (2009), https://Doi.org/10.1007/978-3-642-00328-8_11

[30] van Beest, N., Dumas, M., García-Bañuelos, L., La Rosa, M.: Log delta analysis: Interpretable differencing of business process event logs. In: Proc. of BPM. Springer (2015)

[31] Vanden Broucke, S., Weerdt, J.D., Vanthienen, J., Baesens, B.: An improved process event log artificial negative event generator. Tech. rep., FEB, KU Leuven, Belgium (2012)

[32] Weerdt, J.D., Broucke, S.V., Vanthienen, J., Baesens, B.: Active trace clustering for improved process discovery. IEEE TKDE 25(12), 2708–2720 (Dec 2013)

[33] Wen, L., Aalst, W.M.P., Wang, J., Sun, J.: Mining process models with non-free-choice constructs. Data Mining and Knowledge Discovery 15(2) (2007)