*Consiglio Nazionale delle Ricerche*
*Istituto di Calcolo e Reti ad Alte Prestazioni*

# Process Discovery under Precedence Constraints

Gianluigi Greco, Antonella Guzzo,
Francesco Lupia, Luigi Pontieri

**RT-ICAR-CS-13-03** **Settembre 2013**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.na.icar.cnr.it*
– Sezione di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.pa.icar.cnr.it*

# Process Discovery under Precedence Constraints

GIANLUIGI GRECO, University of Calabria
ANTONELLA GUZZO, University of Calabria
FRANCESCO LUPIA, University of Calabria
LUIGI PONTIERI, ICAR-CNR

Process discovery has recently emerged as a powerful approach to support the analysis and the design of complex processes. It consists of analyzing a set of traces registering the sequence of tasks performed along several enactments of a transactional system in order to build a process model that can explain all the episodes recorded over them. An approach to accomplish this task is presented which can benefit of the background knowledge that, in many cases, is available to the analysts taking care of the process (re-)design. The approach is based on encoding the information gathered from the log and the (possibly) given background knowledge in terms of *precedence constraints*, i.e., of constraints over the topology of the resulting process models. Mining algorithms are eventually formulated in terms of reasoning problems over precedence constraints, and the computational complexity of such problems is thoroughly analyzed by tracing their tractability frontier. Solution algorithms are proposed and their properties analyzed. These algorithms have been implemented in a prototype system, and results of a thorough experimental activity are discussed.

Categories and Subject Descriptors: H.2.8 [**Database Management**]: Database Applications—*Data mining*; D.2.2 [**Software Engineering**]: Design Tools and Techniques; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Logic and constraint programming*

General Terms: Algorithms

Additional Key Words and Phrases: Process mining, graph analysis, computational complexity.

## 1. INTRODUCTION

### 1.1. An Overview of Process Discovery

*Process mining* aims to discover, monitor and improve real processes by extracting knowledge from the event logs that are made available by today's information systems [van der Aalst 2011]. A prominent process mining task is *process discovery*, whose goal is to facilitate the re-design phase and the implementation of complex process models. Indeed, process discovery algorithms are devoted to automatically deriving a model that can explain all the episodes recorded in an event *log* collected by an information system while the activities of an underlying process are executed. Eventually, the "mined" model can be used to design a detailed process model suited to be

A preliminary version of parts of this paper was presented at the 20th European Conference on Artificial Intelligence [Greco et al. 2012].
Authors' full affiliations and addresses: Gianluigi Greco, Dipartimento di Matematica e Informatica - University of Calabria, Via Pietro Bucci 30B, 87036 Rende (CS), Italy, E-mail: ggreco@mat.unical.it. Antonella Guzzo, Dipartimento DIMES - University of Calabria, Via Pietro Bucci 42C, 87036 Rende (CS), Italy, E-mail: guzzo@dimes.unical.it. Francesco Lupia, Dipartimento DIMES - University of Calabria, Via Pietro Bucci 42C, 87036 Rende (CS), Italy, E-mail: lupia@dimes.unical.it. Luigi Pontier, ICAR-CNR, Via Pietro Bucci 41C, 87036 Rende (CS), Italy, E-mail: pontieri@icar.cnr.it.

DOI:http://dx.doi.org/10.1145/0000000.0000000

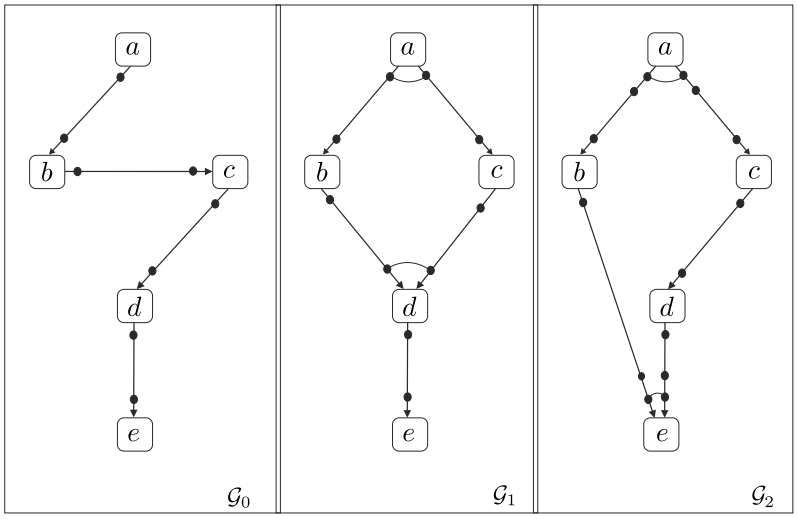



Fig. 1. Process models in the running example.

supported in a workflow management system, or to shed light on the actual process behavior for optimization purposes (for instance, by singling out deviations between the conceptual model and the behavior that is actually registered).

From a conceptual viewpoint, discovery algorithms carry out two different sub-tasks. First, they analyze the traces registering the sequences of activity executions, by mining the causal dependencies that are likely to hold among them. In particular, they present these dependencies in form of *dependency graphs*, that is, directed graphs whose nodes one-to-one correspond with the activities and such that an edge from an activity $a$ to an activity $b$ means that, in some enactment, we expect that an actual flow of information can occur from $a$ to $b$. Second, they enrich dependency graphs with advanced facets of process enactments (such as synchronization and branching constructs, duplicate activities, and invisible activities, just to name a few) and return process models formalized in expressive modeling languages.

EXAMPLE 1.1. Consider the set $\mathcal{A} = \{a, b, c, d, e\}$ of activities, and the structures depicted in Figure 1. Basically, we have three dependency graphs, $\mathcal{G}_0$, $\mathcal{G}_1$, and $\mathcal{G}_2$, which are meant to encode the causal relationships that hold over the activities in $\mathcal{A}$. Moreover, these graphs are adorned with an intuitive notation that expresses routing constructs over them. For instance, in $\mathcal{G}_1$, the flow of execution is split after $a$ over two branches that are to be executed in parallel, and which are eventually synchronized by the activity $d$—see Section 2 for the formalization of the semantics and the notation.

Assume now that the two traces $abcde$ and $acbde$ over $\mathcal{A}$ are given as input to a process discovery algorithm. Then, the graph $\mathcal{G}_0$ will be hardly returned as output, as it does not model the flow associated with the trace $acbde$, where $b$ is executed after $c$.

Instead, $\mathcal{G}_1$ and $\mathcal{G}_2$ are good candidates for a dependency graph supporting the two traces. The crucial observation here is that $b$ is executed before $c$ in one trace, while $b$ is executed after $c$ in the other. This likely witnesses that the two activities are not related by any causal dependency, and that they are executed in "parallel", i.e., over different branches of the process. Moreover, one might argue that the graph $\mathcal{G}_1$, together with the associated routing constraints, is more appropriate than $\mathcal{G}_2$ for being returned as output. Indeed, there is a heuristic evidence in the two given traces that $b$ and $d$ are not parallel, since $b$ seems to be a pre-requisite for the execution of $d$. $\lhd$

Several process models can in general be associated with a log of traces given as input. Ideally, one might select the most appropriate one among them under the assumption that the log is *complete*, i.e., that it registers all the possible traces for the underlying process. Indeed, in this case, if an activity always precedes another, then we can safely discard those models where such activities are executed over different branches. For instance, if we assume that the traces $abcde$ and $acbde$ of Example 1.1 are the only possible ones, then we can safely conclude that $b$ and $d$ are not parallel activities, hence discarding the graph $\mathcal{G}_2$ in Figure 1.

Log completeness has received considerable attention in the literature, and it is a crucial assumption under which a number of discovery methods can be proven to be correct (see, e.g., [van der Aalst et al. 2004; van der Aalst et al. 2003]). In fact, process discovery is often carried out via heuristics approaches, for which the quality of the resulting models grows with the fraction of the traces given at hand over all possible traces for the process. It follows that the quality can be rather poor in those cases where logs are far from being complete due, for instance, to the following reasons:

*Temporal bias:* First, two parallel activities may always appear in the same relative order, simply because one of them always finishes after the other, due to a different duration of them or of some of their predecessor activities. For instance, even for a process that conforms with the schema $\mathcal{G}_2$, we may find no trace where $d$ occurs before $b$, just because $c$ is time consuming, so that $b$ is always completed before $d$.

*Combinatorial explosion:* Second, log completeness might not hold just because the process has not been enacted for a sufficient number of times. Indeed, the number of possible traces grows exponentially w.r.t. the number of activities that can be executed in parallel. For instance, for a process with $n$ branches each one involving $m$ activities, we have more than $n!^m$ possible traces. Thus, for real life processes (even with just two branches and about 20 activities), the number of combinations immediately leads to more than one billion of enactments, which in many application domains are unrealistic.

Because of the issues illustrated above, process discovery techniques are still at an early stage of adoption within enterprises. Indeed, analysts are likely to prefer traditional "top-down" design approaches, where models are eventually built by refining and formalizing a number of desiderata and specifications reflecting the prior knowledge they possess about the process to be automatized. For example, by a-priori knowing that $b$ and $d$ are parallel activities, the analyst can immediately discard $\mathcal{G}_1$ in Example 1.1, even though no trace is given where $d$ actually occurred before $b$.

### 1.2. Bottom-Up vs Top-Down Design Methods

Several process discovery approaches have already been proposed in the literature, mainly differing in the kinds of modeling language they support. For example, processes are intuitively represented in [Agrawal et al. 1998; Weijters and van der Aalst 2001; Weijters and van der Aalst 2003; A.J.M.M. Weijters et al. 2006; Greco et al. 2006; Chen and Yun 2003] via pure directed graphs, while more expressive representations are used in other proposals, ranging from expression tree models [Schimm 2003] and block-structured workflow models [Herbst and Karagiannis 2000; 2003; Hammori et al. 2006], to special classes of Petri-nets [van der Aalst and van Hee 2002; van der Aalst et al. 2003; van der Aalst et al. 2004; de Medeiros et al. 2004; Medeiros et al. 2007]. Moreover, moving from the observation that extracting a single process model may lead to over-generalized process models that mix together different usage scenarios, such classical discovery algorithms have been often combined with methods for clustering log traces, so that a set of process models can be returned as output. Indeed, this allows for improving the precision of the underlying algorithms, by capturing—

from an abstract perspective—constraints that are beyond the expressiveness of their associated modeling languages (cf. [Greco et al. 2006]).

Despite the technical differences that emerge from the non-exhaustive list of proposals discussed above, it must be pointed out that all of them share the idea of mining process models by gathering statistics from the data and by processing them via heuristics. In particular, these "bottom-up" discovery techniques are not capable of taking into account prior knowledge on the underlying process. As a result, over logs that are not complete, mined models may well violate conceptual specifications and domain-constraints (recall that, in Example 1.1, $\mathcal{G}_1$ would be the most probable outcome of a discovery algorithm), hence turning out to be useless in real-life applications.

In fact, the need of defining process discovery algorithms that can take advantage of prior knowledge has been firstly argued by Goedertier et al. [2009]. In particular, the AGNEs (Artificial Generation of Negative Events) technique is presented, which is a mining algorithm founding on an Inductive Logic Programming (ILP) classification-oriented learner. The algorithm is articulated in four steps: First, temporal constraints are extracted from the input log, in order to capture local dependencies, non-local dependencies, and parallelism relationships. Second, the input log and the temporal constraints are used to generate negative examples, i.e., for each prefix of any trace, negative events are generated stating which activities are not allowed to be executed later on in that trace. Third, by using input log traces (as positive examples) together with the artificial negative events, a logic program is induced to predict whether any given activity is allowed to occur at a given position of a given sequence. An important peculiarity of AGNEs is that, in the first of the above four phases, domain experts can directly provide background knowledge. In particular, it is possible to state that two activities are parallel (resp., not parallel), and that one precedes/succeeds (resp., does not precede/succeed) the other.

Another process discovery method (partially) taking into account domain knowledge has been proposed by van der Werf et al. [2009]. As a reference model, Petri nets are considered. Starting with the most liberal (and overgeneralized) net for a given log, with as many transitions as the process activities and with no place, a more refined workflow model is obtained by iteratively adding a new place for restricting the allowed behavior. Each place is chosen greedily, by solving a system of integer linear inequalities, asking for a place with a minimal (resp., maximal) number of incoming edges (resp., outgoing edges). To curb the growth of the mined model, the search of the places is guided by the causal dependencies derived from the log (by using the metrics of van der Aalst et al. [2004]). Moreover, in its implementation in the *ProM* framework [van Dongen et al. 2005], users are allowed to enforce finer grain constraints, by manually modifying the basic activity dependencies extracted from the log, prior to deriving a novel (refined) workflow model. By this way, the learning process can benefit of available domain knowledge expressed in terms of edge constraints and of constraints enforcing parallelism between pairs of activities.

### 1.3. Contribution

As it emerged from the analysis carried out in the above section, top-down design methods and bottom-up process discovery techniques have been almost separate worlds, so far. Indeed, while the use of background knowledge to improve the quality of results has already been considered in a number of traditional data mining tasks (on relational data and on sequence data) including pattern mining [Guns et al. 2011] and clustering [de Amo and Furtado 2007], designing counter-parts of such techniques in the context of process discovery is still largely an open issue.

In this paper, we move a further step to synergically integrate top-down design methods and bottom-up process discovery techniques. Indeed, we propose a "hybrid" ap-

Table I. Summary of constraint support.

| | Traditional | [Goedertier et al. 2009] | [van der Werf et al. 2009] | Here |
|---|---|---|---|---|
| edge constraints | □ | ■ | ■ | ■ |
| path constraints | □ | □ | □ | ■ |
| constraints for parallelism | □ | ■ | ■ | ■ |
| constraints over sets | □ | □ | □ | ■ |

proach to process discovery, where a learning method is conceived which can take into account a wide variety of constraints over the causal dependencies that are possibly available to the analyst. This is particularly useful in order to circumvent the problems emerging when log completeness does not hold. In more detail,

**(1)** We propose a formal framework to specify additional properties on the process models that can be produced as output by process discovery algorithms. The framework is based on defining a set of *precedence constraints* over the activities, and supports the kind of prior knowledge that is usually available to the analyst. Table I summarizes the features of our framework, by comparing them with those supported (to some extent) by earlier approaches in the literature. Note that "traditional" process discovery methods do not provide any support to deal with prior knowledge.

**(2)** In the light of our formulation, process discovery can be conceptually viewed as a *mining task* (i.e., building all possible models for a given input log) followed by a *reasoning task* (i.e., filtering out those models that do not satisfy the precedence constraints defined by the analyst). However, exponentially many process models might be built in general as a result of the mining phase, hence making a literal implementation of such a two-phase approach unfeasible. In fact, we identify relevant classes of constraints where the two tasks can be synergically addressed, and where process discovery can be efficiently carried out.

**(3)** We analyze the computational complexity of the proposed setting, by taking into account various qualitative properties regarding the kinds of constraint being allowed, and by tracing the tractability frontier w.r.t. them. In particular, we show that for the classes of constraints that are not covered by the algorithms discussed in the point **(2)** above, an efficient solution algorithm is unlikely to exist at all, because process discovery turns out to be NP-hard over them.

**(4)** All the algorithms discussed in the paper have been implemented and integrated in a prototype system, which is made available as a plug-in for the well-known process mining suite *ProM* [van Dongen et al. 2005]. In particular, to face the above intractability results, the efficient solution algorithms originally conceived for special cases only are generalized by making them applicable as heuristic solution approaches over arbitrary classes of constraints. Case studies are illustrated, and results for the experimental activity we have conducted in order to validate the effectiveness of the proposed approach are also reported.

**Organization.** The rest of the paper is organized as follows. Section 2 illustrates preliminaries on process logs and models. Precedence constraints are formalized and analyzed in Section 3, while their complexity is studied in Section 4. Efficient solution approaches are illustrated in Section 5. Experimental validation and some concluding remarks are reported in Section 6and Section 7, respectively.

## 2. CAUSAL NETS AND LOGS

In this section, we illustrate and point out basic results about the process modeling language we shall adopt in the rest of the paper.

Several languages have been proposed in the literature which are tailored to the design and the analysis of processes, such as *Petri nets* [van der Aalst 1998] or *event*

*driven process chains* EPCs [van der Aalst et al. 2002; Keller et al. 1992]. Given the focus of the paper, it is convenient to adopt a language that is closer to the needs of process mining applications. Accordingly, our choice is to consider the language of *causal nets* [van Der Aalst et al. 2011], providing a favorable representational bias for such applications while having the same expressiveness as the language of Petri nets.

### 2.1. Preliminaries

Let us hereinafter assume that $\mathcal{A}$ is a given alphabet of symbols, univocally identifying the *activities* of some underlying process. The set $\mathcal{A}$ contains two distinguished activities $a_\perp$ and $a_\top$, called the *starting* and the *terminating* activity, respectively.

A *dependency graph* (over $\mathcal{A}$) is a directed graph $\mathcal{G} = (V, E)$ whose nodes are the activities in the set $V \subseteq \mathcal{A}$, with $V \supseteq \{a_\perp, a_\top\}$, and whose edges in $E \subseteq V \times V$ encode the causal relationships that hold over them. In particular, for each activity $a \in V \setminus \{a_\perp, a_\top\}$, it must be the case that $a$ occurs in some path from $a_\perp$ to $a_\top$. Moreover, $a_\perp$ and $a_\top$ have no ingoing and outgoing edges, respectively.

EXAMPLE 2.1. Consider again the three graphs $\mathcal{G}_0$, $\mathcal{G}_1$, and $\mathcal{G}_2$ depicted in Figure 1. It is immediate to check that they are dependency graphs over the set $\mathcal{A} = \{a, b, c, d, e\}$ of activities. In particular, $a$ (resp., $e$) is the starting (resp., terminating) activity. $\lhd$

A *causal net* (over $\mathcal{A}$) is a tuple $\mathcal{C} = \langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ where $\mathcal{G} = (V, E)$ is a dependency graph and where $\mathcal{I}$ and $\mathcal{O}$ are two functions such that:

— $\mathcal{I}$ maps each activity $a \in V$ to the set $\mathcal{I}(a)$ of the *input bindings* for $a$. For any $a \in V \setminus \{a_\perp\}$, an input binding $ib \in \mathcal{I}(a)$ is a non-empty set of edges such that $ib \subseteq \{(x, a) \mid (x, a) \in E\}$, while the empty set $\emptyset$ is the only input binding for $a_\perp$, i.e., $\mathcal{I}(a_\perp) = \{\emptyset\}$. For each $a \in V$, $\bigcup_{ib \in \mathcal{I}(a)} ib = \{(x, a) \mid (x, a) \in E\}$ must hold.

