Event Log Visualisation with Conditional Partial Order Graphs: from Control Flow to Data

Andrey Mokhov[1], Josep Carmona[2]

[1] Newcastle University, United Kingdom
andrey.mokhov@ncl.ac.uk
[2] Universitat Politècnica de Catalunya, Spain
jcarmona@cs.upc.edu

Abstract Process mining techniques rely on *event logs*: the extraction of a process model (*discovery*) takes an event log as the input, the adequacy of a process model (*conformance*) is checked against an event log, and the *enhancement* of a process model is performed by using available data in the log. Several notations and formalisms for event log representation have been proposed in the recent years to enable efficient algorithms for the aforementioned process mining problems. In this paper we show how *Conditional Partial Order Graphs (CPOGs)*, a recently introduced formalism for compact representation of families of partial orders, can be used in the process mining field, in particular for addressing the problem of compact and easy-to-comprehend visualisation of event logs with data. We present algorithms for extracting both the control flow as well as the relevant data parameters from a given event log and show how CPOGs can be used for efficient and effective visualisation of the obtained results. We demonstrate that the resulting representation can be used to reveal the hidden interplay between the control and data flows of a process, thereby opening way for new process mining techniques capable of exploiting this interplay.

1 Introduction

Event logs are ubiquitous sources of process information that enabled the rise of the *process mining* field, which stands at the interface between data science, formal methods, concurrency theory, machine learning, data visualisation and others [26]. A *process* is a central notion in these fields and in computing science in general, and the ability to automatically discover and analyse evidence-based process models is of utmost importance for many government and business organisations. Furthermore, this ability is gradually becoming a necessity as the digital revolution marches forward and traditional process analysis techniques based on the explicit construction of precise process models are no longer adequate for continuously evolving large scale real-life processes, because our understanding of them is often incomplete and/or inconsistent.

At present, the process mining field is mainly focused on three research directions: i) the *discovery* of a formal process model, typically, a Petri Net or a BPMN (Business Process Model and Notation); ii) the *conformance* analysis of a process model with respect to a given event log; and iii) the *enhancement* of a process model with respect to additional information (i.e., *data*) contained in an event log. The bulk of research in these directions has been dedicated to the design of the algorithmic foundation and associated software tools with many notable successes, such as, e.g. the ProM framework [2]. However, a more basic problem of *event log visualisation* received little attention to date, despite the fact that effective visualisation is essential for achieving a good understanding of the information contained in an event log. Indeed, even basic *dotted charts* prove

very useful for describing many aspects of event logs even though they are just simple views of event log traces plotted over time [25].

In this paper we discuss the application of *Conditional Partial Order Graphs (CPOGs)* for event log visualisation. The CPOG model has been introduced in [17] as a compact graph-based representation for complex concurrent systems, whose behaviour could be thought of as a collection of many partial order scenarios. The key idea behind our approach is to convert a given event log into a collection of partial orders, which can then be compactly described and visualised as a CPOG. Although CPOGs are less expressive than Petri Nets and have important limitations, such as the inability to represent cyclic behaviour, they are perfectly suitable for representing event logs, which are inherently acyclic. We therefore see CPOGs not as the final product of process mining, but as a convenient intermediate representation of an event log that provides much better clarity of visualisation as well as better compactness, which is important for the efficiency of algorithms further in the process mining pipeline. Furthermore, CPOGs can be manipulated using algorithmically efficient operations such as *overlay* (combining several event logs into one), *projection* (extracting a subset of interesting traces from an event log), *equivalence checking* (verifying if two event logs describe the same behaviour) and others, as has been formalised in [19].

The contribution of the paper is twofold. Firstly, we present a method for deriving compact CPOG representations of event logs, which is based on the previous research in CPOG *synthesis* [17]. Secondly, we propose techniques for extracting data parameters from the information typically contained in *event labels* of a log and for using these parameters for annotating the derived CPOG model, thereby providing a direct link between the control and data aspects of a given system.

The remainder of the paper is organised as follows: the next section illustrates the motivation and contributions of the paper with the help of a small example. Section 3 provides the background on event logs, and Section 4 introduces the theory of CPOGs in detail, placing it in the context of process mining. The extraction of CPOGs from event logs is described in Section 5. This is followed by Section 6, which shows how one can automatically incorporate data into CPOGs. Finally, Section 7 provides a discussion about related and future work.

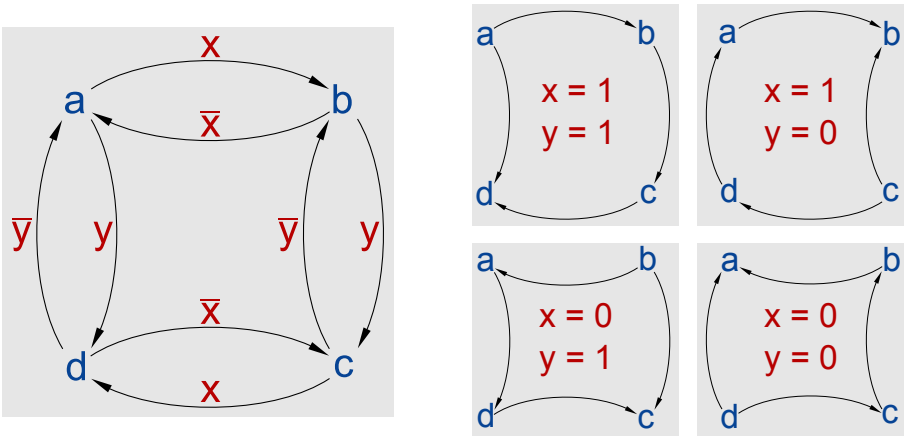
2 Motivating Example

We start by illustrating the reasons that motivate us to study the application of CPOGs in process mining, namely: (i) the ability of CPOGs to compactly represent complex event logs and clearly illustrate their high-level properties, and (ii) the possibility of capturing event log meta data as part of a CPOG representation, thereby taking advantage of the meta data for the purpose of explaining the process under observation.

Consider an event log $L = \{abcd, cdab, badc, dcba\}$. One can notice that the order between events a and b always coincides with the order between events c and d. This is an important piece of information about the process, which however may not be immediately obvious when looking at the log in the text form. To visualise the log one may attempt to use existing process mining techniques and discover a graphical representation for the log, for example in the form of a Petri Net or a BPMN. However, the existing process mining techniques perform very poorly on this log and fail to capture this information. To compare the models discovered from this log by several popular process mining methods, we will describe the discovered behaviour by regular expressions, where operators $||$ and $\cup$ denote interleaving and union, respectively.

The α-algorithm [27] applied to L produces a Petri Net accepting the behaviour $a \cup b \cup c \cup d$, which clearly cannot reproduce any of the traces in L. Methods aimed at deriving block-structured process models [3][13] produce a connected Petri Net that with the help of *silent* transitions reproduces the behaviour $a \,||\, b \,||\, c \,||\, d$, which is a very imprecise model accepting all possible interleavings of the four events. The region-based techniques [4] discover the same behaviour as the block-structured miners, but the derived models are not connected.

CPOGs, however, can represent L exactly and in a very compact form, as shown in Fig. 1(a). Informally, a CPOG is an overlay of several partial orders that can be extracted from it by assigning values to variables that appear in the conditions of the CPOG vertices and arcs, e.g., the upper-left graph shown in Fig. 1(b) (assignment $x = 1$, $y = 1$) corresponds to the partial order containing the causalities $a \prec b$, $a \prec d$, $b \prec c$, $c \prec d$. One can easily verify that the model is precise by trying all possible assignments of variables x and y and checking that they generate the traces $\{abcd, cdab, badc, dcba\}$ as expected, and nothing else. See Fig. 1(b) for the corresponding illustration. The compactness of the CPOG representation of L is due to the fact that several event orderings can be overlayed on top of each other taking advantage of the similarities between them. See Sections 4 and 5 for a detailed introduction to CPOGs and algorithms for automated translation of event logs to CPOGs.



(a) CPOG representation

(b) Four projections of the CPOG

Figure 1: Exact CPOG representation of log $L = \{abcd, cdab, badc, dcba\}$

It is worth mentioning that CPOGs allow us to recognize *second order relations* between events. These are relations that are not relating events themselves, but are relating relations between events: indeed, the CPOG in Fig. 1(a) clearly shows that the relation between a and b is equal to the relation between c and d, and the same holds for pairs (a, d) and (b, c). In principle, one can go even further and consider third order relations and so forth. The practical use of such a relation hierarchy is that it may help to extract an event hierarchy from event logs, thereby simplifying the resulting representation even further.

One may be unsatisfied by the CPOG representation in Fig. 1(a) due to the use of 'artificial' variables x and y. Where do these variables come from and what exactly do they correspond to in the process? We found out that additional data which is often present in event logs can be used to answer such questions. In fact, as we will show in Section 6, it may be possible to use easy-to-understand predicates constructed from the data instead of 'opaque' Boolean variables.

For example, consider the same log L but augmented with temperature data attached to traces:

- $abcd, t = 25°$
- $cdab, t = 30°$
- $badc, t = 22°$
- $dcba, t = 23°$

With this information at hand we can now explain what variable x means. In other words, we can open the previously opaque variable x by expressing it as a predicate involving data parameter t:

$$x = t \geq 25°$$

One can subsequently drop x completely from the CPOG by using conditions $t \geq 25°$ and $t < 25°$ in place of x and $\overline{x}$, respectively. See Fig. 2 for the corresponding illustration.

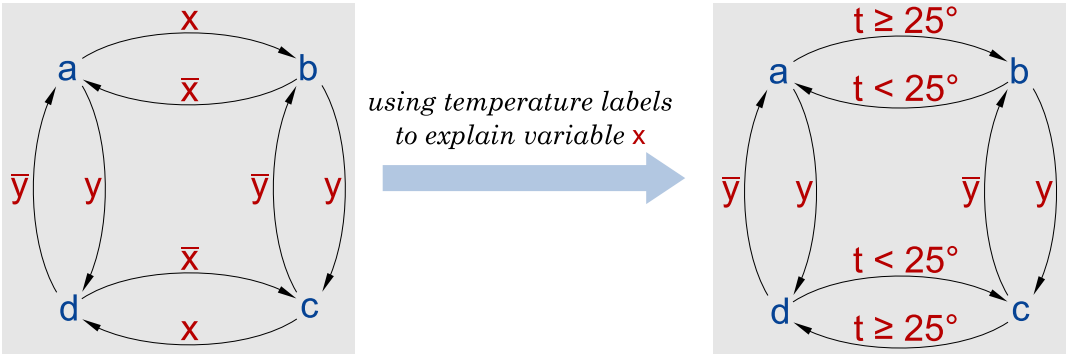


Figure 2: Using data to explain variables

To conclude, we believe that CPOGs bring unique log visualisation capabilities to the process mining field. It is possible to use CPOGs as an intermediate representation of event logs, which can be exact as well as more comprehensible both for humans and for software tools further in the process mining pipeline.