— $\mathcal{O}$ maps each activity $a \in V$ to the set $\mathcal{O}(a)$ of the *output bindings* for $a$. For any $a \in V \setminus \{a_\top\}$, an output binding $ob \in \mathcal{O}(a)$ is a non-empty set of edges such that $ob \subseteq \{(a, y) \mid (a, y) \in E\}$, while the empty set $\emptyset$ is the only output binding for $a_\top$, i.e., $\mathcal{O}(a_\top) = \{\emptyset\}$. For each $a \in V$, $\bigcup_{ob \in \mathcal{O}(a)} ob = \{(a, y) \mid (a, y) \in E\}$ must hold.

Intuitively, input bindings are meant to encode the pre-conditions for the execution of an activity, while output bindings are meant to encode the effects of this execution.

EXAMPLE 2.2. Consider the causal net $\mathcal{C}_0 = \langle \mathcal{G}_0, \mathcal{I}_0, \mathcal{O}_0 \rangle$, where $\mathcal{G}_0 = (\{a, b, c, d, e\}, E_0)$ is the graph in Figure 1, and where $\mathcal{I}_0(z) = \{ib_z\}$ and $\mathcal{O}_0(z) = \{ob_z\}$ are such that $ib_z = \{(x, z) \mid (x, z) \in E_0\}$ and $ob_z = \{(z, y) \mid (z, y) \in E_0\}$, for each $z \in \{a, b, c, d, e\}$. The causal net models that the execution of $z$ can start as soon as its predecessor activity in $\mathcal{G}_0$ is completed—of course, this is immaterial for the starting activity $a$, which is such that $ib_a = \emptyset$. In fact, after its execution and if $z \neq e$, we have that $z$ enables the execution of its unique successor in $\mathcal{G}_0$.

Similarly, consider the causal net $\mathcal{C}_1 = \langle \mathcal{G}_1, \mathcal{I}_1, \mathcal{O}_1 \rangle$, where $\mathcal{G}_1 = (\{a, b, c, d, e\}, E_1)$ is the graph in Figure 1, and where $\mathcal{I}_1(z) = \{ib'_z\}$ and $\mathcal{O}_1(z) = \{ob'_z\}$, for each activity $z$, are such that: $ib'_a = \emptyset$, $ib'_b = \{(a, b)\}$, $ib'_c = \{(a, c)\}$, $ib'_d = \{(b, d), (c, d)\}$, $ib'_e = \{(d, e)\}$, $ob'_a = \{(a, b), (a, c)\}$, $ob'_b = \{(b, d)\}$, $ob'_c = \{(c, d)\}$, $ob'_d = \{(d, e)\}$, and $ob'_e = \emptyset$. Note that after its execution, the activity $a$ enables both $b$ and $c$ (in parallel), and that the flow is synchronized by $d$, whose execution is possible only after $b$ and $c$ are both completed.

Note also that the symbols adorning the edges of the graphs in Figure 1 precisely correspond with the input and the output bindings. Formally, if $\langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ is a causal net, then each input binding $ib \in \mathcal{I}(z)$ (resp., output binding $ob \in \mathcal{O}(z)$) is represented by marking the edges in $ib$ (resp., in $ob$) and by linking all such markings with a line. For instance, in the graph $\mathcal{G}_1$, the edges $(a, b)$ and $(a, c)$ are marked, and these markings are linked together, hence meaning that the output binding $\{(a, b), (a, c)\}$ occurs in $\mathcal{O}(a)$.

For completeness, we point out that any given activity can be associated with more than one input/output binding, in general. As an example, consider the causal net $\mathcal{C}_2 = \langle \mathcal{G}_2, \mathcal{I}_2, \mathcal{O}_2 \rangle$ associated with the graph $\mathcal{G}_2$ shown in Figure 1. Then, we may note for instance that $\mathcal{O}_3(a) = \{\{(a,b)\},\{(a,c)\},\{(a,b),(a,c)\}\}$ holds. Indeed, the edges $(a,b)$ and $(a,c)$ are linked together as in the case of $\mathcal{G}_1$, but we additionally have also the singleton markings now. Thus, $a$ can either activate $b$, or $c$, or even both of them. $\lhd$

Formally, the semantics of causal nets is next given in terms of those instantiations that are "globally valid". This is different from the operational semantics of design-oriented modeling languages, such as the token-game semantics of Petri nets.

A *binding activity* for the causal net $\mathcal{C}$ is a tuple $\langle a, ib, ob \rangle$, where $a \in V$ is an activity and where $ib \in \mathcal{I}(a)$ and $ob \in \mathcal{I}(a)$ hold. A sequence $\sigma$ of binding activities $\langle a_1 = a_\perp, ib_1, ob_1 \rangle, ..., \langle a_n = a_\top, ib_n, ob_n \rangle$ is called a *binding sequence*. The *state* $S^\sigma_j$ of $\mathcal{C}$ at the $j$-th step of $\sigma$ is defined inductively as the multi-set[1] of edges such that $S^\sigma_0 = \emptyset$, and $S^\sigma_j = S^\sigma_{j-1} \cup ob_j \setminus ib_j$, for each $j \in \{1, ..., len(t)\}$. The sequence $\sigma$ is *valid* w.r.t. $\mathcal{C}$ if $S^\sigma_n = \emptyset$ and $ib_j \subseteq S^\sigma_{j-1}$, for each $j \in \{1, ..., n\}$.

EXAMPLE 2.3. Consider the net $\mathcal{C}_1$ discussed in Example 2.2, and the binding sequence $\sigma = \sigma_1\sigma_2...\sigma_5$ such that: $\sigma_1 = \langle a, \{\}, \{(a,b),(a,c)\}\rangle$, $\sigma_2 = \langle b, \{(a,b)\}, \{(b,d)\}\rangle$, $\sigma_3 = \langle c, \{(a,c)\}, \{(c,d)\}\rangle$, $\sigma_4 = \langle d, \{(b,d),(c,d)\}, \{(d,e)\}\rangle$, and $\sigma_5 = \langle e, \{(d,e)\}, \{\}\rangle$. Note that $S^\sigma_0 = \emptyset$, $S^\sigma_1 = \{(a,b),(a,c)\}$, $S^\sigma_2 = \{(a,c),(b,d)\}$, $S^\sigma_3 = \{(b,d),(c,d)\}$, $S^\sigma_4 = \{(d,e)\}$, and $S^\sigma_5 = \emptyset$. In fact, $\sigma$ is valid w.r.t. $\mathcal{C}_1$. $\lhd$

Transactional systems store partial information about binding sequences, by tracing the events related to the execution of the various activities. Formally, a *trace* $t$ (over $\mathcal{A}$) has the form $t[1]t[2]...t[n]$, with $t[i] \in \mathcal{A}$ being an activity, for each $i \in \{1, ..., n\}$, and with $n$ being the *length* of $t$. W.l.o.g., we shall hereinafter assume that $t[1] = a_\perp$, $t[n] = a_\top$, and that for each $i \in \{2, ..., n-1\}$, $t[i] \cap \{a_\perp, a_\top\} = \emptyset$. Indeed, we may view $a_\perp$ and $a_\top$ as two virtual activities, which we add to $t$ in order to satisfy this requirement. The length of $t$ is also denoted as $len(t)$. A multi-set $L$ of traces is hereinafter just called a *log*, and the set of all the activities occurring over the traces in $L$ is denoted by $\mathcal{A}(L)$.

Note that, as commonly done in the literature, we are considering an abstract view of a log, where we focus on the ordering of the execution (in particular, completion) of the various activities, by getting rid of all information about *(i)* timings (e.g., starting times and durations) and about *(ii)* the data involved in them.

Concerning the first assumption, note that our process modeling language is not capable of supporting temporal information. In fact, very few mining approaches have been proposed that are able to discover timed models, such as for instance stochastic Petri nets (see, e.g, [Anastasiou et al. 2011; HU et al. 2011]). Therefore, in our setting, information about timings in the log can be helpful to a limited extent only. For instance, we can immediately conclude that two activities are parallel when one starts before the other is completed. Moreover, for two activities $a$ and $b$ that are not parallel, in order to assess how likely $a$ is a pre-requisite for the execution of $b$, in addition of their relative positions (see Section 5), we can consider the time elapsed between the completion of $a$ and the starting of $b$. In the paper, however, we will not expand on these standard heuristics (conceived for models that do not support temporal information), by referring the interested reader, e.g., to the work by Wen et al. [2009]. Here, we stress instead that it is an interesting avenue for further research to extend the features of causal nets by directly incorporating timing information and to explore how our techniques can be modified as to deal with the resulting model.

[1]Hereinafter, set operations are transparently applied to multi-sets with the usual intended meaning.

Concerning the second assumption, we point out that even the adaptation of basic process discovery algorithms to *multi-dimensional* settings, where modeling languages have again to be extended (in this latter case to deal with data about activity executions), has been only partially explored in the literature (see, e.g., [Greco et al. 2007]). In fact, this is still largely an open research issue, so that exporting our results to such richer settings is outside the scope of the paper, while constituting another interesting avenue for further researcher.

Now that we have clarified the assumptions underlying our setting, we can proceed to formalize when a log can be considered as the result of the enactments of a given process model. To this end, we say that a causal net $\mathcal{C}$ *supports* a trace $t$ if there is a binding sequence $\sigma$ that is valid w.r.t. $\mathcal{C}$ and where the $j$-th binding activity $\langle a_j, ib_j, ob_j \rangle$ of $\sigma$, for each $j \in \{1, ..., len(t)\}$, is such that $a_j = t[j]$. Moreover, we say that the causal net $\mathcal{C}$ *supports* a log $L$, denoted by $\mathcal{C} \vdash L$, if $\mathcal{C}$ supports each trace $t \in L$.

EXAMPLE 2.4. The causal net $\mathcal{C}_1$ discussed in Example 2.2 supports the trace $abcde$, as it is witnessed by the binding sequence $\sigma$ illustrated in Example 2.3. Moreover, it can be checked that $\mathcal{C}_1$ supports the trace $abcde$, and that no further trace is supported.

Consider instead the causal network $\mathcal{C}_2$, again discussed in Example 2.2. Recall that $a$ can activate either $b$, or $c$, or both of them. Moreover, by looking at the dependency graph $\mathcal{G}_2$ depicted in Figure 1, observe that $b$ and $d$ are parallel activities. Thus, the traces that $\mathcal{C}_2$ supports are $abe$, $acde$, $abcde$, $acbde$, and $acdbe$. ◁

We leave the section by pointing out that a useful extension of causal nets consists of allowing input and output bindings to be multi-sets, rather than just sets. With this extended model, for instance, an activity $x$ can activate two different instances of an activity $y$, for which $(x, y)$ is an edge in the underlying dependency graph.[2] A causal net enriched with this capability will be hereinafter called an *extended causal net*.

### 2.2. Dependency Graphs and Process Mining: Basic Results

In process mining, a log $L$ is given and the goal is to derive a process model supporting its traces. We next show that the main task to be carried out to this end is essentially the discovery of the underlying dependency graph. In fact, based on this property, we can contextually show that the whole semantics of causal nets can be recast in simple graph-theoretic terms, which is convenient for our subsequent elaborations.

DEFINITION 2.5. Let $L$ be a log. A dependency graph $\mathcal{G}$ *acyclically supports* $L$, denoted by $\mathcal{G} \vdash_a L$, if for each trace $t \in L$, there is a subgraph $\mathcal{G}_t$ of $\mathcal{G}$ such that:

— $\mathcal{G}_t$ is an acyclic dependency graph over $\mathcal{A}(\{t\})$, and
— $t$ is a *topologic sort* of $\mathcal{G}_t$, i.e., for each edge $(t[i], t[j])$ in $\mathcal{G}_t$, we have that $i < j$. □

EXAMPLE 2.6. Consider again the graph $\mathcal{G}_2$ in Figure 1, and note that $\mathcal{G}_2 \vdash_a \{abe, acde, abcde, acbde, acdbe\}$. For instance, given the trace $abe$, the subgraph of $\mathcal{G}_2$ induced over the activities $\{a, b, e\}$ is an acyclic dependency graph and $abe$ is a topologic sort of it. Moreover, for any log $L$ including a trace not in $\{abe, acde, abcde, acbde, acdbe\}$, we can check that $\mathcal{G}_2 \vdash_a L$ does not hold. ◁

At this point, it is interesting to observe that the set of traces supported by the causal network $\mathcal{C}_2$ (see Example 2.4) precisely coincides with the set of traces acyclically supported by the dependency graph $\mathcal{G}_2$ on top of which $\mathcal{C}_2$ is built. This is not by

[2]Alternatively, we might think that input and output bindings are sets as usual, but that $x$ activates two *hidden activities*, say $h_1$ and $h_2$, which both activate $y$ in their turn. While hidden activities play a role in the enactments, they are not registered in the log. In fact, in process discovery applications, hidden activities are frequently used to enrich the basic expressivity of the process modeling languages.

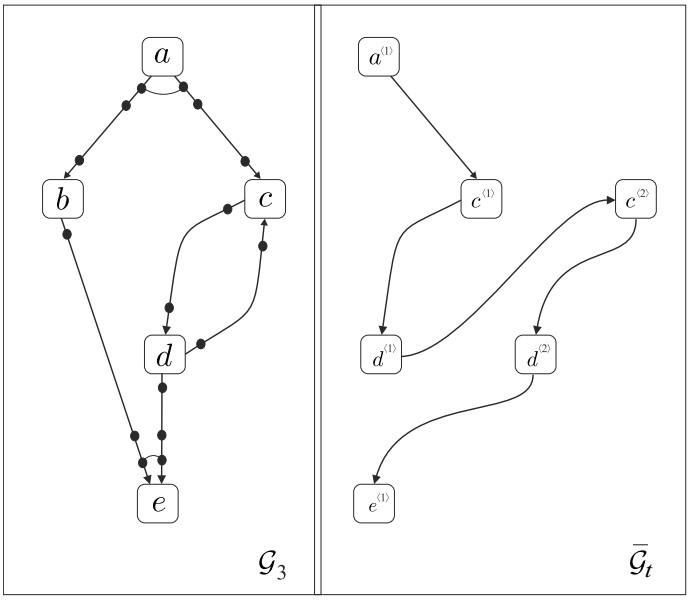



Fig. 2. A process model involving a cycle, with an example unfolding.

chance, and is intimately related with the fact that none of the given traces contains repetitions of the same activity. We next formalize this property.

First, we say that a log $L$ is *linear* if there is no trace in $L$ containing repetitions of the same activity, i.e., for each $t \in L$ and for each $i, j \in \{1, ..., len(t)\}$ with $i \neq j$, $t[i] \neq t[j]$ holds. Then, we derive the following result—proofs in this section are rather technical and are deferred to Appendix A, for the sake of readability.

THEOREM 2.7. *Let $L$ be a linear log, and let $\mathcal{G}$ be a dependency graph. Then, $\mathcal{G} \vdash_a L$ if, and only if, there is a causal net $\mathcal{C} = \langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ such that $\mathcal{C} \vdash L$.*

To deal with arbitrary logs, we next introduce a mechanism to virtually unfold cycles. For each trace $t$, let $unfold(t)$ denote the trace obtained from $t$ by substituting the $i$-th occurrence in $t$ of any activity $a$ with the fresh (virtual) activity $a^{\langle i \rangle}$. Moreover, let $unfold(\mathcal{A})$ denote the (infinite) set of all virtual activities that can be built based on $\mathcal{A}$. The starting and terminating activity in $unfold(\mathcal{A})$ are $a_{\perp}^{\langle 1 \rangle}$ and $a_{\top}^{\langle 1 \rangle}$, respectively.

If $L$ is a log, then we define $unfold(L)$ as the linear log $\{unfold(t) \mid t \in L\}$. Moreover, if $\bar{\mathcal{G}} = (\bar{V}, \bar{E})$ is a graph where $\bar{V} \subseteq unfold(\mathcal{A})$, then we define the *folding* of $\bar{\mathcal{G}}$ as the directed graph $fold(\bar{\mathcal{G}}) = (V, E)$ such that $V = \{x \mid \exists x^{\langle i \rangle} \in unfold(\mathcal{A}) \text{ s.t. } x^{\langle i \rangle} \in \bar{V}\}$ and $E = \{(x, y) \mid \exists x^{\langle i \rangle}, y^{\langle j \rangle} \in unfold(\mathcal{A}) \text{ s.t. } (x^{\langle i \rangle}, y^{\langle j \rangle}) \in \bar{E}\}$.

DEFINITION 2.8. Let $L$ be a log. A dependency graph $\mathcal{G}$ *supports* $L$, denoted by $\mathcal{G} \vdash L$, if for each trace $t \in L$, there is a graph $\bar{\mathcal{G}}_t$ such that $fold(\bar{\mathcal{G}}_t)$ is a subgraph of $\mathcal{G}$, $\bar{\mathcal{G}}_t \vdash_a \{unfold(t)\}$, and the following two conditions hold:

(1) there is no pair of edges $(x^{\langle i \rangle}, y^{\langle j \rangle})$, $(x^{\langle i \rangle}, y^{\langle j' \rangle})$ in $\bar{\mathcal{G}}_t$ such that $j \neq j'$, and
(2) there is no pair of edges $(x^{\langle i \rangle}, y^{\langle j \rangle})$, $(x^{\langle i' \rangle}, y^{\langle j \rangle})$ in $\bar{\mathcal{G}}_t$ such that $i \neq i'$. □

EXAMPLE 2.9. Consider the log consisting of the trace $t = acdcde$, and note that $unfold(acdcde) = a^{\langle 1 \rangle} c^{\langle 1 \rangle} d^{\langle 1 \rangle} c^{\langle 2 \rangle} d^{\langle 2 \rangle} e^{\langle 1 \rangle}$. Moreover, check that the graph $\bar{\mathcal{G}}_t$ depicted on the right part of Figure 2 is such that $\bar{\mathcal{G}}_t \vdash_a \{unfold(acdcde)\}$. In fact, $fold(\bar{\mathcal{G}})$ is a sub-

graph of the dependency graph $\mathcal{G}_3$ depicted in the left part of the figure, and it is easily seen that conditions (1) and (2) of Definition 2.8 hold on $\bar{\mathcal{G}}_t$. Hence, $\mathcal{G}_3 \vdash \{acdcde\}$ holds.