3 Event Logs

In this section we introduce the notion of an *event log*, which is central for this paper and for the process mining field. We also discuss important quality metrics that are typically used to compare methods for event log based process mining.

Table 1 shows a simple event log, which contains not only event information but also *data* in the form of *event attributes*. The example event log matches the log used in the previous section,

Event	Case ID	Activity	Timestamp	Temperature	Resource	Cost	Risk
1	1	a	10-04-2015 9:08am	25.0	Martin	17	Low
2	2	c	10-04-2015 10:03am	28.7	Mike	29	Low
3	2	d	10-04-2015 11:32am	29.8	Mylos	16	Medium
4	1	b	10-04-2015 2:01pm	25.5	Silvia	15	Low
5	1	c	10-04-2015 7:06pm	25.7	George	14	Low
6	1	d	10-04-2015 9:08pm	25.3	Peter	17	Medium
7	2	a	10-04-2015 10:28pm	30.0	George	19	Low
8	2	b	10-04-2015 10:40pm	29.5	Peter	22	Low
9	3	b	11-04-2015 9:08am	22.5	Mike	31	High
10	4	d	11-04-2015 10:03am	22.0	Mylos	33	High
11	4	c	11-04-2015 11:32am	23.2	Martin	35	High
12	3	a	11-04-2015 2:01pm	23.5	Silvia	40	Medium
13	3	d	11-04-2015 7:06pm	28.8	Mike	43	High
14	3	c	11-04-2015 9:08pm	22.9	Silvia	45	Medium
15	4	b	11-04-2015 10:28pm	23.0	Silvia	50	High
16	4	a	11-04-2015 10:40pm	23.1	Peter	35	Medium

Table 1: An example event log

that is the underlying traces are $\{abcd, cdab, badc, dcba\}$ and they correspond to 'case IDs' 1, 2, 3, and 4, respectively. We assume that the set of attributes is fixed and the function *attr* maps pairs of events and attributes to the corresponding values. For each *event* e the log contains the case ID $case(e)$, the activity name $act(e)$, and the set of attributes defined for e, e.g., $attr(e, timestamp)$. For instance, for the event log in Table 1, $case(e_7) = 2$, $act(e_7) = a$, $attr(e_7, timestamp) =$ "10-04-2015 10:28pm", and $attr(e_7, cost) = 19$. Given a set of events E, an *event log* $L \in \mathbb{B}(E^*)$ is a multiset of *traces* E^* of events.

Process mining techniques use *event logs* containing footprints of real process executions for discovering, analysing and extending formal process models, which reveal real processes in a system [26]. The process mining field has risen around a decade ago, and since then it has evolved in several directions, with process discovery being perhaps the most difficult challenge, as demonstrated by the large number of techniques available for it today. What makes process discovery hard is the fact that derived process models are expected to be good across four quality metrics, which are often mutually exclusive:

- *fitness*: the ability of the model to reproduce the traces in the event log (i.e., not too many traces are lost),
- *precision*: the precision of the model in representing the behavior in the log (i.e., not too many new traces are introduced),
- *generalisation*: the ability of the model to generalise the behavior not covered by the log, and
- *simplicity*: the well-known *Occam's Razor* principle that advocates for simpler models.

Although this paper does not focus on the discovery of process models, we will consider these quality metrics when analysing the derived Conditional Partial Order Graphs, which are formally described in the next section.

4 Conditional Partial Order Graphs

Conditional Partial Order Graphs (CPOGs) were introduced for the compact specification of concurrent systems comprised from multiple behavioural scenarios [17]. CPOGs are particularly effective when scenarios of the system share common patterns, which can be exploited for the automated derivation of a compact combined representation of the system's behaviour. CPOGs have been used for the design of asynchronous circuits [20] and for optimal encoding of processor instructions [18]. In this paper we demonstrate how CPOGs can be employed in process mining.

4.1 Basic definitions

A CPOG is a directed graph (V, E), whose *vertices* V and *arcs* $E \subseteq V \times V$ are labelled with Boolean functions, or *conditions*, $\phi : V \cup E \to (\{0,1\}^X \to \{0,1\})$, where $\{0,1\}^X \to \{0,1\}$ stands for a Boolean function defined on Boolean *variables* X.

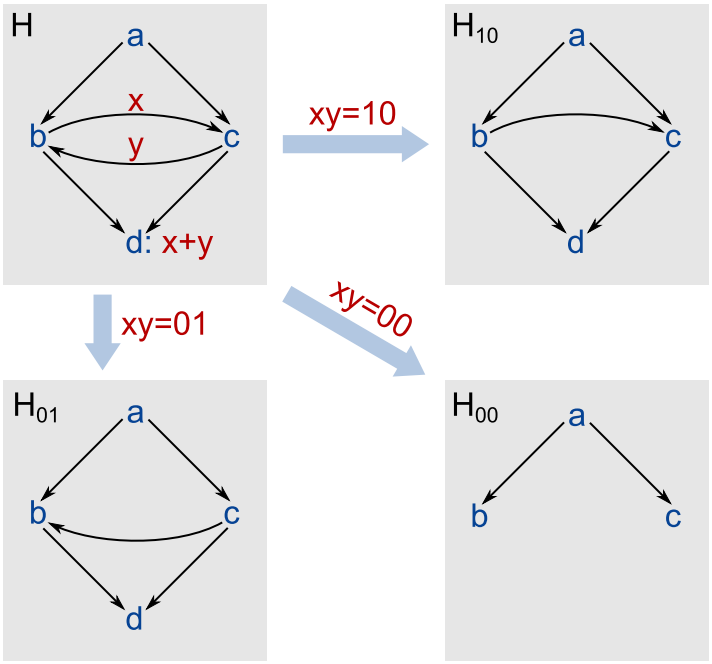


Figure 3: Example of a Conditional Partial Order Graph and the associated family of graphs

Fig. 3 (the top left box) shows an example of a CPOG H containing 4 vertices $V = \{a, b, c, d\}$, 6 arcs and 2 variables $X = \{x, y\}$. Vertex d is labelled with condition $x + y$ (that is, 'x OR y'), arcs (b, c) and (c, b) are labelled with conditions x and y, respectively. All other vertices and arcs are labelled with trivial conditions 1 (trivial conditions are not shown for clarity); we call such vertices and arcs *unconditional*.

There are $2^{|X|}$ possible assignments of variables X, called *codes*. Each code induces a subgraph of the CPOG, whereby all the vertices and arcs, whose conditions evaluate to 0 are removed. For

example, by assigning $x = y = 0$ one obtains graph H_{00} shown in the bottom right box in Fig. 3; vertex d and arcs (b, c) and (c, b) have been removed from the graph, because their conditions are equal to 0 when $x = y = 0$. Different codes can produce different graphs, therefore a CPOG with $|X|$ variables can potentially specify a *family* of $2^{|X|}$ graphs. Fig. 3 shows two other members of the family specified by CPOG H: H_{01} and H_{10}, corresponding to codes 01 and 10, respectively, which differ only in the direction of the arc between vertices b and c.

It is often useful to focus only on a subset $C \subseteq \{0,1\}^X$ of codes, which are meaningful in some sense. For example, code 11 applied to CPOG H in Fig. 3 produces a graph with a loop between vertices b and c, which is undesirable if arcs are interpreted as causality. We use a Boolean *restriction function* $\rho : \{0,1\}^X \to \{0,1\}$ to compactly specify the set $C = \{\mathbf{x} \mid \rho(\mathbf{x}) = 1\}$ and its complement $DC = \{\mathbf{x} \mid \rho(\mathbf{x}) = 0\}$, which are often referred to as the *care* and *don't care* sets [8]. By setting $\rho = \overline{xy}$ one can disallow code $\mathbf{x} = 11$, thereby restricting the family of graphs specified by CPOG H to three members only, which are all shown in Fig. 3.

The *size* $|H|$ of a CPOG $H = (V, E, X, \phi, \rho)$ is defined as:

$$|H| = |V| + |E| + |X| + \left| \bigcup_{z \in V \cup E} \phi(z) \cup \rho \right|,$$

where $|\{f_1, f_2, \ldots, f_n\}|$ stands for the size of the smallest circuit [30] that computes all Boolean functions in set $\{f_1, f_2, \ldots, f_n\}$.

4.2 Families of partial orders

A CPOG $H = (V, E, X, \phi, \rho)$ is *well-formed* if every allowed code $\mathbf{x}$ produces an acyclic graph $H_{\mathbf{x}}$. By computing the transitive closure $H^*_{\mathbf{x}}$ one can obtain a *strict partial order*, an irreflexive and transitive relation on the set of *events* corresponding to vertices of $H_{\mathbf{x}}$.

We can therefore interpret a well-formed CPOG as a specification of a family of partial orders. We use the term *family* instead of the more general term *set* to emphasise the fact that partial orders are *encoded*, that is each partial order $H^*_{\mathbf{x}}$ is paired with the corresponding code $\mathbf{x}$. For example, the CPOG shown in Fig. 3 specifies the family comprising the partial order H^*_{00}, where event a precedes concurrent events b and c, and two total orders H^*_{01} and H^*_{10} corresponding to sequences *acbd* and *abcd*, respectively.

The *language* $\mathcal{L}(H)$ *of a CPOG* H is the set of all possible linearisations of partial orders contained in it. For example, the language of the CPOG shown in Fig. 3 is $\mathcal{L}(H) = \{abc, acb, abcd, acbd\}$. One of the limitations of the CPOG model is that it can only describe finite languages. However, this limitation is irrelevant for the purposes of this paper since event logs are always finite.

It has been demonstrated in [14] that CPOGs are a very efficient model for representing families of partial orders. In particular, they can be exponentially more compact than *Labelled Event Structures* [21] and *Petri Net unfoldings* [15]. Furthermore, for some applications CPOGs provide more comprehensible models than other widely used formalisms, such as *Finite State Machines* and *Petri Nets*, as has been shown in [17] and [20]. This motivated the authors to investigate the applicability of CPOGs to process mining.

4.3 Synthesis

In the previous sections we have demonstrated how one can extract partial orders from a given CPOG. However, the opposite problem is more interesting: *derive the smallest CPOG description*

for a given a set of partial orders. This problem is called *CPOG synthesis* and it is an essential step in the proposed CPOG-based approach to process mining.

A number of CPOG synthesis methods have been proposed to date. In this paper we will rely on the one based on graph colouring [17], which produces CPOGs with all conditions having at most one literal. Having at most one literal per condition is a serious limitation for many applications, but we found that the method works well for process mining. A more sophisticated approach, which produces CPOGs with more complex conditions has been proposed in [18], however, it has poor scalability and cannot be applied to large process mining instances. Both methods are implemented in open-source Workcraft framework [1], which we used in our experiments.

In general, the CPOG synthesis problem is still under active development and new approximate methods are currently being studied, e.g., see [7]. Another promising direction for overcoming this challenge is based on reducing the CPOG synthesis problem to the problem of Finite State Machine synthesis [29].