Consider now the trace $t' = abbe$ and $unfold(abbe) = a^{\langle 1\rangle}b^{\langle 1\rangle}b^{\langle 2\rangle}e^{\langle 1\rangle}$. Let $\bar{\mathcal{G}}_{t'}$ be the graph consisting of the edges $(a^{\langle 1\rangle}, b^{\langle 1\rangle})$, $(a^{\langle 1\rangle}, b^{\langle 2\rangle})$, $(b^{\langle 1\rangle}, e^{\langle 1\rangle})$, and $(b^{\langle 2\rangle}, e^{\langle 1\rangle})$. Then, $\bar{\mathcal{G}}_{t'} \vdash_a \{unfold(abbe)\}$, and $fold(\bar{\mathcal{G}}_{t'})$ is a subgraph of $\mathcal{G}_3$. However, $\bar{\mathcal{G}}_{t'}$ violates conditions (1) and (2) in Definition 2.8. Therefore, $\mathcal{G}_3$ does not support $\{abbe\}$. ◁

Note that whenever the log $L$ is linear, the above definition reduces to Definition 2.5, and in particular conditions (1) and (2) are immaterial. More formally, in this case, $\mathcal{G} \vdash L$ holds if, and only if, $\mathcal{G} \vdash_a L$ holds. Hence, for linear logs, we are in the position of applying Theorem 2.7 with '$\vdash$' in place of '$\vdash_a$'. More generally, the following result is established in order to relate the notion of support over dependency graphs with the notion of support over causal nets. The result is of interest in its own, as it allows to restate the semantics of causal nets in pure graph-theoretic terms. In fact, it will play an important role in role in our subsequent elaborations.

THEOREM 2.10. *Let $L$ be a log, and let $\mathcal{G}$ be a dependency graph. Then, $\mathcal{G} \vdash L$ if, and only if, there is a causal net $\mathcal{C} = \langle \mathcal{G}, \mathcal{I}, \mathcal{O}\rangle$ such that $\mathcal{C} \vdash L$.*

EXAMPLE 2.11. Consider the causal net $\mathcal{C}_3 = \langle \mathcal{G}_3, \mathcal{I}_3, \mathcal{O}_3\rangle$, where $\mathcal{G}_3$ is the dependency graph shown in Figure 2, and where $\mathcal{I}_3$ and $\mathcal{O}_3$ are the functions such that: $\mathcal{I}_3(a) = \{\emptyset\}$, $\mathcal{I}_3(b) = \{\{(a,b)\}\}$, $\mathcal{I}_3(c) = \{\{(b,c)\}, \{d,c\}\}$, $\mathcal{I}_3(d) = \{\{(c,d)\}\}$, $\mathcal{I}_3(e) = \{\{(b,e)\}, \{(c,e)\}, \{(b,e),(c,e)\}\}$, $\mathcal{O}_3(a) = \{\{(a,b),(a,c)\}, \{(a,b\}, \{(a,c)\}\}$, $\mathcal{O}_3(b) = \{\{(b,d)\}\}$, $\mathcal{O}_3(c) = \{\{(c,d)\}\}$, $\mathcal{O}_3(d) = \{\{(d,e)\}\}$, and $\mathcal{O}_3(e) = \{\emptyset\}$. Note that $\mathcal{C}_3 \vdash \{acdcde\}$ holds.

Hence, by the above result, we can conclude that $\mathcal{G}_3 \vdash \{acdcde\}$ also holds, as we have in fact already observed in Example 2.9. ◁

We leave the section, by noticing that if conditions (1) and (2) in Definition 2.8 are not guaranteed to hold, then a weaker variant of Theorem 2.10 can be still established.

THEOREM 2.12. *Let $L$ be a log, and let $\mathcal{G}$ be a dependency graph such that for each trace $t \in L$, there is a graph $\bar{\mathcal{G}}_t$ such that $fold(\bar{\mathcal{G}}_t)$ is a subgraph of $\mathcal{G}$ and $\bar{\mathcal{G}}_t \vdash_a \{unfold(t)\}$. Then, there is a possibly extended causal net $\mathcal{C} = \langle \mathcal{G}, \mathcal{I}, \mathcal{O}\rangle$ such that $\mathcal{C} \vdash L$.*

## 3. PRECEDENCE CONSTRAINTS: FORMAL FRAMEWORK AND COMPLEXITY

In this section, we propose and analyze a framework to specify additional properties on the process models that can be produced as output by process discovery algorithms. In particular, we formalize the concept of precedence constraints, and discuss its application to the problem of mining causal nets.

### 3.1. Syntax and Semantics

A *precedence constraint* is an assertion aimed at expressing a relationship of precedence among some of the activities in the underlying set $\mathcal{A}$. The language of precedence constraints is next defined in order to smoothly allow the formalization of the prior knowledge that is usually available to the analyst, such as, parallelism, locality, or exclusivity of activities (cf. [Goedertier et al. 2009]).

DEFINITION 3.1. A *positive precedence* constraint $\pi$ over $\mathcal{A}$ is either

— an expression of the form $S \to T$, called *edge* constraint, or
— an expression of the form $S \rightsquigarrow T$, called *path* constraint,

where $S, T \subseteq \mathcal{A}$, with $|S| \geq 1$ and $|T| \geq 1$, are non-empty sets of activities.

For a positive constraint $\pi$, $\neg(\pi)$ is a *negative* precedence constraint. □

Precedence constraints are interpreted over directed graphs as follows.

DEFINITION 3.2. Let $\mathcal{G} = (V, E)$ be a directed graph such that $V \subseteq \mathcal{A}$. Then,

(1) $\mathcal{G}$ satisfies an edge constraint $S \to T$, if there is an edge $(x, y) \in E$ with $x \in S$ and $y \in T$;
(2) $\mathcal{G}$ satisfies a path constraint $S \rightsquigarrow T$, if there is a sequence $x = a_0, a_1, ..., a_n = y$, with $n > 0$, such that $x \in S$, $y \in T$ and $(a_i, a_{i+1}) \in E$, for each $i \in \{0, ..., n-1\}$;
(3) $\mathcal{G}$ satisfies $\neg(\pi)$, if $\mathcal{G}$ does not satisfy $\pi$.

If $\mathcal{G}$ satisfies each constraint in a set $\Pi$ of precedence constraints, then $\mathcal{G}$ is a *model* of $\Pi$, denoted by $\mathcal{G} \models \Pi$. The set of all activities occurring in the constraints in $\Pi$ is hereinafter denoted by $\mathcal{A}(\Pi)$. □

As we have already informally discussed, a foundational task in process mining consists of automatically building a process model that can explain the behavior registered in all the traces of some log $L$ given as input. In this context, precedence constraints can be naturally exploited to formalize additional requirements that the model discovered from $L$ has to satisfy. This gives rise to the following problem

CN-MINING: Given a set $\mathcal{A}$ of activities, a log $L$ with $\mathcal{A}(L) \subseteq \mathcal{A}$, and a set $\Pi$ of precedence constraints with $\mathcal{A}(\Pi) \subseteq \mathcal{A}$, compute a possibly extended causal net $\mathcal{C} = \langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ over $\mathcal{A}$ such that $\mathcal{C} \vdash L$ and $\mathcal{G} \models \Pi$, or check that no net with these properties exists.

Note that, for $\Pi = \emptyset$, the problems above reduce to the standard ones considered in the literature. Moreover, observe that we are not considering for the moment quality measures on the net to be computed. This issue will be explored in the following. Finally, note that extended causal nets are allowed as solutions to the problems. In fact, throughout the paper, we shall explicitly discuss and show how our results extend to the more stringent setting where the focus is on (standard) causal nets. Here, we stress that on linear logs the two settings coincide, as the ability of extended causal nets to activate multiple instances of the same activity is useless in this case.

FACT 3.3. *Over linear logs,* CN-MINING *admits a solution if, and only if, it admits a causal net as a solution.*

EXAMPLE 3.4. Consider the set $\Pi = \{ \neg(\{b\} \rightsquigarrow \{d\}), \neg(\{d\} \rightsquigarrow \{b\})\}$ of precedence constraints. We have two negative path constraints, stating that $b$ and $d$ must be executed over "parallel" branches of the given process.

Consider then the traces $abcde$ and $acbde$ within the setting of Example 1.1, plus the causal nets $\mathcal{C}_1$ and $\mathcal{C}_2$ discussed in Example 2.2 and depicted in Figure 1. Without additional constraints, we have already noticed that $\mathcal{C}_1$ and $\mathcal{C}_2$ are such that $\mathcal{C}_1 \vdash \{abcde, acbde\}$ and $\mathcal{C}_2 \vdash \{abcde, acbde\}$. However, the dependency graph $\mathcal{G}_2$ (associated with $\mathcal{C}_2$) is a model of $\Pi$, while $\mathcal{G}_1$ is not as it violates the constraint $\neg(\{b\} \rightsquigarrow \{d\})$. Thus, $\mathcal{C}_2$ is a solution to CN-MINING on input $\{abcde, acbde\}$ and $\Pi$.

Finally, consider the causal net $\mathcal{C}_3$ discussed in Example 2.11 and illustrated in Figure 2. Then, $\mathcal{C}_3$ is a solution to CN-MINING on input $\{abcde, acbde\}$ and $\Pi$. Note that in this case $\mathcal{G}_3$ is not acyclic. ◁

As a further remark, note that a solution to the mining problem might require activities that do not occur in the log and in the constraints. For instance, consider the set $\{\neg(\{a\} \to \{b\}), \{a\} \rightsquigarrow \{b\}\}$ of constraints prescribing the existence of a path from $a$ to $b$, but forbidding the existence of a direct connection. If the log provided as input to the mining problem (together with the above constraints) is defined over these two activities only, then any solution has to be clearly defined over (at least) one "fresh" activity, say $v$, in order to include an edge from $a$ to $v$ plus an edge from $v$ to $b$.

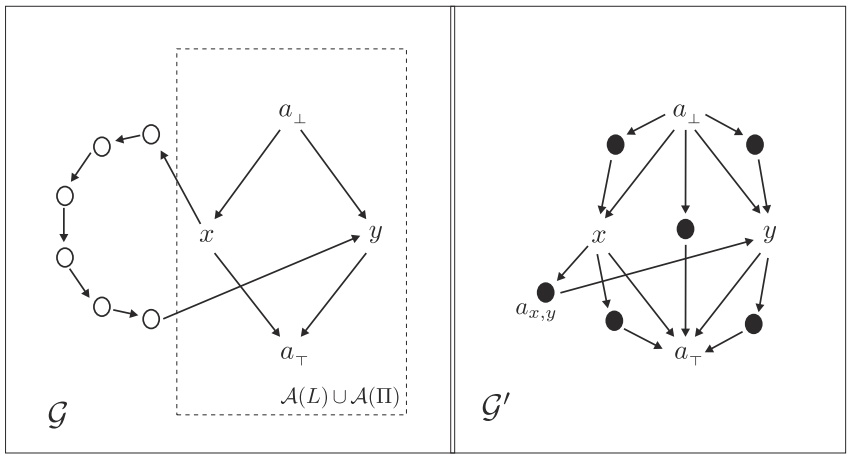



Fig. 3. Example construction in the proof of Theorem 3.5.

According to the formulation of CN-MINING, any "fresh" activity has to be explicitly provided as input to the mining problem, too. Therefore, one might wonder whether this is restrictive and, in particular, whether scenarios exist where all solutions to such problems need exponentially many activities (w.r.t. those occurring in $\mathcal{A}(L)$ and $\mathcal{A}(\Pi)$), so that explicitly listing all of them would artificially blow-up the size of the input. The following result shows that these scenarios are not possible, since a "small" solution always exists if the problem admits any solution at all. Hence, there is no loss of generality by assuming that the set $\mathcal{A}$ is given as input (possibly implicitly, i.e., by just specifying the upper bound on the number of activities which is defined in the statement of the result). To the contrary, in our formulation, we gain flexibility as we can, for instance, specify that we are interested in solutions defined over the symbols occurring in the log and in the constraints only (so that the solution requiring the fresh activity $v$ would be not admissible).

THEOREM 3.5. *Let $L$ be a log, let $\Pi$ be a set of precedence constraints, and let $\mathcal{C} = \langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ be a causal net (resp., an extended causal net) with $\mathcal{C} \vdash L$ and $\mathcal{G} \models \Pi$. Then, a causal net (resp., an extended causal net) $\mathcal{C}' = \langle \mathcal{G}', \mathcal{I}', \mathcal{O}' \rangle$ exists such that $\mathcal{C}' \vdash L$, $\mathcal{G}' \models \Pi$, and $|V'| \leq |\mathcal{A}(L) \cup \mathcal{A}(\Pi)|^2 + |\mathcal{A}(L) \cup \mathcal{A}(\Pi)|$, with $V'$ being the set of nodes of $\mathcal{G}'$. Moreover, if $\mathcal{G}$ is acyclic, then $\mathcal{G}'$ is acyclic, too.*

PROOF. Let $\mathcal{C} = \langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$, with $\mathcal{G} = (V, E)$, be a (resp., an extended) causal net such that $\mathcal{C} \vdash L$ and $\mathcal{G} \models \Pi$. Consider the graph $\mathcal{G}' = (V', E'_1 \cup E'_2)$ built as follows. The set $V'$ consists of all the activities in $\mathcal{A}(L) \cup \mathcal{A}(\Pi)$ plus a fresh activity $a_{x,y}$ for each pair of activities $x, y \in \mathcal{A}(L) \cup \mathcal{A}(\Pi)$ such that there is a path in $\mathcal{G}$ from $x$ to $y$. The set $E'_1$ consists of all the edges in $E$ defined over the activities in $\mathcal{A}(L) \cup \mathcal{A}(\Pi)$, i.e., $E'_1 = \{(x, y) \in E \mid \{x, y\} \subseteq \mathcal{A}(L) \cup \mathcal{A}(\Pi)\}$. The set $E'_2$ contains the edges $(x, a_{x,y})$ and $(a_{x,y}, y)$, for each activity $a_{x,y} \in V' \setminus V$, and no further edge is in $E'_2$.

As an example, consider the graph $\mathcal{G}$ reported on the left of Figure 3. The graph is defined over the activities in $\mathcal{A}(L) \cup \mathcal{A}(\Pi) = \{a_\perp, a_\top, x, y\}$ plus 6 additional activities, which are depicted as circles. The graph $\mathcal{G}'$ that is built based on $\mathcal{G}$ is illustrated on the right part of the same figure. All nodes in $\mathcal{G}'$ that do not occur in $\mathcal{G}$ are depicted as black circles. In particular, observe that the node $a_{x,y}$ is responsible of preserving the connectivity that is supported in $\mathcal{G}$ by the nodes not occurring in $\mathcal{A}(L) \cup \mathcal{A}(\Pi)$.

Let us now analyze the properties of $\mathcal{G}'$. First, note that the activities $a_\perp$ and $a_\top$ are in $V'$, as in fact they occur in $\mathcal{A}(L)$. Moreover, since $\mathcal{G}$ is a dependency graph, no edge ingoing into $a_\perp$ (resp., outgoing from $a_\top$) occurs in $E'_2$. Hence, given the construction of the edges in $E'_1$, we conclude that $a_\perp$ and $a_\top$ have no ingoing and outgoing edges in $\mathcal{G}'$.

Now, we claim that for each pair of activities $x, y \in \mathcal{A}(L) \cup \mathcal{A}(\Pi)$, there is a path from $x$ to $y$ in $\mathcal{G}$ if, and only if, there is a path from $x$ to $y$ in $\mathcal{G}'$. Indeed, if there is a path from $x$ to $y$ in $\mathcal{G}$, then the edges $(x, a_{x,y})$ and $(a_{x,y}, y)$ occur in $E'_2$. Conversely, assume that there is a path $\pi$ from $x$ to $y$ in $\mathcal{G}'$, and for the sake of contradiction that there is no path from $x$ to $y$ in $\mathcal{G}$. As all edges of $E$ defined over the activities in $\mathcal{A}(L) \cup \mathcal{A}(\Pi)$ are in $E'_1$, it must be the case that two edges occur in $\pi$ having the form $(\bar{x}, a_{\bar{x},\bar{y}})$ and $(a_{\bar{x},\bar{y}}, \bar{y})$ and such that there is no path from $\bar{x}$ to $\bar{y}$ in $\mathcal{G}$. However, this is impossible given the construction of the edges in $E'_2$.

In the light of the above property, it follows that if $\mathcal{G}$ is acyclic, then $\mathcal{G}'$ is acyclic, too. Indeed, just notice that any cycle in $\mathcal{G}'$ must necessarily include a node in $\mathcal{A}(L) \cup \mathcal{A}(\Pi)$. Moreover, we can conclude that each activity $a \in \mathcal{A}(L) \cup \mathcal{A}(\Pi) \setminus \{a_\perp, a_\top\}$ is in a path in $\mathcal{G}'$ from $a_\perp$ to $a_\top$. Consider then an activity of the form $a_{x,y}$, which occurs in $V' \setminus V$. Since $x$ (resp., $y$) either coincides with $a_\perp$ (resp., $a_\top$) or is reachable from $a_\perp$ (can reach $a_\top$) in $\mathcal{G}'$, because this property hold in fact on $\mathcal{G}$, we also conclude that $a_{x,y}$ is in a path in $\mathcal{G}'$ from $a_\perp$ to $a_\top$. By putting the above observations together, it follows that $\mathcal{G}'$ is a dependency graph over the set $V'$ of activities. Moreover, $|V'| \leq |\mathcal{A}(L) \cup \mathcal{A}(\Pi)|^2 + |\mathcal{A}(L) \cup \mathcal{A}(\Pi)|$.

Recall now that $\mathcal{G}'$ preserves all the edges defined over the activities in $\mathcal{A}(L)$, and that $\mathcal{C} = \langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ is a causal net (resp., extended causal net) such that $\mathcal{C} \vdash L$. It follows that there is a (resp., an extended) causal net $\mathcal{C}' = \langle \mathcal{G}', \mathcal{I}', \mathcal{O}' \rangle$ such that $\mathcal{C}' \vdash L$, where $\mathcal{I}'$ and $\mathcal{O}'$ just extend $\mathcal{I}$ and $\mathcal{O}$ as to include, for each given activity, a binding defined over the whole set of its ingoing and outgoing edges, respectively. Note that $\mathcal{C}'$ supports all the traces in $L$, even without using such fresh bindings. In fact, the construction of $\mathcal{I}'$ and $\mathcal{O}'$ is just required to ensure that $\mathcal{C}'$ is formally a (possibly extended) causal net.

In order to conclude, we have then to show that $\mathcal{G}' \models \Pi$. To this end, note that edge constraints and negated edge constraints are satisfied by $\mathcal{G}'$, because they are satisfied by $\mathcal{G}$ and since the two graphs coincide over the activities in $\mathcal{A}(\Pi)$. Eventually, recall that, for each pair of activities $x, y \in \mathcal{A}(L) \cup \mathcal{A}(\Pi)$, there is a path from $x$ to $y$ in $\mathcal{G}$ if, and only if, there is a path from $x$ to $y$ in $\mathcal{G}'$. Hence, also path constraints and negated path constraints are satisfied by $\mathcal{G}'$, because they are satisfied by $\mathcal{G}$. That is, $\mathcal{G}' \models \Pi$. □

### 3.2. Process Mining and Precedence Constraints

From a conceptual viewpoint, the problems defined above comprise a mining task, i.e., mining a process model supporting a given log, and a reasoning task, i.e., to check whether the model additionally satisfies some precedence constraints. In fact, we next show that even the learning task can be declaratively formulated in terms of reasoning about precedence constraints. This property will be crucial to study the intrinsic complexity of the framework and to design efficient solution algorithms (by also allowing to greatly simplify their analysis). The basic idea is to characterize the notions of support in Definition 2.5 and Definition 2.8 in terms of precedence constraints.