5 From Event Logs to CPOGs

When visualising behaviour of an event log, it is difficult to identify a single technique that performs well for any given log due to the *representational bias* exhibited by existing process discovery algorithms. For example, if the event log describes a simple workflow behaviour, then the α-algorithm [27] is usually the best choice. However, if non-local dependencies are present in the behaviour, the α-algorithm will not be able to find them, and then other approaches, e.g. based on the theory of regions [4][24][28], may deliver best results. The latter techniques in turn are not tailored for dealing with noise, and alternative approaches such as [9][31] should be considered. There are event logs for which none of the existing process discovery techniques seem to provide a satisfactory result according to the quality metrics presented in Section 3; see the simple event log shown in Section 2 as an example.

In this section we describe two approaches for translating a given event log L into a compact CPOG representation H. The first approach, which we call the *exact CPOG mining*, treats each trace as a totally ordered sequence of events and produces CPOG H such that $L = \mathcal{L}(H)$. The second approach attempts to extract concurrency between the events, hence we call it the *concurrency-aware CPOG mining*. The former approach does not introduce any new behaviours, while the latter one may in fact introduce new behaviours, which could be interpreted as new possible interleavings of the traces contained in the given log, hence producing CPOG H such that $L \subseteq \mathcal{L}(H)$.

5.1 Exact CPOG mining

The problem of the *exact CPOG mining* is formulated as follows: given an event log L, derive a CPOG H such that $L = \mathcal{L}(H)$. This can be trivially reduced to the CPOG synthesis problem. Indeed, each trace $t = e_1 e_2 \cdots e_m$ can be considered a total order of events $e_1 \prec e_2 \prec \cdots \prec e_m$. Therefore, a log $L = \{t_1, t_2, \cdots, t_n\}$ can be considered a set of n total orders and its CPOG representation can be readily obtained via CPOG synthesis.

For example, given event log $L = \{abcd, cdab, badc, dcba\}$ described in Section 2, the exact mining approach produces the CPOG shown in Fig. 1. As has already been discussed in Section 2, the resulting CPOG is very compact and provides a more comprehensible representation of the event log compared to conventional models used in process mining, such as Petri Nets or BPMNs.

When a given event log contains concurrency, the exact CPOG mining approach may lead to suboptimal results. For example, consider a simple event log $L = \{abcd, acbd\}$. If we directly synthesise a CPOG by considering each trace of this log a total order, we will obtain the CPOG H shown in Fig. 4 (left). Although $L = \mathcal{L}(H)$ as required, the CPOG uses a redundant variable x to distinguish between the two total orders even though they are just two possible linearisations of the same partial order, where $a \prec b$, $a \prec c$, $b \prec d$, and $c \prec d$. It is therefore desirable to recognise and extract the concurrency between events b and c in this event log and use the information for simplifying the derived CPOG, as shown in Fig. 4 (right). Note that the simplified CPOG H' still preserves the language equality, i.e. $L = \mathcal{L}(H')$.

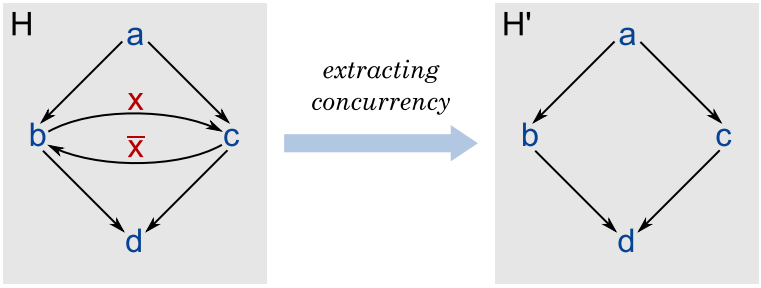


Figure 4: CPOG mining from event log $L = \{abcd, acbd\}$

5.2 Concurrency-aware CPOG mining

This section presents an algorithm for extracting concurrency from a given event log and using this information for simplifying the result of the CPOG mining. Classic process mining techniques generally follow the same principle; in particular, the α-algorithm [26] is often used to extract concurrency in the context of process mining based on Petri Nets. We introduce a new concurrency extraction algorithm, which is more conservative than the α-algorithm: it uses stronger restrictions when declaring two events concurrent, which leads to higher accuracy of process mining. This method works particularly well in combination with CPOGs due to their compactness, however, we believe that it can also be useful in combination with other formalisms.

First, let us introduce convenient operations for extracting subsets of traces from a given event log L. Given an event e, the subset of L's traces containing e will be denoted as $L|_e$, while the subset of L's traces not containing e will be denoted as $L|_{\overline{e}}$. Clearly, $L|_e \cup L|_{\overline{e}} = L$. Similarly, given two events e and f, the subset of L's traces containing both e and f with e occurring before f will be denoted as $L|_{e \to f}$. Note that $L|_e \cap L|_f = L|_{e \to f} \cup L|_{f \to e}$, i.e., if two events appear in a trace, they must be ordered one way or another. For instance, if $L = \{abcd, acbd, abce\}$ then $L_e = \{abce\}$, $L_{\overline{a}} = \emptyset$, $L_{a \to b} = L$, and $L_{a \to d} = \{abcd, acbd\}$. An event e is *conditional* if $L|_e \neq \emptyset$ and $L|_e \neq L$, otherwise it is *unconditional*. A conditional event will necessarily have a non-trivial condition (neither 0 nor 1) in the mined CPOG. Similarly, a pair of events e and f is *conditionally ordered* if $L|_{e \to f} \neq \emptyset$ and $L|_{e \to f} \neq L$. Otherwise, e and f are *unconditionally ordered*.