DEFINITION 3.6. Let $L$ be a log. For each trace $t \in L$, the set of precedence constraints *induced by* $t$ is defined as follows:

$$\begin{aligned}\pi(t) = \ & \{\ \{t[1], ..., t[i-1]\} \rightarrow \{t[i]\} \mid 1 < i \leq len(t)\ \} \cup \\ & \{\ \{t[i]\} \rightarrow \{t[i+1], ..., t[len(t)]\} \mid 1 \leq i < len(t)\ \}.\end{aligned}$$

The set of precedence constraints induced by $L$ is defined as $\pi(L) = \bigcup_{t \in L} \pi(t)$. □

Intuitively, we just state that each activity in the trace $t$ can be directly reached by at least one of its predecessors in $t$, and it can directly reach at least one of its successors

in $t$. This suffices to precisely characterize the semantics of causal nets over logs that are linear, as illustrated below.

THEOREM 3.7. *Let $L$ be a linear log and let $\mathcal{G}$ be a dependency graph. Then, $\mathcal{G} \models \pi(L)$ if, and only if, $\mathcal{G} \vdash_a L$.*

PROOF. *(if part).* Assume that $\mathcal{G} \vdash_a L$, i.e., for each $t \in L$, there is a subgraph $\mathcal{G}_t = (V_t, E_t)$ of $\mathcal{G} = (V, E)$ such that $\mathcal{G}_t$ is an acyclic dependency graph, and $t$ is a topologic sort of $\mathcal{G}_t$. Therefore, for each $i \in \{2, ..., len(t)\}$ (resp., $i \in \{1, ..., len(t) - 1\}$), there is a path from $t[1]$ (resp., $t[i]$) to $t[i]$ (resp., $t[len(t)]$) in $\mathcal{G}_t$, by definition of dependency graph. In particular, since $t$ is a topologic sort of $\mathcal{G}_t$, we are guaranteed about the existence of an edge in $E_t$ (and then in $E$) having the form $(t[j], t[i])$ (resp., $(t[i], t[j'])$) and such that $j < i$ (resp., $i < j'$) holds. Hence, the set $\pi(t)$ of the precedence constraints induced by $t$ are satisfied by $\mathcal{G}$, for each trace $t$ in $L$. That is, $\mathcal{G} \models \pi(L)$.

*(only-if part).* Assume that $\mathcal{G} \models \pi(L)$, with $\mathcal{G} = (V, E)$. Let $t$ be a trace in $L$, and let $\mathcal{G}_t = (V_t, E_t)$ be the graph such that $V_t = \{t[1], ..., t[len(t)]\}$ and $E_t = \{(t[i], t[j]) \in E \mid 1 \leq i < j \leq len(t)\}$. Since $L$ is linear, we can note that $\mathcal{G}_t$ is acyclic, that $t[1] = a_\perp$ has no ingoing edges, and that $t[len(t)] = a_\top$ has no outgoing edges. We now claim that each activity $t[i] \in V_t \setminus \{t[1]\}$ can be reached from $t[1]$. The above property can be shown by induction on the index $i > 1$. In the case where $i = 2$, $(t[1], t[2])$ must belong to $E$ (and hence to $E_t$) in order to satisfy the constraint $\{t[1]\} \rightarrow \{t[2]\}$ in $\pi(t)$. Assume now that the activities in the set $\{t[2], ..., t[i-1]\}$ can be reached from $t[1]$. Then, because of the constraint $\{t[1], ..., t[i-1]\} \rightarrow \{t[i]\}$ in $\pi(t)$, we again have that $t[i]$ can be reached from $t[1]$. Similarly, it can be checked that the terminating activity $t[len(t)]$ can be reached by each activity $t[i] \in V_t \setminus \{t[len(t)]\}$, by using this time the fact that $\{t[i]\} \rightarrow \{t[i+1], ..., t[len(t)]\}$ is in $\pi(t)$. Hence, $\mathcal{G}_t$ is an acyclic dependency graph. Moreover, for each edge $(t[i], t[j])$ in $E_t$, we have that $i < j$ holds by construction. Thus, $t$ is a topologic sort of $\mathcal{G}_t$. As $\mathcal{G}_t$ is a subgraph of $\mathcal{G}$, we then have $\mathcal{G} \vdash_a L$. □

Theorem 3.7 and Theorem 2.7 imply the following corollary, where process mining over linear logs is restated in terms of reasoning about precedence constraints.

COROLLARY 3.8. *Let $\mathcal{G}$ be a dependency graph over a set $\mathcal{A}$ of activities, let $L$ be a linear log with $\mathcal{A}(L) \subseteq \mathcal{A}$, and let $\Pi$ be a set of precedence constraints with $\mathcal{A}(\Pi) \subseteq \mathcal{A}$. Then, the following statements are equivalent:*

(1) *The graph $\mathcal{G}$ is a model of $\Pi \cup \pi(L)$.*
(2) *There is a causal net $\mathcal{C} = \langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ that is a solution to* CN-MINING *on input $\mathcal{A}$, $L$, and $\Pi$.*

Note that, because of Fact 3.3, the above result completely characterizes the cases where the problem CN-MINING admit solutions. Indeed, point (2) above can be equivalently restated as the existence of a possibly extended causal net that is a solution to the mining problem.

EXAMPLE 3.9. Let $\Pi$ be the set of constraints in Example 3.4, and consider the novel set $\Pi' = \Pi \cup \pi(\{abcde, acbde\})$. It can be checked that the dependency graph $\mathcal{G}_2$ in Figure 2 is a model of $\Pi'$. Thus, by Corollary 3.8, we are guaranteed about the existence of a causal net that can be defined on top of $\mathcal{G}_2$ and that is a solution to CN-MINING on input $\{abcde, acbde\}$ and $\Pi$. In fact, we already know that the causal net $\mathcal{C}_2$ defined in Example 2.2 is a solution. ◁

Moreover, it is useful to remark that the '(1)⇒(2)'-part of Corollary 3.8 can be stated constructively. That is, the causal net $\langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ can be efficiently built given the graph $\mathcal{G}$. The method is just based on inspecting the proofs for the results in Section 2.2,

which are reported in Appendix A. In fact, we anticipate that the method will be formalized algorithmically and analyzed in Section 5.

At this point, the natural question is whether we can extended Theorem 3.7 to deal with arbitrary logs. An answer that is however only partially positive is stated below.

THEOREM 3.10. *Let $L$ be a log and let $\mathcal{G}$ be a dependency graph. Then, $\mathcal{G} \vdash L$ implies that $\mathcal{G} \models \pi(L)$.*

PROOF. Assume that $\mathcal{G} \vdash L$ holds, with $\mathcal{G} = (V, E)$. By Definition 2.8, for each trace $t \in L$, there is a graph $\bar{\mathcal{G}}_t = (\bar{V}_t, \bar{E}_t)$ such that, in particular, $fold(\bar{\mathcal{G}}_t)$ is a subgraph of $\mathcal{G} = (V_t, E_t)$ and $\bar{\mathcal{G}}_t \vdash_a \{unfold(t)\}$. Then, we apply Theorem 3.7 on $\bar{\mathcal{G}}_t$, and we conclude that $\bar{\mathcal{G}}_t \models \pi(unfold(t))$. Moreover, we note that $fold(\bar{\mathcal{G}}_t) \models \pi(t)$ also holds. Indeed, for each $i \in \{1, ..., len(t)\}$, if $(unfold(t)[j], unfold(t)[i])$ (resp., $(unfold(t)[i], unfold(t)[h])$) is in $\bar{E}_t$, then $(t[j], t[i])$ (resp., $(t[i], t[h])$) is in $E_t$ with $j < i$ (resp., $h > i$), because $fold(\bar{\mathcal{G}}_t)$ is a subgraph of $\mathcal{G}_t$. Finally, define $\bar{\mathcal{G}} = (\bigcup_{t \in L} \bar{V}_t, \bigcup_{t \in L} \bar{E}_t)$. Then, we claim that $fold(\bar{\mathcal{G}}) \models \pi(L)$. Indeed, the constraints induced by the traces in $L$ are positive ones, so that since $fold(\bar{\mathcal{G}}_t) \models \pi(t)$, the graph $fold(\bar{\mathcal{G}})$ (of which $fold(\bar{\mathcal{G}}_t)$ is a subgraph) is still such that $fold(\bar{\mathcal{G}}) \models \pi(t)$. In order to conclude, we can eventually observe that $fold(\bar{\mathcal{G}})$ is a subgraph of $\mathcal{G}$, and hence $\mathcal{G} \models \pi(L)$. □

This is the best one can hope to do. Indeed, we can see that $\mathcal{G} \models \pi(L)$ does not imply $\mathcal{G} \vdash L$, by just looking again at Example 2.9. There, we have noticed that the graph $\mathcal{G}_3$ does not support $\{abbe\}$. However, it can be checked that $\mathcal{G}_3 \models \pi(\{abbe\})$ holds.

Despite the above limitation, the counterpart of Corollary 3.8 for arbitrary logs can still be obtained by moving to possibly extended causal nets—as with Corollary 3.8, the '(1)⇒(2)'-part below will be formalized algorithmically and analyzed in Section 5.

THEOREM 3.11. *Let $\mathcal{G}$ be a dependency graph over a set $\mathcal{A}$ of activities, let $L$ be a log with $\mathcal{A}(L) \subseteq \mathcal{A}$, and let $\Pi$ be a set of precedence constraints with $\mathcal{A}(\Pi) \subseteq \mathcal{A}$. Then, the following statements are equivalent:*

(1) *The graph $\mathcal{G}$ is a model of $\Pi \cup \pi(L)$.*
(2) *There is a possibly extended causal net $\mathcal{C} = \langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ that is a solution to* CN-MINING *on input $\mathcal{A}$, $L$, and $\Pi$.*

PROOF. (1)⇒(2). Assume that $\mathcal{G}$ is a model of $\Pi \cup \pi(L)$, with $\mathcal{G} = (V, E)$. Let $\bar{\mathcal{G}} = (\bar{V}, \bar{E})$ be the graph such that $\bar{V} = \mathcal{A}(unfold(L))$ and $\bar{E} = \{(x^{\langle i \rangle}, y^{\langle j \rangle}) \mid (x, y) \in E, x^{\langle i \rangle} \in \bar{V}, y^{\langle j \rangle} \in \bar{V}\}$. Since $\bar{\mathcal{G}}$ is a dependency graph, it is also the case that $\bar{\mathcal{G}}$ is a dependency graph (over $\mathcal{A}(unfold(L))$). Moreover, $\bar{\mathcal{G}} \models \pi(unfold(L))$ holds. Indeed, just note that for each trace $t \in L$ and for each $i \in \{1, ..., len(t)\}$, if $(t[j], t[i])$ (resp., $(t[i], t[h])$) is in $E$ with $j < i$ (resp., $h > i$), then $(unfold(t)[j], unfold(t)[i])$ (resp., $(unfold(t)[i], unfold(t)[h])$) is in $\bar{E}$ by construction. Thus, we can apply Theorem 3.7 in order to conclude that $\bar{\mathcal{G}} \vdash_a unfold(L)$. Eventually, we observe that $fold(\bar{\mathcal{G}})$ is clearly a subgraph of $\mathcal{G}$. Then, we distinguish two cases. In the case where $\mathcal{G}$ satisfies conditions (1) and (2) in Definition 2.8, then we can apply Theorem 2.10 and conclude that a causal net $\mathcal{C}$ can be built over $\mathcal{G}$ such that $\mathcal{C} \vdash L$. Instead, in the case where one of the above conditions does not hold, we can apply Theorem 2.12 and conclude that there is a possibly extended causal net $\mathcal{C}$ built over $\mathcal{G}$ and such that $\mathcal{C} \vdash L$.

(2)⇒(1). Assume that $\mathcal{C} = \langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ is a solution to CN-MINING on input $\mathcal{A}$, $L$, and $\Pi$, and for the sake of contradiction that $\mathcal{G}$ is not a model of $\Pi \cup \pi(L)$. The fact that $\mathcal{G}$ is not a model of $\Pi$ is trivially impossible. Thus, assume that $\mathcal{G}$ is not a model of $\pi(L)$, i.e., there is a trace $t \in L$ such that $\mathcal{G}$ does not satisfy the constraints in $\pi(t)$. According to Definition 3.6, there are two possible cases: (a) there is an index $i \in \{2, ..., len(t)\}$ such that there is no edge in $\mathcal{G}$ from one of the activities in $\{t[1], ..., t[i-1]\}$ to $t[i]$; (b) there

is an index $i \in \{1, ..., len(t) - 1\}$ such that there is no edge in $\mathcal{G}$ from $t[i]$ to one of the activities in $\{t[i + 1], ..., t[len(t)]\}$.

Now, as $\mathcal{C}$ is a solution, it must be the case that $\mathcal{C}$ supports the trace $t$, i.e., there is a binding sequence $\sigma$ that is valid w.r.t. $\mathcal{C}$ and where the $j$-th binding activity $\langle a_j, ib_j, ob_j\rangle$ of $\sigma$, for each $j \in \{1, ..., len(t)\}$, is such that $a_j = t[j]$. In particular, the $i$-th activity is $\langle t[i], ib_i, ob_i\rangle$. In the case (a) above, the set $ib_i$, which is a non-empty set of edges ingoing into $t[i]$, is a subset of the set $\{(x, t[i]) \mid x \notin \{t[1], ..., t[i - 1]\}\}$, while the state $S^{\sigma}_{i-1}$ of the causal net consists of edges having the form $(x, y)$, with $x \in \{t[1], ..., t[i - 1]\}$. It follows that $ib_i$ is not contained in $S^{\sigma}_{i-1}$, and $\sigma$ is not valid, which is impossible as $\sigma$ is valid. In the case (b), the set $ob_i$, which is a non-empty set of edges outgoing from $t[i]$, is included in the set $\{(t[i], y) \mid y \notin \{t[1 + 1], ..., t[len(t)]\}\}$. Thus, $S^{\sigma}_i$ necessarily includes an element having the form $(t[i], y)$, with $y \notin \{t[1 + 1], ..., t[len(t)]\}$. However, as there is no edge in $\mathcal{G}$ from $t[i]$ to one of the activities in $\{t[i + 1], ..., t[len(t)]\}$, this element will eventually occur at the last step of the execution of $\sigma$, i.e., $S^{\sigma}_{len(t)} \neq \emptyset$, which again is impossible as $\sigma$ is valid. □

For instance, since $\mathcal{G}_3 \models \pi(\{abbe\})$ holds, we are guaranteed that a net supporting this trace can be built on top of $\mathcal{G}_3$. To be concrete, we can consider the extended causal net $\mathcal{C}'_3 = \langle \mathcal{G}_3, \mathcal{I}'_3, \mathcal{O}'_3\rangle$ where $\mathcal{I}'_3(a) = \{\emptyset\}$, $\mathcal{I}'_3(b) = \{\{(a, b)\}\}$, $\mathcal{I}'_3(e) = \{\{(b, e), (b, e)\}\}$, $\mathcal{O}'_3(a) = \{\{(a, b), (a, b)\}\}$, $\mathcal{O}'_3(b) = \{\{(b, e)\}\}$, and $\mathcal{O}'_3(e) = \{\emptyset\}$—the specification of the bindings over the activities $c$ and $d$ is irrelevant here.

### 3.3. Significance of the Results and Possible Applications

At the beginning, we have anticipated that the results derived in this section will be crucial to design our algorithms and to conduct the complexity analysis. Indeed, we can now see that Theorem 3.11 reformulates the whole mining problem and the "dynamics" of the causal nets in purely "static" terms, i.e., in terms of reasoning about the satisfaction of precedence constraints. In particular, we can completely get rid of the concepts of bindings and of support of a trace and a log.

In addition to the above described technical (and conceptual) role, the results in this section have also an immediate concrete application. Indeed, given that the whole mining problem has been reformulated in terms of a reasoning problem, a common environment emerged where the tasks of mining a process model supporting a given log and checking whether the model additionally satisfies some precedence constraints are combined synergically and can be simultaneously carried out. In particular, it is immediate to encode the reasoning problem in terms of a "standard" *constraints satisfaction problem*, CSP for short (see, e.g., [Dechter 1992]), and to reuse existing constraint programming platforms to compute models for it, in the spirit of the works by De Raedt et al. [2008] and Nijssen et al. [2009]. This approach has been originally discussed in the conference version of this paper [Greco et al. 2012], and details can be found there.

An advantage of the CSP-based framework is that it can transparently handle arbitrary sets of precedence constraints. The price to be paid however is that computing a solution in some of these settings is very challenging from a computational viewpoint, as we shall formally illustrate in Section 4. In this extended version, we decided therefore to depart from the original approach[3] by proposing solution algorithms that are specific for some classes of constraints (over which solutions can be efficiently computed) as well as methods that handle the general setting heuristically. In fact, from our experimentation, the heuristics methods emerged to be definitively much faster than the CSP-based method, while being capable to end up with an exact solution in

[3]In any case, note that the approach in [Greco et al. 2012] was not designed to deal with causal nets (i.e., with full process models), but its focus was on the discovery of the underlying dependence graphs only.

most cases. Accordingly, for those cases where a heuristic solution (i.e., where some constraints might be violated) is acceptable, our earlier approach has to be considered as pragmatically superseded by the methods proposed here. However, it constitutes an interesting avenue of further research to define different kinds of encodings (possibly, still CSP-based ones) and exact solution approaches for them that have acceptable scalings over real logs and with arbitrary classes of precedence constraints.

## 4. COMPLEXITY ANALYSIS

We now turn to study the computational complexity of the problem CN-MINING, which is an important step towards developing effective algorithms for its solution. We start by recalling some notions of complexity theory, by referring the reader to the book by Garey and Johnson [1979] for more on this subject.

### 4.1. Preliminaries on Complexity Theory

Decision problems are maps from strings (encoding the input instance over a fixed alphabet, e.g., the binary alphabet $\{0, 1\}$) to the set $\{\text{“}yes\text{”}, \text{“}no\text{”}\}$. We are often interested in computations carried out by *non-deterministic* Turing machines. We recall that these are Turing machines that, at some points of the computation, may not have one single next action to perform, but a *choice* between several possible next actions. A non-deterministic Turing machine answers a decision problem if, on any input $x$, (*i*) there is at least one sequence of choices leading to halt in an accepting state if $x$ is a “yes” instance (such a sequence is called accepting computation path); and (*ii*) all possible sequences of choices lead to a rejecting state if $x$ is a “no” instance. The class of decision problems that can be solved by non-deterministic Turing machines in polynomial time is denoted by NP.