We say that a conditional event r *indicates* the order between events e and f in an event log L if one of the following criteria holds:

- $L|_r \subseteq L|_{e \to f}$
- $L|_r \subseteq L|_{f \to e}$
- $L|_{\overline{r}} \subseteq L|_{e \to f}$
- $L|_{\overline{r}} \subseteq L|_{f \to e}$

In other words, the existence or non-existence of the event r can be used as an indicator of the order between the events e and f. For example, if $L = \{abcd, acbd, abce\}$, then e indicates the order between b and c. Indeed, whenever we observe event e in a trace we can be sure that b occurs before c in that trace: $L|_e \subseteq L|_{b \to c}$.

Similarly, we say that a conditionally ordered pair of events r and s *indicates* the order between events e and f in an event log L if one of the following criteria holds:

- $L|_{r \to s} \subseteq L|_{e \to f}$
- $L|_{r \to s} \subseteq L|_{f \to e}$
- $L|_{s \to r} \subseteq L|_{e \to f}$
- $L|_{s \to r} \subseteq L|_{f \to e}$

In other words, the order between the events r and s can be used as an indicator of the order between the events e and f. For example, if $L = \{abcd, cdab, badc, dcba\}$, then the order between events a and b indicates the order between events c and d (and vice versa). Indeed, whenever a occurs before b in a trace, we know that c occurs before d: $L|_{a \to b} = L|_{c \to d}$.

The *indicates relation* has been inspired by and is somewhat similar to the *reveals relation* introduced in [10].

We are now equipped to describe the algorithm for concurrency-aware CPOG mining. The algorithm takes an event log L as input and produces a CPOG H such that $L \subseteq \mathcal{L}(H)$.

1. Extract concurrency: find all conditionally ordered pairs of events e and f such that the order between them is not indicated by any other events or pairs of events. Call the resulting set of pairs C.
2. Convert each trace $t \in L$ into a partial order p by relaxing the corresponding total order according to the set of concurrent pairs C. Call the resulting set of partial orders P.
3. Perform the CPOG synthesis on the obtained set of partial orders P to produce the resulting CPOG H.

Note that the resulting CPOG H indeed satisfies the condition $L \subseteq \mathcal{L}(H)$, since we can only add new linearisations into H in step (2) of the algorithm, when we relax a total order corresponding to a particular trace by discarding some of the order relations.

Let us apply the algorithm to the previous examples. Given log $L = \{abcd, cdab, badc, dcba\}$ from Section 2, the algorithm does not find any concurrent pairs, because the order between each pair of events is indicated by the order between the complementary pair of events (e.g., $L|_{a \to b} = L|_{c \to d}$). Hence, $C = \emptyset$ and the result of the algorithm coincides with the exact CPOG mining, as shown in Section 2. Given log $L = \{abcd, acbd\}$ from Section 5.1, the algorithm finds one pair of concurrent events, namely $\{b, c\}$, which results in collapsing of both traces of L into the same partial order with trivial CPOG representation shown in Fig. 4 (right).

6 From Control Flow to Data

As demonstrated in the previous section, one can derive a compact CPOG representation from a given event log using CPOG mining techniques. The obtained representations however rely on

opaque Boolean variables, which make the result difficult to comprehend. For example, Fig. 1(a) provides no intuition on how a particular variable assignment can be interpreted with respect to the process under observation. The goal of this section is to present a method for the automated extraction of useful data labels from a given event log (in particular from available event attributes) and using these labels for constructing 'transparent' and easy-to-comprehend predicates, which can substitute the opaque Boolean variables. This is similar to the application of conventional machine learning techniques for learning 'decision points' in process models or in general for the automated enhancement of a given model by leveraging the available data present in the event log [26].

More formally, given an event log L and the corresponding mined CPOG H our goal is to explain how a particular condition f can be interpreted using data available in the log L. Note that the condition f can be as simple as just a single literal $x \in X$ (e.g., the arc $a \to b$ in Fig. 1(a)), in which case our goal is to explain a particular Boolean variable; however, the technique introduced in this section is applicable to any Boolean function of the CPOG variables $f : \{0,1\}^X \to \{0,1\}$, in particular, one can use the technique for explaining what the restriction function ρ corresponds to in the process, effectively discovering the process *invariants*. We achieve the goal by constructing an appropriate instance of the *classification problem* [16].

Let $n = |E|$ be the number of different events in L, and k be the number of different event attributes available in L. Remember that attributes of an event e can be accessed via function $attr(e)$, see Section 3. Hence, every event e in the log defines a *feature vector* $\hat{e}$ of dimension k where the value at i-th position corresponds to the value of the i-th attribute of e[3]. For instance, the feature vector $\hat{e}_1$ corresponding to the event e_1 in the log shown in Table 1 is ("10-04-2015 9:08am", 25.0, "Martin", 17, Low). Some of the features, e.g. timestamp, may need to be abstracted before applying the technique described below in order to produce better results. For example, timestamps can be mapped to five discrete classes *morning*, *noon*, *afternoon*, *evening* and *night*.

The key observation for the proposed method is that all traces in the log L can be split into two disjoint sets, or *classes*, with respect to the given function f: i) set $L|_f$, containing the traces where f evaluates to 1, and ii) set $L|_{\overline{f}}$ containing the traces where f evaluates to 0. This immediately leads to an instance of the *binary classification problem* on n feature vectors, as illustrated in Table 2.

Feature vectors	Class
$\{\hat{e} \mid e \in \sigma \wedge \sigma \in L\|_f\}$	True
$\{\hat{e} \mid e \in \sigma \wedge \sigma \in L\|_{\overline{f}}\}$	False

Table 2: Binary classification problem for function f and event log L.