A decision problem $A_1$ is *polynomially reducible* to a decision problem $A_2$, if there is a polynomial-time computable function $h$ (called reduction) such that, for every $x$, $h(x)$ is defined and $x$ is a “yes” instance of $A_1$ if and only if $h(x)$ is a “yes” instance of $A_2$. A decision problem $A$ is NP-*hard* if every problem in NP is polynomially reducible to $A$; if $A$ is NP-hard and belongs to NP, then $A$ is said to be NP-*complete*. Thus, problems that are complete for NP are the most difficult problems in NP. In particular, it is unlikely that an NP-complete problem can be solved in polynomial time.

### 4.2. Summary of Results

In the analysis that follows, we take into account various qualitative properties regarding the kinds of constraint being allowed, by tracing the tractability frontier w.r.t. them. Formally, let $\mathtt{S}$ be a subset of the following set of symbols $\{\rightarrow, \rightsquigarrow, \not\rightarrow, \not\rightsquigarrow\}$. Let $\mathcal{C}[\mathtt{S}]$ denote all the possible constraints that can be built in a way that if $\rightarrow \notin \mathtt{S}$ (resp., $\rightsquigarrow \notin \mathtt{S}$, $\not\rightarrow \notin \mathtt{S}$, $\not\rightsquigarrow \notin \mathtt{S}$), then no edge (resp., path, negated edge, negated path) constraint is in $\mathcal{C}[\mathtt{S}]$). Let CN-MINING[S] denote the restriction of the problem over any set $\Pi$ of precedence constraints such that $\Pi \subseteq \mathcal{C}[\mathtt{S}]$. And, finally, let CN-EXISTENCE[S] be the *decision version* of this problem, where we have just to decide whether a solution exist at all and we are not asked to compute a solution, if any. Then, our results can be summarized as it is stated next (see also Figure 4).

THEOREM 4.1. *If* $\mathtt{S} \subseteq \{\rightarrow, \rightsquigarrow, \not\rightarrow\}$ *or* $\mathtt{S} \subseteq \{\not\rightsquigarrow\}$*, then* CN-MINING[S] *is feasible in polynomial time. Otherwise, it is even intractable to check whether there is a solution at all (formally, the problem* CN-EXISTENCE[S] *is* NP*-complete).* □

In particular, the proofs of the hardness results are based on linear logs only. Hence, according to Fact 3.3, for these results it is immaterial whether CN-MINING is defined over extended causal nets or over (standard) causal nets. These proofs are discussed in the rest of the section. Instead, the classes of constraints over which the mining prob-

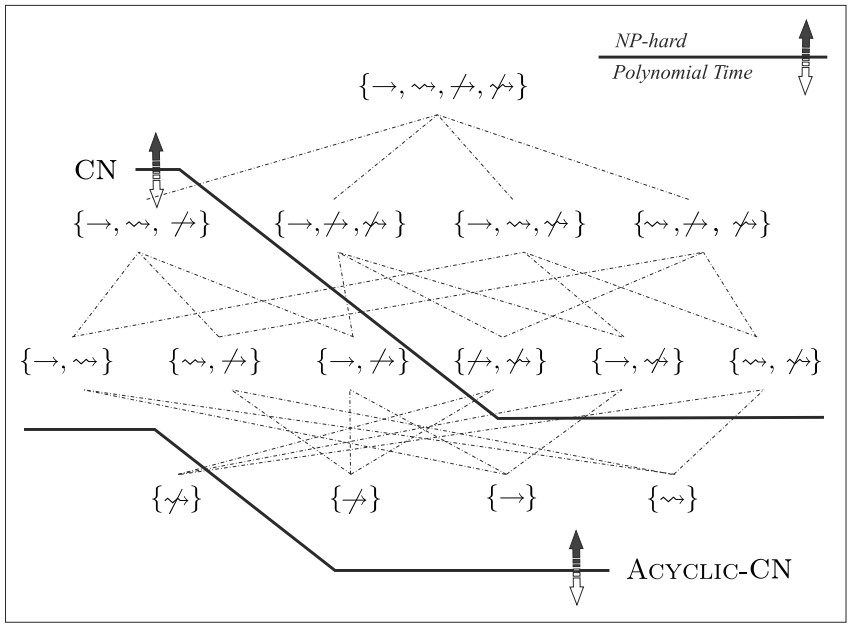



Fig. 4. Tractability frontiers. A set $\mathtt{S} \subseteq \{\rightarrow, \rightsquigarrow, \not\rightarrow, \not\rightsquigarrow\}$ above (resp., below) the frontier means that the corresponding problem is NP-hard (resp., feasible in polynomial time) on $\mathcal{C}[\mathtt{S}]$. Problem ACYCLIC-CN-EXISTENCE is defined in Section 4.3.

lems can be solved in polynomial time will be discussed in Section 5, where efficient solution approaches will be proposed.

### 4.3. Hardness Results

For the analysis that follows, we find convenient to introduce a variant of the problem CN-MINING, which we call ACYCLIC-CN-MINING. Given a set $\mathcal{A}$ of activities, a log $L$ with $\mathcal{A}(L) \subseteq \mathcal{A}$, and a set $\Pi$ of precedence constraints with $\mathcal{A}(\Pi) \subseteq \mathcal{A}$, the problem asks to compute a possibly extended causal net $\mathcal{C} = \langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ over $\mathcal{A}$ where $\mathcal{G}$ is acyclic, $\mathcal{C} \vdash L$, and $\mathcal{G} \models \Pi$, or check that no net with these properties exists.

We start the proofs of the hardness results stated in Theorem 4.1, by focusing on edge and path constraints imposed over the problem ACYCLIC-CN-EXISTENCE.

LEMMA 4.2. ACYCLIC-CN-EXISTENCE[$\{\rightarrow\}$] *is* NP*-hard.*

PROOF. Consider the *monotone one-in-three 3SAT* problem defined as follows. We are given as input a Boolean formula in conjunctive normal form $\Phi = c_1 \wedge \ldots \wedge c_m$ over the variables $X_1, \ldots, X_n$, where each clause $c_j$, with $j \in \{1, ..., m\}$, has the form $(t_{j,1} \vee t_{j,2} \vee t_{j,3})$, and $t_{j,1}, t_{j,2}$, and $t_{j,3}$ are three distinct variables. Note that $\Phi$ is always satisfiable by the truth assignment where all variables evaluate true. In fact, the problem asks whether there is a satisfying truth assignment where each clause has exactly one variable evaluating true (and hence two variables evaluating false). This problem is known to be NP-complete [Schaefer 1978].

Based on any formula $\Phi$ as above, we build the set $\mathcal{A}(\Phi)$ consisting of the variables in $\Phi$, which are transparently viewed as activities, plus the three distinguished activities $c$, $a_\perp$ and $a_\top$, where $a_\perp$ and $a_\top$ are as usual the starting and the terminating activity, respectively. So, we formally have $\mathcal{A}(\Phi) = \{c\} \cup \{X_1, ..., X_n\} \cup \{a_\perp, a_\top\}$.

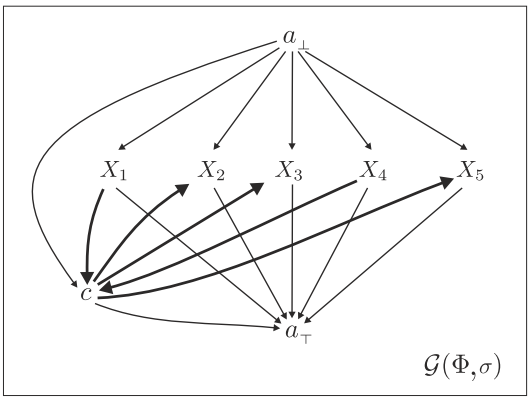



Fig. 5. Example reduction in the proof of Lemma 4.2.

Moreover, we build the set $\Pi(\Phi) \subseteq \mathcal{C}[\{\rightarrow\}]$ of edge constraints as follows. For each clause $c_j$, $\Pi(\Phi)$ contains the constraints $\{t_{j,1}, t_{j,2}, t_{j,3}\} \rightarrow \{c\}$, $\{c\} \rightarrow \{t_{j,1}, t_{j,2}\}$, $\{c\} \rightarrow \{t_{j,1}, t_{j,3}\}$, and $\{c\} \rightarrow \{t_{j,2}, t_{j,3}\}$. No further constraint is in $\Pi(\Phi)$.

We now claim that: There is a satisfying truth assignment to the variables of $\Phi$ such that each clause has exactly one variable evaluating true $\Leftrightarrow$ there is an acyclic dependency graph $\mathcal{G}$ (over $\mathcal{A}(\Phi)$) such that $\mathcal{G} \models \Pi(\Phi)$.

($\Rightarrow$) Assume that $\sigma$ is a satisfying truth assignment such that each clause has exactly one variable evaluating true. Consider the graph $\mathcal{G}(\Phi, \sigma) = (\mathcal{A}(\Phi), E)$ whose set of edges is defined as follows. For each variable $X_h$, with $h \in \{1, ..., n\}$, the edges $(a_\perp, X_h)$ and $(X_h, a_\perp)$ are in $E$. For each clause $c_j$ and each variable $t_{j,i}$ evaluating true (resp., false) in $\sigma$, the edge $(t_{j,i}, c)$ (resp., $(c, t_{j,i})$) is in $E$. The edges $(a_\perp, c)$ and $(c, a_\top)$ are in $E$, and no further edge is in $E$. As an example, the graph $\mathcal{G}(\Phi, \sigma)$ associated with the formula $\Phi = (X_1 \vee X_2 \vee X_3) \wedge (X_3 \vee X_4 \vee X_5)$ and the truth assignment $\sigma$, where $X_1$ and $X_4$ are the only variables evaluating true, is reported in Figure 5. In particular, note that the edges whose definition depend on $\sigma$ are depicted in bold.

We first show that $\mathcal{G}(\Phi, \sigma)$ is a dependency graph. Indeed, $a_\perp$ and $a_\top$ have no ingoing and outgoing edges, respectively. Moreover, for each activity $a \in \mathcal{A}(\Phi) \setminus \{a_\perp, a_\top\}$, the edges $(a_\perp, a)$ and $(a, a_\top)$ are in $E$, so that $a$ occurs in a path from $a_\perp$ to $a_\top$.

Then, we show that $\mathcal{G}(\Phi, \sigma)$ satisfies all the constraints in $\Pi(\Phi)$. Recall that $\sigma$ is a satisfying truth assignment such that each clause has exactly one variable evaluating true. Thus, for each clause $c_j$, with $j \in \{1, ..., m\}$, by construction there is an edge of the form $(t_{j,i}, c)$, so that the constraint $\{t_{j,1}, t_{j,2}, t_{j,3}\} \rightarrow \{c\}$ is satisfied. Moreover, there are also two edges of the form $(c, t_{j,i'})$ and $(c, t_{j,i''})$, where $i' \neq i$, $i'' \neq i$, $i' \neq i''$, and $\{i', i''\} \subseteq \{1, 2, 3\}$. Thus, the constraints $\{c\} \rightarrow \{t_{j,1}, t_{j,2}\}$, $\{c\} \rightarrow \{t_{j,1}, t_{j,3}\}$, and $\{c\} \rightarrow \{t_{j,2}, t_{j,3}\}$ are also satisfied, for each clause $c_j$. It follows that all constraints in $\Pi(\Phi)$ are satisfied by $\mathcal{G}(\Phi, \sigma)$, i.e., $\mathcal{G}(\Phi, \sigma) \models \Pi(\Phi)$.

In order to conclude, we now just need to point out that $\mathcal{G}(\Phi, \sigma)$ is acyclic. To this end, assume for the sake of contradiction that a cycle exists in $\mathcal{G}(\Phi, \sigma)$. Note that this cycle is necessarily defined over the variables and the distinguished activity $c$. Therefore, the set $E$ of edges must contain an edge of the form $(c, X_h)$ and an edge of the form $(X_h, c)$. By construction, the existence of the edge $(c, X_h)$ implies that $X_h$ is a variable evaluating false in $\sigma$. However, the existence of the edge $(X_h, c)$ implies that $X_h$ is a variable evaluating true in $\sigma$. Contradiction.

($\Leftarrow$) Assume that $\mathcal{G} = (V, E)$ is an acyclic dependency graph satisfying all the constraints in $\Pi(\Phi)$, and define the truth assignment $\sigma_\mathcal{G}$ such that $X_h$ evaluates true

if, and only if, the edge $(X_h, c)$ occurs in $E$. We first show that $\sigma_{\mathcal{G}}$ is satisfying. Indeed, for each clause $c_j$, with $j \in \{1, ..., m\}$, consider the associated constraint $\{t_{j,1}, t_{j,2}, t_{j,3}\} \rightarrow \{c\}$, and note that since $\mathcal{G} \models \Pi(\Phi)$, we are guaranteed about the existence of an edge from one of the variables in $\{t_{j,1}, t_{j,2}, t_{j,3}\}$ to $c$. Hence, for each clause $c_j$, at least one of the variables occurring in $c_j$ evaluates true in $\sigma_{\mathcal{G}}$, by definition of this assignment, which is therefore satisfying.

Now, we show that $\sigma_{\mathcal{G}}$ is a such that each clause has exactly one variable evaluating true. Assume, for the sake of contradiction, that a clause $c_j$ exists such that two variables, say $t_{j,i'}$ and $t_{j,i''}$ with $i' \neq i''$, evaluate true in $\sigma_{\mathcal{G}}$. Thus, the edges $(t_{j,i'}, c)$ and $(t_{j,i''}, c)$ are both in $E$. Consider then the constraint $\{c\} \rightarrow \{t_{j,i'}, t_{j,i''}\}$ associated with the clause $c_j$, and note that it prescribes that at least one of the edges in $\{(c, t_{j,i'}), (c, t_{j,i''})\}$ occurs in $E$. Assume, w.l.o.g., that $(c, t_{j,i'})$ is in $E$ and observe that we have eventually a cycle over $c$ and $t_{j,i'}$. Contradiction.

By Corollary 3.8 and Fact 3.3, the above entails that ACYCLIC-CN-MINING[$\{\rightarrow\}$] on input $\mathcal{A}(\Phi)$, the empty log, and the set $\Pi(\Phi)$ has a solution if, and only if, there is a satisfying truth assignment to the variables of $\Phi$ such that each clause has exactly one variable evaluating true. As the reduction is feasible in polynomial time, it follows that ACYCLIC-CN-EXISTENCE[$\{\rightarrow\}$] is NP-hard. □

A straightforward adaptation of the above proof, where each edge constraint is replaced by a path constraint over the same sets of activities, can be used to show that the problem remains intractable if we consider path constraints in place of edge constraints. The proof is reported below, for the sake of completeness.

LEMMA 4.3. ACYCLIC-CN-EXISTENCE[$\{\rightsquigarrow\}$] *is* NP-*hard.*

PROOF. Consider again the setting in the proof of Lemma 4.2 and, for any formula $\Phi$, define $\Pi'(\Phi) \subseteq \mathcal{C}[\{\rightsquigarrow\}]$ as the set of constraints obtained from $\Pi(\Phi)$ by replacing each edge constraint with the analogous path constraint. Therefore, for each clause $c_j$, $\Pi'(\Phi)$ contains the constraints $\{t_{j,1}, t_{j,2}, t_{j,3}\} \rightsquigarrow \{c\}$, $\{c\} \rightsquigarrow \{t_{j,1}, t_{j,2}\}$, $\{c\} \rightsquigarrow \{t_{j,1}, t_{j,3}\}$, and $\{c\} \rightsquigarrow \{t_{j,2}, t_{j,3}\}$. We now claim that: There is a satisfying truth assignment to the variables of $\Phi$ such that each clause has exactly one variable evaluating true $\Leftrightarrow$ there is an acyclic dependency graph $\mathcal{G}$ (over $\mathcal{A}(\Phi)$) such that $\mathcal{G} \models \Pi'(\Phi)$.

($\Rightarrow$) Assume that $\sigma$ is a satisfying truth assignment such that each clause has exactly one variable evaluating true, and consider the graph $\mathcal{G} = (\Phi, \sigma)$ built in the proof of Lemma 4.2. Recall that $\mathcal{G} \models \Pi(\Phi)$. Hence, we trivially have that $\mathcal{G} \models \Pi'(\Phi)$, in particular because all path constraints are satisfied by direct connections.

($\Leftarrow$) Assume that $\mathcal{G} = (V, E)$ is an acyclic dependency graph satisfying all the constraints in $\Pi'(\Phi)$, and define the truth assignment $\sigma'_{\mathcal{G}}$ such that $X_h$ evaluates true if, and only if, there is a path from $X_h$ to $c$ in $E$. Because of the constraints $\{t_{j,1}, t_{j,2}, t_{j,3}\} \rightsquigarrow \{c\}$ occurring in $\Pi'(\Phi)$ and associated with the clause $c_j$, for each $j \in \{1, ..., m\}$, $\sigma'_{\mathcal{G}}$ is satisfying. In order to conclude, we need to show that $\sigma'_{\mathcal{G}}$ is a such that each clause has exactly one variable evaluating true. Assume, for the sake of contradiction, that a clause $c_j$ exists such that two variables, say $t_{j,i'}$ and $t_{j,i''}$ with $i' \neq i''$, evaluate true in $\sigma_{\mathcal{G}}$. Thus, there is a path from $t_{j,i'}$ to $c$ and a path from $t_{j,i''}$ to $c$ are both in $E$. Consider then the constraint $\{c\} \rightsquigarrow \{t_{j,i'}, t_{j,i''}\}$ associated with the clause $c_j$, and note that it prescribes that there is a path from $c$ to at least one of the nodes in $\{t_{j,i'}, t_{j,i''}\}$. Assume, w.l.o.g., that a path from $c$ to $t_{j,i'}$ exists. Then, we have a cycle involving the activities $c$ and $t_{j,i'}$. Contradiction.

By the above result, Corollary 3.8, Fact 3.3, and the fact that the reduction is feasible in polynomial time, it follows that ACYCLIC-CN-EXISTENCE[$\{\rightsquigarrow\}$] is NP-hard. □

Let us now turn to the case of negated edge constraints, but still focusing on the acyclic variant of the mining problem.