In other words, every event belonging to a trace where the function evaluates to 1 is considered to belong to the class we learn, that is, the class labelled as True in Table 2 (the remaining events do not belong to this class). Several methods can be applied to solve this problem, including *decision trees* [23], *support vector machines* [6], and others. In this work we focus on decision trees as they provide a convenient way to extract predicates defined on event attributes, which can be directly used for substituting opaque CPOG conditions. The method is best explained by way of an example.

[3] We assume a total order on the set of event attributes.

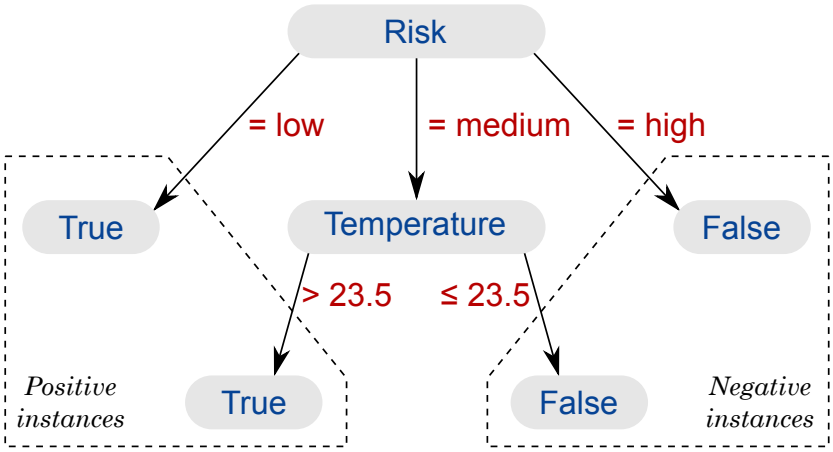


Figure 5: Decision tree built for function $f = x$ in the CPOG of Fig. 1(a).

Consider the event log shown in Table 1, which contains a number of data attributes for each event. The traces underlying the log are $\{abcd, cdab, badc, dcba\}$. Fig. 1(a) shows the corresponding CPOG produced by the CPOG mining techniques presented in the previous section. Let us try to find an interpretation of the variable x by applying the above procedure with $f = x$. The set $L|_f$ equals to $L|_{a \to b}$, i.e. it contains traces 1 and 2, wherein event a occurs before event b and therefore $f = 1$. Therefore, feature vectors $\hat{e}_1$-$\hat{e}_8$ provide the positive instances of the class to learn (the first eight events of the log belong to traces 1 and 2), while feature vectors $\hat{e}_9$-$\hat{e}_{16}$ provide the negative ones. The decision tree shown in Fig. 5 is a possible classifier for this function, which has been derived automatically using machine learning software Weka [11]. By combining the paths in the tree that lead to positively classified instances, one can derive the following predicate for f: $risk = low \vee (risk = medium \wedge temperature > 23.5)$. This predicate can be used to substitute the opaque variable x in the mined CPOG.

One can use the same procedure for deriving the explanation for all variables and/or conditions in the mined CPOG, thereby providing a much more comprehensible representation for the event log. Note that for complementary functions, taking the negation of the classification description will suffice, e.g., conditions $\overline{x}$ in Fig. 1(a) can be substituted with predicate $risk \neq low \wedge (risk \neq medium \vee temperature \leq 23.5)$. Alternatively, one can derive the predicate for a complementary function by combining paths leading to the negative instances; for example, for $f = \overline{x}$ the resulting predicate is $risk = high \vee (risk = medium \wedge temperature \leq 23.5)$.

The learned classifier can be tested for evaluating the quality of representation of the learned concept. If the quality is unacceptable then the corresponding condition may be left unexplained in the CPOG. Therefore in general the data extraction procedure may lead to partial results when the process contains concepts which are 'difficult to learn'. For example, in the discussed case study the condition $f = y$ could not be classified exactly.

A coarse-grain alternative to the technique discussed in this section is to focus on *case attributes* instead of event attributes. Case attributes are attributes associated with a case (i.e., a trace) as a whole instead to individual events [26]. Furthermore, the two approaches can be combined with the aim of improving the quality of obtained classifiers.

7 Discussion

The techniques presented in this paper are currently being implemented as part of the Workcraft framework [1][22], and the next step is to evaluate them on real-life event logs containing data attributes. Several challenges need to be faced, e.g., the complexity of the concurrency extraction algorithm (the first step in the algorithm presented in Section 5.2), the fine-tuning of parameters of the machine learning techniques, and some others.

Due to the inability of CPOGs to directly represent cyclic behavior, we currently only focus on using CPOGs for visualisation and as an intermediate representation of event logs, which can be further transformed into an appropriate process mining formalism, such as Petri Nets or BPMNs. Although some syntactic transformations already exist to transform CPOGs into contextual Petri nets [22], we believe that finding new methods for discovery of process mining models from CPOGs is an interesting direction for future research.

Another research direction is to consider CPOGs as compact algebraic objects that can be used to efficiently manipulate and compare event logs [19]. Since a CPOG corresponding to an event log can be exponentially smaller, this may help to alleviate the memory requirements bottleneck for current process mining tools that store 'unpacked' event logs in memory.

Event logs are not the only suitable input for the techniques presented in this paper: we see an interesting link with the work on *discovery of frequent episodes*, e.g., as reported recently in [12]. *Episodes* are partially ordered collections of events (not activities), and as such they can also be represented by CPOGs. This may help to compress the information provided by frequent episodes, especially if one takes into account the fact that current algorithms may extract a large number of episodes, which then need to be visualised for human understanding.