LEMMA 4.4. ACYCLIC-CN-EXISTENCE[$\{\not\rightarrow\}$] *is* NP*-hard.*

PROOF. Let $\Pi = \{\{b_i^1, ..., b_i^{k_i}\} \rightarrow \{a_i^1, ..., a_i^{h_i}\} \mid i \in \{1, ..., m\}\} \subseteq \mathcal{C}[\{\rightarrow\}]$ be a (non-empty) set of edge constraints, such that $\{b_i^1, ..., b_i^{k_i}\} \cap \{a_i^1, ..., a_i^{h_i}\} = \emptyset$, for each $i \in \{1, ..., m\}$. For each $i \in \{1, ..., m\}$, let $c_i$ denote a fresh activity not in $\mathcal{A}(\Pi)$. Moreover, let $a_\perp$ and $a_\top$ be two activities not in $\mathcal{A}(\Pi)$ playing the role of the starting and terminating activity, respectively. Then, consider the log $L(\Pi) = \{t_1, ..., t_m\}$ such that $t_i = a_\perp b_i^1 ... b_i^{k_i} c_i a_i^1 ... a_i^{h_i} a_\top$, for each $i \in \{1, ..., m\}$. Moreover, consider the set $\Pi^\neg$ of negated edge constraints such that $\Pi^\neg = \{\neg(\{a_\perp\} \rightarrow \{c_i\}), \neg(\{c_i\} \rightarrow \{a_\top\}) \mid i \in \{1, ..., m\}\}$.

We claim that: There is an acyclic dependency graph $\mathcal{G}$ over the set $\mathcal{A}(\Pi) \cup \{a_\perp, a_\top\}$ of activities and such that $\mathcal{G} \models \Pi \Leftrightarrow$ there is an acyclic dependency graph $\mathcal{G}'$ over the set $\mathcal{A}(\Pi) \cup \{a_\perp, a_\top\} \cup \{c_1, ..., c_m\}$ of activities and such that $\mathcal{G}' \models \pi(L(\Pi)) \cup \Pi^\neg$.

($\Rightarrow$) Assume that $\mathcal{G} = (V, E)$ is an acyclic dependency graph over the set $\mathcal{A}(\Pi) \cup \{a_\perp, a_\top\}$ of activities and such that $\mathcal{G} \models \Pi$. Consider the graph $\mathcal{G}' = (V', E \cup E_1' \cup E_2')$ where $V' = V \cup \{c_1, ..., c_m\}$ and whose set of edges is built as follows. The set $E_1'$ contains the edges $(a_\perp, a)$ and $(a, a_\top)$, for each activity $a \in \mathcal{A}(\Pi)$, and no further edge is in $E_1'$. Moreover, for each $i \in \{1, ..., m\}$, and for each edge $(x, y) \in E$ such that $x \in \{b_i^1, ..., b_i^{k_i}\}$ and $y \in \{a_i^1, ..., a_i^{h_i}\}$, $E_2'$ includes the edges $(x, c_i)$ and $(c_i, y)$. No further edge is in $E_2'$. It is immediately checked that $\mathcal{G}'$ is an acyclic dependency graph.

Now, we first claim that $\mathcal{G}' \models \Pi^\neg$. Indeed, $\{a_\perp, a_\top\} \cap \mathcal{A}(\Pi) = \emptyset$, so that $E_2'$ does not include any edge of the form $(a_\perp, c_i)$ or of the form $(c_i, \top)$, with $i \in \{1, ..., m\}$. Let now $t_i$ be a trace in $L(\Pi)$, with $i \in \{1, ..., m\}$. Consider the set $\pi(t_i)$ of constraints induced by $t_i$ according to Definition 3.6. Because of the edges in $E_1'$, all constraints in $\pi(t_i)$ are trivially satisfied by $\mathcal{G}'$, but the two constraints $\{a_\perp, b_i^1, ..., b_i^{k_i}\} \rightarrow \{c_i\}$ and $\{c_i\} \rightarrow \{a_i^1, ..., a_i^{h_i}, a_\top\}$, because there is no edge in $\mathcal{G}'$ from $a_\perp$ to $c_i$, and from $c_i$ to $a_\top$. Now, recall that $\mathcal{G}$ satisfies $\Pi$ and hence, an edge $(x, y)$ occurs in $E$ such that $x \in \{b_i^1, ..., b_i^{k_i}\}$ and $y \in \{a_i^1, ..., a_i^{h_i}\}$. By construction of the edges in $E_2'$, we therefore have that $(x, c_i)$ and $(c_i, y)$ are edges of $\mathcal{G}'$, which proves that the two constraints are satisfied, too. Hence, $\mathcal{G}' \models \pi(L(\Pi))$.

($\Leftarrow$) Assume there is an acyclic dependency graph $\mathcal{G}' = (V', E')$ over the set $\mathcal{A}(\Pi) \cup \{a_\perp, a_\top\} \cup \{c_1, ..., c_m\}$ of activities and such that $\mathcal{G}' \models \pi(L(\Pi)) \cup \Pi^\neg$. Let $t_i$ be a trace in $L(\Pi)$, with $i \in \{1, ..., m\}$, and consider the two constraints $\{a_\perp, b_i^1, ..., b_i^{k_i}\} \rightarrow \{c_i\}$ and $\{c_i\} \rightarrow \{a_i^1, ..., a_i^{h_i}, a_\top\}$ in the set $\pi(t_i)$, which are satisfied by $\mathcal{G}'$. Since $\mathcal{G}' \models \Pi^\neg$, from the above we conclude that $\mathcal{G}' \models \{\{b_i^1, ..., b_i^{k_i}\} \rightarrow \{c_i\}, \{c_i\} \rightarrow \{a_i^1, ..., a_i^{h_i}\}\}$ holds.

Consider now the graph $\mathcal{G} = (V, E)$ where $V = V' \setminus \{c_1, ..., c_m\}$ and where the set of edges is defined as follows. The set $E$ contains the edges $(a_\perp, a)$ and $(a, a_\top)$, for each activity $a \in V$. Moreover, for each $i \in \{1, ..., m\}$, $E$ includes all edges of the form $(x, y)$ such that $\{(x, c_i), (c_i, y)\} \subseteq E'$, $x \in \{b_i^1, ..., b_i^{k_i}\}$, and $y \in \{a_i^1, ..., a_i^{h_i}\}$. Note that, in the light of the above observation, at least an edge of this kind is included in $E$, so that $\mathcal{G} \models \Pi$ holds. Note also that $\mathcal{G}$ is a dependency graph. Eventually, to conclude the proof, just note that $\mathcal{G}$ is acyclic, as the existence of a cycle in $\mathcal{G}$ would immediately entail the existence of a cycle in $\mathcal{G}'$.

Observe now that the log $L(\Pi)$ is linear, so that we are in the position of applying Corollary 3.8 and Fact 3.3 on the result proven above. By this way, we conclude that ACYCLIC-CN-MINING[$\{\rightarrow\}$] on input $\mathcal{A}(\Pi) \cup \{a_\perp, a_\top\}$, the empty log, and the set $\Pi$ has a solution if, and only if, ACYCLIC-CN-MINING[$\{\not\rightarrow\}$] on input $\mathcal{A}(\Pi) \cup \{a_\perp, a_\top\} \cup$

$\{c_1, ..., c_m\}$, the log $L(\Pi)$, and the set $\Pi^{\neg}$ has a solution. Eventually, by inspecting the proof of Lemma 4.2, note that the constraints used to prove the NP-hardness of ACYCLIC-CN-MINING[$\{\rightarrow\}$] are precisely of the form considered here for the set $\Pi$, and that the result is established even for logs that are empty. Therefore, we have reduced ACYCLIC-CN-MINING[$\{\not\rightarrow\}$] to ACYCLIC-CN-MINING[$\{\rightarrow\}$], so that the problem is hence shown to be NP-hard, too. □

To complete the picture, we now move to the case of arbitrary process models, i.e., of models that are not required to be acyclic. In this context, the picture is easily completed, as negated path constraints can be used to enforce acyclicity.

THEOREM 4.5. *The problems* CN-MINING[$\{\rightarrow, \not\rightarrow\}$]*,* CN-MINING[$\{\rightsquigarrow, \not\rightarrow\}$]*, and* CN-MINING[$\{\not\rightarrow, \not\rightarrow\}$] *are* NP*-hard.*

PROOF. Let $\Pi$ be a set of constraints in $\mathcal{C}[\{\rightarrow\}]$ (resp., $\mathcal{C}[\{\rightsquigarrow\}]$, $\mathcal{C}[\{\not\rightarrow\}]$). Based on $\Pi$, we build the set $\Pi'$ of constraints including all the constraints in $\Pi$, plus the novel constraint $\neg(\{a\} \not\rightarrow \{a\})$, for each activity $a$ taken from the underlying set of symbols $\mathcal{A}$. Of course, $\Pi'$ belongs to $\mathcal{C}[\{\rightarrow, \not\rightarrow\}]$ (resp., $\mathcal{C}[\{\rightsquigarrow, \not\rightarrow\}]$, $\mathcal{C}[\{\not\rightarrow, \not\rightarrow\}]$). In particular, the novel constraints just enforce that the resulting process model is acyclic. Then, ACYCLIC-CN-MINING[$\{\rightarrow\}$] (resp., ACYCLIC-CN-MINING[$\{\rightsquigarrow\}$], ACYCLIC-CN-MINING[$\{\not\rightarrow\}$]) has a solution on input the set $\mathcal{A}$ of activities, a log $L$, and the set $\Pi$ of constraints if, and only if, CN-MINING[$\{\rightarrow, \not\rightarrow\}$] (resp., CN-MINING[$\{\rightsquigarrow, \not\rightarrow\}$], CN-MINING[$\{\not\rightarrow, \not\rightarrow\}$]) has a solution on input $\mathcal{A}$, $L$, and $\Pi'$. The result therefore follows from Lemma 4.2, Lemma 4.3, and Lemma 4.4. □

To complete the proof of Theorem 4.1, we now show that the decision version of the mining problem belongs to the complexity class NP. Combined with the NP-hardness results discussed above, this entails the corresponding NP-completeness results.

THEOREM 4.6. CN-EXISTENCE[S] *is in* NP*, for each set* $\mathrm{S} \subseteq \{\rightarrow, \rightsquigarrow, \not\rightarrow, \not\rightarrow\}$.

PROOF. We need to decide about the existence of an extended causal net $\mathcal{C} = \langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ over $\mathcal{A}$ such that $\mathcal{C} \vdash L$ and $\mathcal{G} \models \Pi$. Thus, we can build a non-deterministic Turing machine that guesses a graph $\mathcal{G} = (V, E)$, with $V \subseteq \mathcal{A}$, and checks that $\mathcal{G} \models \Pi$. Moreover, for each trace $t \in L$, the machine guesses a sequence $\sigma_t$ of binding activities $\langle t[1], ib_1, ob_1 \rangle, ..., \langle t[len(t)], ib_n, ob_n \rangle$ and checks that $\sigma_t$ is valid w.r.t. $\mathcal{C}$. Note that, by Theorem 3.5, we can assume w.l.o.g. that the size of $\mathcal{G}$ is polynomially bounded. Thus, the overall size of the structures guessed by the machine is polynomially bounded. Moreover, the operations performed on them are feasible in polynomial time. Hence, the problem belongs to NP.

For the sake of completeness, note that if one restricts the problem to causal nets only, then the problem is still feasible in NP, as the machine has just to additionally check that $\mathcal{I}$ and $\mathcal{O}$ are sets rather than multi-sets as in the extended model. □

## 5. CLASSES OF TRACTABLE PRECEDENCE CONSTRAINTS

In this section, we define two algorithms to efficiently solve CN-MINING over the classes $\mathcal{C}[\{\rightarrow, \rightsquigarrow, \not\rightarrow\}]$ and $\mathcal{C}[\{\not\rightarrow\}]$, respectively. The algorithms will be designed as to make them still applicable over larger classes of constraints, by providing heuristic solution methods in these (NP-hard) cases. The efficiency of the methods and their efficacy as heuristics will be eventually assessed in Section 6.

In the exposition below, we assume that a set $\mathcal{A}$ of activities is given, together with a log $L$ such that $\mathcal{A}(L) \subseteq \mathcal{A}$ and with a set $\Pi$ of precedence constraints such that $\mathcal{A}(\Pi) \subseteq \mathcal{A}$. Accordingly, to simplify the notation, we shall omit to indicate $\mathcal{A}$, $L$, and $\Pi$, unless we want to explicitly point out a dependency from some of them. Moreover, we

shall denote by $\Pi^{\rightarrow}$ and $\Pi^{\rightsquigarrow}$ (resp., $\Pi^{\not\rightarrow}$ and $\Pi^{\not\rightsquigarrow}$) the sets of all positive (resp., negated) edge and path constraints in $\Pi$, respectively.

Both algorithms are based on a succession of graph manipulations, i.e., insertions and deletions of edges, starting with an initial (dependency) graph built from the log. In order to facilitate reasoning about such graph manipulations, for any directed graph $\mathcal{G} = (V, E)$ and for any set $E' \subseteq V \times V$ of edges, we define $\mathcal{G} \oplus E'$ as the graph $(V, E \cup E')$, and $\mathcal{G} \ominus E'$ as the graph $(V, E \setminus E')$. Moreover, we observe that while performing these operations, it is practically relevant to include (resp., exclude) only those edges that very likely (resp., hardly) witness the existence of true causal relations. In order to provide a formal measure of the 'quality' of an edge, we consider here the notion of *causal score* inspired by the works of Weijters and van der Aalst [2001], Weijters and van der Aalst [2003], and A.J.M.M. Weijters et al. [2006].

Let $\delta$ be a real number with $0 < \delta < 1$. Then, the causal score (w.r.t. $\delta$) is defined as the function $\mathbf{cs}_\delta : \mathcal{A} \times \mathcal{A} \mapsto \mathbb{R}$ such that $\mathbf{cs}_\delta(a_i, a_j) = D(a_i, a_j)/|\{t \in L \mid a_i = t[k], \text{ for some index } k\}|$, and where:

$$D(a_i, a_j) = \sum_{t \in L | a_i = t[h] \wedge a_j = t[k] \wedge h < k} \delta^{k-h-1} - \sum_{t \in L | a_i = t[k] \wedge a_j = t[h] \wedge h < k} \delta^{k-h-1}.$$

To illustrate the above definition, note that, for each trace $t[1]...t[n]$, $D(a_i, a_j)$ is incremented of a term $\delta^{k-h-1}$ if $a_i$ occurs $k - h$ positions before $a_j$, and decremented of the same term (in absolute value) if $a_i$ occurs $k - h$ positions after $a_j$. Moreover, the positive and the negative terms exponentially decrease at the growing of the distances between $a_i$ and $a_j$ in the traces. Note that $\mathbf{cs}_\delta(a_i, a_j) \leq 1$ holds, since $\delta < 1$.

### 5.1. Precedence Constraints without Negated Paths

We start by illustrating an algorithm to solve CN-MINING over $\mathcal{C}[\{\rightarrow, \rightsquigarrow, \not\rightarrow\}]$. The algorithm is design so that, over linear logs, a causal net is always returned as a solution, whenever a solution in fact exists. Instead, over arbitrary logs, the algorithm might well return an extended causal net.

*5.1.1. Precedence Graphs.* The starting point of the algorithm is the construction of the *precedence graph* $\texttt{PG}(L, \Pi)$ over $\mathcal{A}(L)$. In this graph, we avoid the edges that are forbidden to satisfy the negated edge constraints in $\Pi$, i.e., the edges in $\texttt{FE}(\Pi) = \{(x, y) \mid x \in S, y \in T, \neg(S \rightarrow T) \in \Pi^{\not\rightarrow}\}$. We start with an intuition of this concept.

EXAMPLE 5.1. Consider a log $L$ consisting of the trace $abcde$, and assume that $\Pi$ consists of the constraint $\neg(\{c\} \rightarrow \{d\})$. Then, the precedence graph $\texttt{PG}(L, \Pi)$ is the one illustrated in the left part of Figure 6. Intuitively, each activity $x \in \{a, b, c, d, e\}$ has an edge incoming (resp., outgoing) to any activity that precedes (resp., follows) $x$ in some trace. However, we avoid the edges that are forbidden according to the constraints. In particular, the graph does not contain the edge $(c, d)$, and the connectivity of $d$ is guaranteed via the edge $(b, d)$, i.e., $d$ is reached by the node closest to it in the trace $abcde$ and for which no violation in $\Pi^{\not\rightarrow}$ occurs. Moreover, $c$ can reach the final activity via to the direct connection $(c, e)$. ◁

Formally, $\texttt{PG}(L, \Pi) = (V, E)$ is defined as the directed graph where $V = \mathcal{A}(L)$ and where the set $E$ of its edges is built as follows. For each trace $t \in L$ and for each $i \in \{2, ..., len(t)\}$ (resp., $i \in \{1, ..., len(t) - 1\}$), if there is an index $j \in \{1, ..., i - 1\}$ (resp., $h \in \{i + 1, ..., len(t)\}$) such that $(t[j], t[i]) \notin \texttt{FE}(\Pi)$ (resp., $(t[i], t[h]) \notin \texttt{FE}(\Pi)$), then $E$ contains the edge $(t[j^*], t[i])$ (resp., $(t[i], t[h^*])$) witnessing that the property hold (i.e., $(t[j^*], t[i]) \notin \texttt{FE}(\Pi)$ (resp., $(t[i], t[h^*]) \notin \texttt{FE}(\Pi)$)) and having the highest causal score. No further edge is in $E$. The basic properties of precedence graphs are stated below.

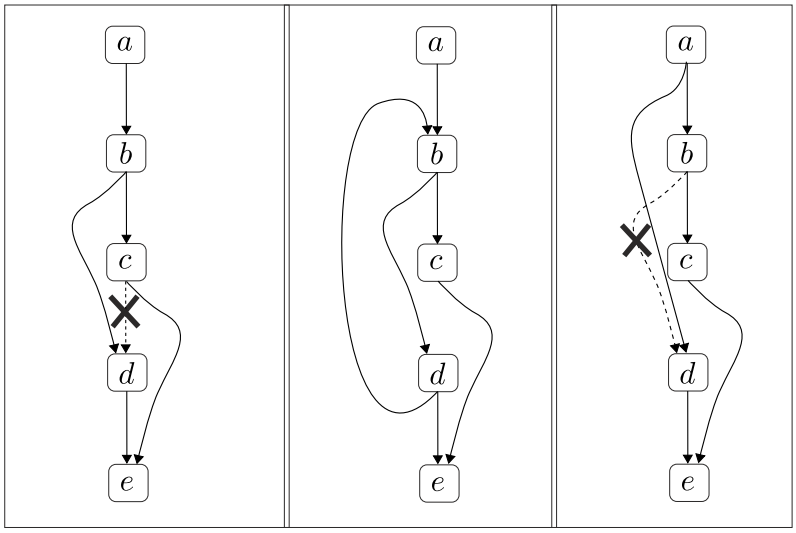



Fig. 6. Precedence graphs for the examples in Section 5.

LEMMA 5.2. *Assume that* $\texttt{PG}(L,\Pi) \not\models \pi(L)$ *holds. Then, there is no graph* $\mathcal{G}$ *such that* $\mathcal{G} \models \pi(L) \cup \Pi$. *Otherwise, i.e., if* $\texttt{PG}(L,\Pi) \models \pi(L)$, *then* $\texttt{PG}(L,\Pi)$ *is a dependency graph over* $\mathcal{A}(L)$ *such that* $\texttt{PG}(L,\Pi) \models \Pi^{\not\rightarrow}$.

PROOF. Assume that $\texttt{PG}(L,\Pi) \not\models \pi(L)$. Then, there is a trace $t \in L$ such that $\texttt{PG}(L,\Pi) \not\models \pi(t)$. By Definition 3.6, there are two possible cases. First, there might be an index $i \in \{2,...,len(t)\}$ such that $\{t[1],...,t[i-1]\} \rightarrow \{t[i]\}$ is not satisfied by $\texttt{PG}(L,\Pi)$. By construction of $\texttt{PG}(L,\Pi)$, this means that $(t[j],t[i]) \in \texttt{FE}(\Pi)$ holds, for each $j \in \{1,...,i-1\}$. Consider then a dependency graph $\mathcal{G}$ such that $\mathcal{G} \models \pi(L)$. Note that $\mathcal{G}$ must include an edge $(t[j^*],t[i])$, with $j^* \in \{1,...,i-1\}$, in order to satisfy the above constraint. However, $(t[j^*],t[i])$ is in $\texttt{FE}(\Pi)$, and there is a negated edge constraint $\neg(S \rightarrow T) \in \Pi$ such that $t[j^*] \in S$ and $t[i] \in T$. So, if $\mathcal{G} \models \pi(L)$, then $\mathcal{G} \not\models \Pi$.

Assume now that $\texttt{PG}(L,\Pi) \models \pi(L)$. We first observe that $\texttt{PG}(L,\Pi)$ contains the two activities $a_{\perp}$ and $a_{\top}$, playing the role of the starting and terminating activity. In particular, $a_{\perp}$ and $a_{\top}$ have no ingoing and outgoing edges, respectively. Consider then any other activity $a$ occurring in $\texttt{PG}(L,\Pi)$, and note that there is a trace $t \in L$ and an index $i \in \{2,...,len(t)-1\}$ such that $t[i] = a$. Since $\texttt{PG}(L,\Pi) \models \pi(L)$, we are then guaranteed about the existence of two edges having the form $(t[j],t[i])$ and $(t[i],t[h])$, with $j \in \{1,...,i-1\}$ and $h \in \{i+1,...,len(t)\}$. By structural induction on the index $i$, it then follows that $t[i]$ occurs in a path connecting $t[1] = a_{\perp}$ to $t[len(t)] = a_{\top}$. Hence, $\texttt{PG}(L,\Pi)$ satisfies all the conditions for being a dependency graph over $\mathcal{A}(L)$. Then, in order to conclude the proof, we need to show that $\texttt{PG}(L,\Pi) \models \Pi^{\not\rightarrow}$. Indeed, assume by contradiction that a negated edge constraint $\neg(S \rightarrow T)$ exists in $\Pi$ such that $x \in S$, $y \in T$, and the edge $(x,y)$ is in $\texttt{PG}(L,\Pi)$. Hence, $(x,y) \in \texttt{FE}(\Pi)$, which is impossible as all edges of $\texttt{PG}(L,\Pi)$ do not belong to $\texttt{FE}(\Pi)$, by construction. □

*5.1.2. Positive Precedence Constraints.* According to the above result, precedence graphs can be used as a preliminary representation of inter-activity dependencies. However, these graphs do not guarantee that positive constraints are satisfied. Therefore, we define a method to identify the edges needed to satisfy the positive constraints in $\Pi$.

EXAMPLE 5.3. Consider the precedence graph $\texttt{PG}(L, \Pi)$ discussed in Example 5.1 and assume that $\Pi$ also contains the constraint $\{d\} \rightsquigarrow \{b\}$. Note that $\texttt{PG}(L, \Pi)$ does not satisfy the positive constraint $\{d\} \rightsquigarrow \{b\}$. Thus, we have to update the graph by including a path starting from $d$ and terminating into $b$ and where the edge $(c, d)$ does not occur in it. Of course, in this case we can just simply add and edge from $d$ to $b$, as it is shown in the center of Figure 6. $\lhd$

Let $\mathcal{G} = (V, E)$ be a dependency graph with $V \supseteq \mathcal{A}(\Pi)$. For each pair of nodes $x, y \in V$, define the *weight of* $(x, y)$ *for* $\mathcal{G}$ *w.r.t.* $\delta$, as the real number $w_\delta(\mathcal{G}, x, y)$ such that[4]:

$$w_\delta(\mathcal{G}, x, y) = \begin{cases} 0 & \text{if } (x, y) \in E \\ +\infty & \text{if } x = a_\top \text{; or } y = a_\bot \text{; or } x \in S, y \in T, \text{ and } \neg(S \to T) \in \Pi \\ 2 - \mathbf{cs}_\delta(x, y) & \text{in the remaining cases} \end{cases}$$

If $a_1, ..., a_h$ is a sequence of nodes forming a path in $\mathcal{G}$, with $h \geq 2$, then we define its weight $w_\delta(\mathcal{G}, a_1, ..., a_h)$ as the value $\sum_{i=1}^{h-1} w_\delta(\mathcal{G}, a_i, a_{i+1})$. Note that $w_\delta(\mathcal{G}, a_1, ..., a_h) \geq 0$, because $\mathbf{cs}_\delta(x, y) \leq 1$ holds, for each pair $x, y$. Moreover, for each path constraint $S \rightsquigarrow T$ (resp., edge constraint $S \to T$) in $\Pi$, we define $\texttt{BestPath}_\delta(\mathcal{G}, S \rightsquigarrow T) = a_1, ..., a_h$ (resp. $\texttt{BestEdge}_\delta(\mathcal{G}, S \to T) = a_1, a_2$) as the minimum-weight path (resp., minimum-weight edge) such that $a_1 \in S$ and $a_h \in T$. Note that, if there is a precedence constraint $S \rightsquigarrow T$ (resp., $S \to T$) such that the weight of $\texttt{BestPath}_\delta(\mathcal{G}, S \rightsquigarrow T)$ (resp., $\texttt{BestEdge}_\delta(\mathcal{G}, S \to T)$) is $+\infty$, then all the paths (resp., edges) connecting any activity in $S$ to any activity in $T$ must include an edge that cannot occur in a model of $\Pi^{\not\to}$. Hence, the following is immediately established.

LEMMA 5.4. *Let* $\mathcal{G}$ *be a graph such that* $\mathcal{G} \models \Pi^{\not\to}$. *If there is a path constraint* $S \rightsquigarrow T$ *(resp., edge constraint* $S \to T$*) such that the weight of* $\texttt{BestPath}_\delta(\mathcal{G}, S \rightsquigarrow T)$ *(resp.,* $\texttt{BestEdge}_\delta(\mathcal{G}, S \to T)$*) is* $+\infty$*, then* CN-MINING *has no solution (on* $\mathcal{A}$*,* $L$*, and* $\Pi$*).*

If the hypothesis in the above lemma does not hold, in order to satisfy a path constraint $S \rightsquigarrow T$ (resp., edge constraint $S \to T$) we can just update the graph $\mathcal{G}$ as to include $\texttt{BestPath}_\delta(\mathcal{G}, S \rightsquigarrow T)$ (resp., $\texttt{BestEdge}_\delta(\mathcal{G}, S \to T)$) as a path (resp., an edge).

*5.1.3. Putting It All Together.* Now that we have discussed all the salient ingredients, we can illustrate the algorithm COMPUTE-CN shown in Figure 7.

The algorithm starts in step 1 by adding to $\Pi$ a set of path constraints stating that each activity $a \in \mathcal{A} \setminus \mathcal{A}(L)$, i.e., not occurring in the log, has still to occur in a path from the starting activity to the terminating one. Step 2 is responsible of checking whether the precedence graph supports the constraints induced by the log. Note that we directly check the satisfaction of the constraints induced by the log. Indeed, if this graph does not satisfy the condition, then we report that no solution exists all.

In step 3, a graph $\mathcal{G}_1$ is initially built over the nodes in $\mathcal{A}$ and the edges in $\texttt{PG}(L, \Pi)$. In the subsequent steps, $\mathcal{G}_k$ denotes the graph obtained from $\mathcal{G}_1$ after having performed $k - 1$ manipulations on it. The process starts with step 4, which is a heuristic step that removes any edge whose causal score is below a given threshold $\tau$, received as an additional parameter. In fact, it can be checked that for $\tau = 1$, this is immaterial and the graph remains unchanged. Step 5–11 (resp., 12–18) are responsible of adding a number of edges to the precedence graph as to satisfy all edge constraints (resp., path constraints), according to the strategy described in Section 5.1.2. A failure in this step leads the algorithm to exit by reporting that no solution exist at all.

Finally, a (possibly extended) causal net $\langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ is built and returned as output. This latter step is carried out by a function that just implements the strategy of in-

[4]Here, $+\infty$ stands for any large enough positive real number, e.g., $+\infty > |\mathcal{A}|^2 \times \max_{(x,y)\in E} \mathbf{cs}_\delta(x, y)$.

**Input**: A set $\mathcal{A}$ of activities, with $a_\perp$ ($a_\top$) being the starting (terminating) one, a log $L$ with $\mathcal{A}(L) \subseteq \mathcal{A}$, and a set $\Pi \in \mathcal{C}[\{\rightarrow, \rightsquigarrow, \not\rightarrow\}]$ of precedence constraints with $\mathcal{A}(\Pi) \subseteq \mathcal{A}$;
**Parameters**: Two real numbers $\delta > 0$ and $\tau > 0$;
**Output**: A triple $\langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$, or 'no';

1. $\Pi := \Pi \cup \{\{a_\perp\} \rightsquigarrow \{a\}, \{a\} \rightsquigarrow \{a_\top\} \mid a \in \mathcal{A} \setminus \mathcal{A}(L)\}$;
2. **if** $\mathtt{PG}(L, \Pi) \not\models \pi(L)$ **then return** 'no' (and HALT);
3. **let** $\mathcal{G}_1 := (\mathcal{A}, E_1)$ be the graph where $E_1$ consists of the edges in $\mathtt{PG}(L, \Pi)$;
4. $\mathcal{G}_1 := \mathcal{G}_1 \ominus \{(x, y) \mid \mathbf{cs}_\delta(x, y) < \tau\}$, $k := 1$;
5. $\mathcal{F} := \{S \rightarrow T \in \Pi \mid \mathcal{G}_k \not\models S \rightarrow T\}$;
6. **while** $\mathcal{F} \neq \emptyset$ **do**
7. **let** $S \rightarrow T$ be in $\mathcal{F}$, and **let** $\mathtt{BestEdge}_\delta(\mathcal{G}_k, S, T) = a_1, a_2$;
8. **if** $w_\delta(\mathcal{G}_k, a_1, a_2) \geq +\infty$ **then return** 'no' (and HALT);
9. $\mathcal{G}_{k+1} := \mathcal{G}_k \oplus \{(a_1, a_2)\}$, $k := k + 1$;
10. $\mathcal{F} := \{S \rightarrow T \in \Pi \mid \mathcal{G}_k \not\models S \rightarrow T\}$;
11. **end while**
12. $\mathcal{F} := \{S \rightarrow T \in \Pi \mid \mathcal{G}_k \not\models S \rightsquigarrow T\}$;
13. **while** $\mathcal{F} \neq \emptyset$ **do**
14. **let** $S \rightsquigarrow T$ be in $\mathcal{F}$, and **let** $\mathtt{BestPath}_\delta(\mathcal{G}_k, S \rightsquigarrow T) = a_1, ..., a_h$;
15. **if** $w_\delta(\mathcal{G}_k, a_1, ..., a_h) \geq +\infty$ **then return** 'no' (and HALT);
16. $\mathcal{G}_{k+1} := \bar{\mathcal{G}}_k \oplus \{(a_i, a_{i+1}) \mid i \in \{1, ..., h-1\}\}$, $k := k + 1$;
17. $\mathcal{F} := \{S \rightsquigarrow T \in \Pi \mid \mathcal{G}_k \not\models S \rightsquigarrow T\}$;
18. **end while**
19. **return** computeBindings($\mathcal{G}_k$,$L$);

**Function** computeBindings($\mathcal{G}$, $L$), with $\mathcal{G} = (\mathcal{A}, E)$;
$\forall a \in \mathcal{A}$, let $\mathcal{I}(a) := I_a$ and $\mathcal{O}(a) := O_a$, where
$I_a = \{(x, a) \mid (x, a) \in E\}$ and $O_a = \{(a, y) \mid (a, y) \in E\}$;
**for each** trace $t$ in $L$, and for each $i \in \{1, ..., len(t)\}$ **do**
$\mathcal{I}(t[i]) := \mathcal{I}(t[i]) \cup \{(t[j], t[i]) \in E \mid \mathbf{cs}_\delta(t[j], t[i]) \geq \tau, j < i\}$; (*to be treated as multi-sets*)
$\mathcal{O}(t[i]) := \mathcal{O}(t[i]) \cup \{(t[i], t[j]) \in E \mid \mathbf{cs}_\delta(t[i], t[j]) \geq \tau, i < j\}$; (*to be treated as multi-sets*)
**end for**
**return** $\langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$; (* possibly extended causal net *)

Fig. 7. Algorithm COMPUTE-CN (on $\mathcal{C}[\{\rightarrow, \rightsquigarrow, \not\rightarrow\}]$).

cluding an input (resp., output) binding for each trace $t$, and of defining this binding with all predecessor (resp., successor) activities in $t$. Moreover, in order to formally guarantee that $\langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ is a (possibly extended) causal net, we add the binding $I_a$ (resp., $O_a$) to $\mathcal{I}(a)$ (resp., $\mathcal{O}(a)$), for each activity $a$ in $\mathcal{G}$, being defined as the union of all incoming (resp. outgoing) edges. As above, we heuristically get rid of any edge whose causal score is below $\tau$. The correctness is stated next.

THEOREM 5.5. *The following properties hold on* COMPUTE-CN*, receiving as input* $\mathcal{A}$*,* $L$*,* $\Pi \in \mathcal{C}[\{\rightarrow, \rightsquigarrow, \not\rightarrow\}]$*, the parameter* $\delta$*, and for* $\tau = 1$*:*

- *if it returns 'no', then there is no solution;*
- *if it returns* $\langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ *and* $L$ *is (resp., is not) a linear log, then* $\langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ *is a (resp., possibly extended) causal net such that* $\langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle \vdash L$ *and* $\mathcal{G} \models \Pi$.

PROOF. By Lemma 5.2 and Theorem 3.11, if the algorithm returns 'no' at step 2, then we are guaranteed that there is no solution. Assume that we are not in this case. Then, by Lemma 5.2, we know that $\mathtt{PG}(L, \Pi)$ is a dependency graph over $\mathcal{A}(L)$ with $\mathtt{PG}(L, \Pi) \models \pi(L)$ and $\mathtt{PG}(L, \Pi) \models \Pi^{\not\rightarrow}$. So, $\mathcal{G}_1$ is such that $\mathcal{G}_1 \models \pi(L)$ and $\mathcal{G}_1 \models \Pi^{\not\rightarrow}$.

Consider now the steps 5–18. Note that whenever the current graph $\mathcal{G}$ is updated (in steps 9 and 16), we are guaranteed that any edge $(x, y)$ that is inserted does not

violate negated edge constraints, for otherwise the weight of $\texttt{BestEdge}_\delta(\mathcal{G}, S \to T)$ or $\texttt{BestPath}_\delta(\mathcal{G}, S \rightsquigarrow T)$ would be $+\infty$. Moreover, in the remaining steps, no edge is added. Therefore, $\mathcal{G}_k \models \Pi^{\not\rightarrow}$ holds, for each $k \geq 1$. In fact, steps 6–18 try to enforce the satisfaction of the edge and the path constraints. By Lemma 5.4 and since $\mathcal{G}_k \models \Pi^{\not\rightarrow}$ holds, for each $k \geq 1$, we derive that if the the algorithm returns 'no', then we are guaranteed that no solution exists at all. Again, let us assume that this is not the case, in order to complete the analysis. So, we have now reached step 19, where we are guaranteed that the current graph $\mathcal{G}_k$ is such that $\mathcal{G}_k \models \Pi$. Moreover, as we have just added edges and initially $\mathcal{G}_1 \models \pi(L)$ holds, then $\mathcal{G}_k \models \pi(L)$ holds, too. In particular, the subgraph of $\mathcal{G}_k$ induced over the nodes in $\mathcal{A}(L)$ is a dependency graph. Moreover, because of the constraints added in step 1, every other activity in $a \in \mathcal{A} \setminus \mathcal{A}(L)$ is also in a path from the starting to the terminating activity. Hence, $\mathcal{G}_k$ is a dependency graph.

Finally, step 19 invokes a function that equips $\mathcal{G}_k$ with the sets $\mathcal{I}$ and $\mathcal{O}$. The reader may check that the function is a constructive implementation of the proof of Theorem 3.11 (which basically founds on Theorem 2.7). Therefore, the tuple $\mathcal{C} = \langle \mathcal{G}_k, \mathcal{I}, \mathcal{O} \rangle$ returned as output is such that $\mathcal{C} \vdash L$ and $\mathcal{G}_k \models \Pi \setminus \Pi^{\not\rightarrow}$. Hence, if $\Pi \in \mathcal{C}[\{\rightarrow, \rightsquigarrow, \not\rightarrow\}]$, then we have actually computed a solution. □

Note that the net computed by the function COMPUTE-CN might well be an extended causal net. In fact, in Section 6 we shall discuss heuristics making this scenario rather unlikely, as we have also verified in our experiential activity. Moreover, in absence of negated edge constraints, it is easy to see that there is always a causal net that is a solution and that can be trivially built from the dependency graph containing all possible edges—heuristics can be also defined to remove unnecessary edges, but we do not expand on this aspect. However, we leave it open the question about whether CN-MINING is still tractable when restricted over causal nets only, and when negated edge constraints can occur in addition to positive constraints.

*Implementation issues and computation time analysis.* Let $n_t = |L|$ and $n_a = |\mathcal{A}|$ be the number of traces and activities in input, and $l_t$ be the maximal trace length. Let $n^{\rightarrow}$ be the number of precedence constraints, and (resp., $n^{\not\rightarrow}$, $n^{\rightsquigarrow}$) be the number of constraints of type $\mathcal{C}[\{\rightarrow\}]$ (resp., $\mathcal{C}[\{\not\rightarrow\}]$, $\mathcal{C}[\{\rightsquigarrow\}]$) given as input. Moreover, let $n_c$ be the total number of input constraints (independently of the type), and $k$ be the maximum number of elements in either side of them all, i.e., the maximum size of all their associated sets $S$ and $T$.