8 Conclusions

This paper describes the first steps towards the use of CPOGs in the field of process mining. In particular, the paper presented the automatic derivation of the control flow part of the CPOG representation from a given event log, and then the incorporation of meta data contained in the log as conditions of the CPOG vertices and arcs. We have implemented some of the reported techniques, in particular the extraction of a CPOG from an event log as described in Section 5, and some preliminary experiments have been carried out.

The future work includes addressing the challenges described in the previous section, as well as a thorough practical evaluation of the algorithms described in this paper. The developed software tool may then be used within a more general framework such as ProM [2], Workcraft [1] or PMLAB [5].

Acknowledgments. This work as been partially supported by funds from the Spanish Ministry for Economy and Competitiveness (MINECO) and the European Union (FEDER funds) under grant COMMAS (ref. TIN2013-46181-C2-1-R).

References

1. The Workcraft framework homepage. http://www.workcraft.org/, 2009.
2. The ProM framework homepage. http://www.promtools.org/, 2010.

3. Joos C. A. M. Buijs, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. A genetic algorithm for discovering process trees. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2012, Brisbane, Australia, June 10-15, 2012*, pages 1–8, 2012.
4. Josep Carmona, Jordi Cortadella, and Michael Kishinevsky. New region-based algorithms for deriving bounded Petri nets. *IEEE Trans. Computers*, 59(3):371–384, 2010.
5. Josep Carmona and Marc Solé. PMLAB: an scripting environment for process mining. In *Proceedings of the BPM Demo Sessions 2014 Co-located with the 12th International Conference on Business Process Management (BPM 2014), Eindhoven, The Netherlands, September 10, 2014.*, pages 16 – 21, 2014.
6. Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.
7. A. de Gennaro and P. Stankaitis. Efficient encoding of instructions of ARM Cortex M0+ processor. Technical report, Newcastle University, 2015.
8. G. de Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.
9. Christian W. Günther and Wil M. P. van der Aalst. Fuzzy mining - adaptive process simplification based on multi-perspective metrics. In Gustavo Alonso, Peter Dadam, and Michael Rosemann, editors, *BPM*, volume 4714 of *Lecture Notes in Computer Science*, pages 328–343. Springer, 2007.
10. Stefan Haar, Christian Kern, and Stefan Schwoon. Computing the reveals relation in occurrence nets. *Theor. Comput. Sci.*, 493:66–79, 2013.
11. Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.
12. Maikel Leemans and Wil M. P. van der Aalst. Discovery of frequent episodes in event logs. In *Proceedings of the 4th International Symposium on Data-driven Process Discovery and Analysis (SIMPDA 2014), Milan, Italy, November 19-21, 2014.*, pages 31–45, 2014.
13. Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. Discovering block-structured process models from event logs - A constructive approach. In *Application and Theory of Petri Nets and Concurrency - 34th International Conference, PETRI NETS 2013, Milan, Italy, June 24-28, 2013. Proceedings*, pages 311–329, 2013.
14. H. Ponce De Leon and A. Mokhov. Building bridges between sets of partial orders. In *International Conference on Language and Automata Theory and Applications (LATA)*, 2015.
15. KL McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proceedings of Computer Aided Verification conference (CAV)*, volume 663, page 164, 1992.
16. Tom M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997.
17. A. Mokhov. *Conditional Partial Order Graphs*. PhD thesis, Newcastle University, 2009.
18. A. Mokhov, A. Alekseyev, and A. Yakovlev. Encoding of processor instruction sets with explicit concurrency control. *Computers & Digital Techniques, IET*, 5(6):427–439, 2011.
19. A. Mokhov and V. Khomenko. Algebra of Parameterised Graphs. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):143, 2014.
20. A. Mokhov and A. Yakovlev. Conditional Partial Order Graphs: Model, Synthesis, and Application. *IEEE Transactions on Computers*, 59(11):1480–1493, 2010.
21. M. Nielsen, G. D. Plotkin, and G. Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13:85–108, 1981.
22. Ivan Poliakov, Danil Sokolov, and Andrey Mokhov. Workcraft: a static data flow structure editing, visualisation and analysis tool. In *Petri Nets and Other Models of Concurrency–ICATPN 2007*, pages 505–514. Springer, 2007.
23. J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.
24. Marc Solé and Josep Carmona. Light region-based techniques for process discovery. *Fundam. Inform.*, 113(3-4):343–376, 2011.
25. Minseok Song and Wil MP van der Aalst. Supporting process mining by showing events at a glance. *Proceedings of the 17th Annual Workshop on Information Technologies and Systems (WITS)*, pages 139–145, 2007.

26. Wil van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.
27. Wil M. P. van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE TKDE*, 16(9):1128–1142, 2004.
28. Jan Martijn E. M. van der Werf, Boudewijn F. van Dongen, Cor A. J. Hurkens, and Alexander Serebrenik. Process discovery using integer linear programming. In *ATPN*, pages 368–387, 2008.
29. Tiziano Villa, Timothy Kam, Robert K Brayton, and Alberto L Sangiovanni-Vincentelli. *Synthesis of finite state machines: logic optimization*. Springer Publishing Company, Incorporated, 2012.
30. Ingo Wegener. *The Complexity of Boolean Functions*. Johann Wolfgang Goethe-Universitat, 1987.
31. A.J.M.M. Weijters, W.M.P. van der Aalst, and A.K. Alves de Medeiros. Process mining with the heuristics miner-algorithm. Technical Report WP 166, BETA Working Paper Series, Eindhoven University of Technology, 2006.