In the implementation, constraints are indexed with a number in $\{1, \ldots, n_c\}$, and two arrays of activity identifiers's lists are used to keep trace of the activities appearing in the left and right sides, respectively, of each constraint. Causal scores, forbidden edges, and dependency graphs are all represented via $n_a \times n_a$ matrices. The initialization of these structures and steps 1-5 can be done in $O(n_t \times l_t^2 + (n^{\rightarrow} + n^{\not\rightarrow} + n^{\rightsquigarrow}) \times k + n_a^2)$ time, where the leftmost term corresponds to both computing the causal score matrix and $\texttt{PG}(L, \Pi)$, and assessing whether this latter satisfies $\pi(L)$.

The cost of the first loop (steps 6-11) is $O(n^{\rightarrow} \times k^2)$. This accounts for checking the satisfaction of all positive edge constraints, and for the cost of computing the "best edge" for each of them. Since no edge is removed, each constraint is considered at most in one iteration, and will remain satisfied in the subsequent ones. The second loop (steps 13-18) resemble the first one, but for the focus on computing "best paths" rather than "best edges". For each constraint, the task can be carried out via the classical Djikstra algorithm (possibly provided with an artificial "super"-source node, linked to all activities in the lefthand set of the constraint), with a cost $O(n_a^2)$. Since $k \leq n_a$ and we have $n^{\rightsquigarrow}$ constraints, the overall cost is $O(n^{\rightsquigarrow} \times n_a^2)$.

**Input**: A set $\mathcal{A}$ of activities, with $a_\perp$ ($a_\top$) being the starting (terminating) one, a log $L$ with $\mathcal{A}(L) \subseteq \mathcal{A}$, and a set $\Pi \in \mathcal{C}[\{\not\rightsquigarrow\}]$ of precedence constraints with $\mathcal{A}(\Pi) \subseteq \mathcal{A}$;
**Parameters**: Two real numbers $\delta > 0$ and $\tau > 0$;
**Output**: A triple $\langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$, or 'no';

1. **let** $\mathcal{G}_1 := \mathtt{PG}(L, \emptyset)$;
2. $\mathcal{G}_1 := \mathcal{G}_1 \ominus \{(x, y) \mid \mathbf{cs}_\delta(x, y) < \tau\}$, $k := 1$;
3. **while** $\mathtt{FakeEdge}(\mathcal{G}_k) \neq \emptyset$ **do**
4. **let** $(x^*, y^*) := \arg\min_{(x,y) \in \mathtt{FakeEdge}(\mathcal{G}_k)} \{\mathbf{cs}_\delta(x, y)\}$;
5. **let** $z, w$ be any pair of nodes in $\mathcal{G}_k$ **such that**
   **(c1)** $z \in \mathtt{succ}(\mathcal{G}_k, x^*)$, $w \in \mathtt{pred}(\mathcal{G}_k, y^*)$ and there are no paths from either $z$ to a node in $T \cup \{w\}$, or from a node in $S \cup \{z\}$ to $w$.
   **(c2)** $\forall z', w'$ satisfying **(c1)**, $\mathbf{cs}_\delta(x^*, z) + \mathbf{cs}_\delta(w, y^*) \geq \mathbf{cs}_\delta(x^*, z') + \mathbf{cs}_\delta(w', y^*)$;
6. $\mathcal{G}_{k+1} := \mathcal{G}_k \oplus \{(x^*, z), (w, y^*)\} \ominus \{(x^*, y^*)\}$, $k := k + 1$
7. **end while**
8. **if** $\mathcal{G}_k \not\models \Pi^{\not\rightsquigarrow}$ **then return** 'no' (and HALT);
9. **return** computeBindings($\mathcal{G}_k$,$L$);

Fig. 8. Algorithm COMPUTE-CN (on $\mathcal{C}[\{\not\rightsquigarrow\}]$).

Finally, the input and output bindings of each activity can be kept in two dictionaries, whose entries are multi-sets of activities, with each activity identifying the other vertex of an associated incoming/outgoing edge. Such multi-sets (which reduces to sets in the case of pure causal nets) are simply stored as vectors, encoding edges' multiplicity. Regarding such vectors as strings of length $n_a$ (i.e. the sequence of occurrence counts, one per edge), these dictionaries can be implemented as a tries (i.e. prefix trees), which can be built in $O(n_t \times l_t \times n_a)$. This cost accounts for (i) scanning each input trace $s$ in both directions (forward and backward) with an index, say $i$, while incrementally building the multi-set of all activities in its first (resp., last) $i$ positions, and for (ii) generating the resulting multi-set and adding it to input (resp., output) bindings of $s[i]$. [5]

In total, we get $O(n_t \times l_t \times \max(l_t, n_a) + n_a^2 \times (1 + n^{\rightsquigarrow}) + n^{\rightarrow} \times k^2 + n^{\not\rightarrow} \times k)$.

### 5.2. The Case of Negated Path Constraints

We now present an algorithm to solve CN-MINING on the class $\mathcal{C}[\{\not\rightsquigarrow\}]$, which is illustrated in Figure 8. Most of the ingredients discussed so far will still play a role, but the approach is substantially different. The algorithm starts again with the construction of the precedence graph, but this time building it without taking care of negated edge constraints—note that edges with low causal scores are removed. That is, we start with the graph $\mathtt{PG}(L, \emptyset)$, whose main properties are stated below and are easily seen to hold by inspecting the proof of Lemma 5.2.

LEMMA 5.6. $\mathtt{PG}(L, \emptyset)$ *is a dependency graph over* $\mathcal{A}(L)$ *and* $\mathtt{PG}(L, \emptyset) \models \pi(L)$.

The algorithm subsequently breaks any path that witnesses a violation of the negated path constraints in $\Pi$. Formally, if $\mathcal{G} = (V, E)$ is a dependency graph over

[5]The idea of using an efficient data structure to store the bindings of each node in an extended (resp., pure) causal net stems from the observation that a safe upper bound to their number is $\min(l_t^{n_a}, l_t \times n_t)$ (resp., $\min(2^{n_a}, l_t \times n_t)$) — even though, in many practical cases, the it can be assumed as constant. Several finer-grain policies can be adopted to reduce the actual (space and/or time) costs of maintaing such structures. For example, one can store only once the elements shared by multiple tries, adopt a more concise representation for any multi-set (e.g., as a pair of the form $\langle activity, occurrences \rangle$), as well as resort to radix ("patricia") tries. Anyway, none of these improvements change our asymptotical analysis.

$\mathcal{A}$, then the set of all *fake edges* is defined as $\texttt{FakeEdge}(\mathcal{G}) = \{(x,y) \in E \mid \neg(S \rightsquigarrow T) \in \Pi, \mathcal{G} \models \{\{S\} \rightsquigarrow \{x\}, \{y\} \rightsquigarrow \{T\}\}\}$. Intuitively, we would like to remove these edges from the graph, but while doing so we might miss the ability of supporting the log $L$. In some cases, we need in fact to repair the connectivity.

Let $\mathcal{G} = (V, E)$ be a graph over $\mathcal{A}$. Let $y$ be an activity in $V$. The set $\texttt{pred}(\mathcal{G}, y)$ of the *causal predecessors* of $y$ in $L$ is defined as the set of all the activities $w \in V$ such that there is a path from $a_{\perp}$ to $w$ in $\mathcal{G} \setminus \{(x', y) \mid (x', y) \in E\}$ and, for each trace $t \in L$ where $y$ occurs, i.e., $y = t[i]$ for some $i \in \{1, ..., len(t)\}$, then $w$ also occurs in $t$ before $y$, i.e., $w = t[j]$ where $j < i$ holds. Symmetrically, let $x$ be an activity in $V$. The set $\texttt{succ}(\mathcal{G}, x)$ of the *causal successors* of $x$ in $L$ is the set of all the activities $z \in V$ such that there is a path from $z$ to $a_{\top}$ in $\mathcal{G} \setminus \{(x, y') \mid (x, y') \in E\}$ and, for each trace $t \in L$ where $x$ occurs, i.e., $x = t[i]$ for some $i \in \{1, ..., len(t)\}$, then $z$ also occurs in $t$ after $x$, i.e., $z = t[j]$ where $j > i$ holds. Note that the following is immediate.

LEMMA 5.7. *Let $\mathcal{G} = (V, E)$ be a dependency graph such that $\mathcal{G} \models \pi(L)$, and let $(x, y)$ be an edge in $E$, hence with $x \neq a_{\perp}$ and $y \neq a_{\top}$. Then, $a_{\perp} \in \texttt{pred}(\mathcal{G}, y)$ and $a_{\top} \in \texttt{succ}(\mathcal{G}, x)$.*

The following result shows how to safely update a given dependency graph $\mathcal{G}$.

LEMMA 5.8. *Let $\mathcal{G} = (V, E)$ be a dependency graph such that $\mathcal{G} \models \pi(L)$, let $(x, y)$ be an edge, and let $w \neq a_{\perp}$ and $z \neq a_{\top}$ be in $\texttt{pred}(\mathcal{G}, y)$ and $\texttt{succ}(\mathcal{G}, x)$, respectively. Then, $\mathcal{G}' = \mathcal{G} \oplus \{(w, y), (x, z)\} \ominus \{(x, y)\}$ is a dependency graph such that $\mathcal{G}' \models \pi(L)$.*

PROOF. Note first that, since $\mathcal{G}$ is a dependency graph, it must be the case that $x \neq a_{\top}$ and $y \neq a_{\perp}$. Moreover, it can be also checked that $w \cap \{y, a_{\top}\} = \emptyset$ and $z \cap \{x, a_{\perp}\}$, by definition of causal predecessor and successor. Therefore, the graph $\mathcal{G}' = \mathcal{G} \oplus \{(w, y), (x, z)\} \ominus \{(x, y)\}$ is such that the starting activity $a_{\perp}$ and the terminating activity $a_{\top}$ have no ingoing and outgoing edges, respectively. Consider now an activity $a \in V \setminus \{a_{\perp}, a_{\top}\}$. Note that, since $\mathcal{G}$ is a dependency graph, either *(i)* there is a path in $\mathcal{G} \ominus \{(x, y)\}$ from $a_{\perp}$ to $a$, or *(ii)* the edge $(x, y)$ occurs in each path in $\mathcal{G}$ from $a_{\perp}$ to $a$. In the case *(i)*, we immediately conclude that there is a path from $a_{\perp}$ to $a$ in $\mathcal{G}'$, too. Hence, let us focus on case *(ii)*. Recall that in $\mathcal{G}'$ we have the edge $(w, y)$ where $w \neq x$. In particular, there is a path from $a_{\perp}$ to $w$ in $\mathcal{G} \setminus \{(x', y) \mid (x', y) \in E\}$. Hence, we have again derived that there is a path from $a_{\perp}$ to $a$ in $\mathcal{G}'$, too. By symmetric arguments, we derive also that there is a path from $a$ to $a_{\top}$ in $\mathcal{G}'$. Hence, $\mathcal{G}'$ is a dependency graph.

In order to conclude the proof, we have now to show that $\mathcal{G}' \models \pi(L)$, too. Indeed, assume by contradiction that a trace $t$ exists in $L$ such that $\mathcal{G}'$ does not model $\pi(t)$. Given the differences between $\mathcal{G}$ and $G'$ and since the constraints induced by the traces are only positive ones, it must be the case that the removal of the edge $(x, y)$ is the source of the violation. Formally, we can be in one of the following two scenarios:

(1) It holds that $y = t[i]$ and $x = t[j]$, with $j < i$, and $\mathcal{G}'$ does not satisfy the constraint $\{t[1], ..., t[i-1]\} \rightarrow \{y\}$, which is instead satisfied by $\mathcal{G}$ precisely because of the edge $(x, y)$. However, we recall that the edge $(w, y)$ occurs in $\mathcal{G}'$ and that $w$ occurs before $y$ in any trace where $y$ occurs. That is, there is an index $j' \in \{1, ..., i-1\}$ such that $w = t[j']$ where $j' < i$ holds. Hence, $\mathcal{G}'$ satisfies the constraint. Contradiction.
(2) It holds that $x = t[i]$ and $y = t[j]$, with $i < j$, and $\mathcal{G}'$ does not satisfy the constraint $\{x\} \rightarrow \{t[i+1], ..., t[len(t)]\}$, which is instead satisfied by $\mathcal{G}$ precisely because of the edge $(x, y)$. However, we recall that the edge $(x, z)$ occurs in $\mathcal{G}'$ and that $z$ occurs after $x$ in any trace where $x$ occurs. That is, there is an index $j' \in \{i+1, ..., len(t)\}$ such that $z = t[j']$ where $j' > i$ holds. Hence, $\mathcal{G}'$ satisfies the constraint. Contradiction.

Therefore, $\mathcal{G}' \models \pi(L)$ holds. □

EXAMPLE 5.9. Consider the trace $abcde$ and the constraints $\neg(\{c\} \rightsquigarrow \{d\})$ and $\neg(\{b\} \rightsquigarrow \{d\})$. In this setting, for the dependency graph $\texttt{PG}(\{abcde\}, \emptyset)$, the edges $(c, d)$ and $(b, c)$ are fake ones. The left part of Figure 6 evidences how the graph has to be updated when removing the edge $(c, d)$—in fact, this coincides with the graph built when the edge $(c, d)$ is forbidden, as we already discussed in Example 5.1.

Note that, in the resulting graph, $(b, c)$ is no longer a fake edge. However, the resulting graph does not still satisfy the constraints, because $(b, d)$ has became a fake edge. On the right part of Figure 6, a further update is reported accommodating the deletion of the edge $(b, d)$. Note that the connectivity is now guaranteed via a direct connection from $a$ to $d$. $\lhd$

The specific strategy adopted to select causal predecessors and causal successors is formalized in the steps 3–7. Eventually we return the causal net built on top of $\mathcal{G}_k$ via the function `computeBindings`. The correctness of the whole approach is shown below.

THEOREM 5.10. *The following properties hold on* COMPUTE-CN*, receiving as input $\mathcal{A}$, $L$, $\Pi \in \mathcal{C}[\{\not\rightsquigarrow\}]$, the parameter $\delta$, and for $\tau = 1$:*

— *if it returns 'no', then there is no solution;*
— *if it returns $\langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ and $L$ is (resp., is not) a linear log, then $\langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ is a (resp., possibly extended) causal net such that $\langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle \vdash L$ and $\mathcal{G} \models \Pi$.*

PROOF. Because of Lemma 5.6, we know that $\mathcal{G}_1$ is a dependency graph over $\mathcal{A}(L)$ and $\mathcal{G}_1 \models \pi(L)$. Consider then all the update operations performed in the steps 3–7. Because of Lemma 5.7 these operations are well-defined, and by Lemma 5.8, the graph $\mathcal{G}_k$ is still a dependency graph with $\mathcal{G}_k \models \pi(L)$.

Let us now focus on step 8. Note that when all fake edges are removed, the only edges that remain and that can violate a negated path constraints have the form $(a_\perp, a)$, $(a, a_\top)$, or $(a_\perp, a_\top)$. However, if there is a constraint preventing the existence of a path from $a_\perp$ to $a$, or from $a$ to $a_\top$, or from $a_\perp$ to $a_\top$, then there can be no dependency graph at all satisfying them. Hence, if the algorithm halts there, then we are guaranteed that no solution exists. Otherwise, for the analysis of the last step, recall that the function `computeBindings` is a constructive implementation of the proof of Theorem 3.11, in order to build a possibly extended causal net from the given graph $\mathcal{G}_k$. In particular, it is immediate to check that whenever $L$ is linear, we end up with a causal net. $\square$

*Implementation issues and computation time analysis.* In addition to the symbols already used when analyzing the costs of the algorithm of Figure 7, let us denote by $n^{\not\rightsquigarrow}$ the number of constraints of type $\mathcal{C}[\{\not\rightsquigarrow\}]$ that are taken as input, and by $m^{\not\rightsquigarrow}$ the number of (distinct) paths prohibited by them, i.e., $m^{\not\rightsquigarrow} = |\{(x, y) \in \mathcal{A} \times \mathcal{A} \mid \exists\, \neg(S \not\rightsquigarrow T) \text{ s. t. } x \in S \text{ and } y \in T\}|$. Notice that typically $m^{\not\rightsquigarrow} << k^2$, where $k$ still denotes the maximum number of elements appearing in either side of any constraint.

As above, constraints are indexed with numbers in $\{1, \ldots, n_c\}$ and associated with two lists of activity identifiers, storing the activities in their left and right sides, respectively, while $n_a \times n_a$ matrices are used to store causal scores, dependency graphs, as well as the collection of prohibited paths mentioned above. An additional vector of flags is kept to indicate whether each constraint is currently satisfied or not.

Finally, in order to speed up the calculation of $\texttt{pred}(\mathcal{G}_k, x)$ and $\texttt{succ}(\mathcal{G}_k, x)$, for each activity $x$, we precompute a relaxed version of both sets, denoted by $\texttt{pred}(x)$ and $\texttt{succ}(x)$, only accounting for the ordering of activities in the log traces. More precisely, for any activity $y$, it is $y \in \texttt{succ}(x)$ (resp., $y \in \texttt{pred}(x)$) iff for each trace $t \in L$ where $x$ occurs, then $y$ also occurs in $t$ after (resp., before) $x$. All such sets of (potential) successors and predecessors are stored as boolean vectors, one for each activity.

Initializing all these data structures and performing the first two steps in the algorithm takes $O(n_t \times l_t^2 + n^{\not\rightarrow} \times k + n_a^2)$ time.

The loop spanning over steps 3-7 can be iterated $m^{\not\rightarrow}$ times at most, seeing as each iteration removes at least one of the fake edges and one of the paths violating some constraint — which are, at most, $n_a^2$ and $m^{\not\rightarrow}$, respectively (with $m^{\not\rightarrow} \leq k^2 \leq n_a^2$).

The computation of all fake edges, at the beginning of each iteration, is accomplished in $O(n^{\not\rightarrow} \times n_a^2)$ time as follows. For each constraint that is still unsatisfied, two symmetric multiple-source visits of $\mathcal{G}_k$ are carried out, starting from all the activities in its left and right sides, respectively; in particular, in the latter case, edges are considered as they were reversed. By reckoning all edges traversed in both directions as fake edges, we compute that with the minimum causal score — denoted by $(x^*, y^*)$ in the figure.

We then compute the transitive closure $\mathcal{G}_k^+$ of the current dependency graph via a matrix-multiplication method, in $O(n_a^\omega)$ time, and materialize it into a matrix. In our current implementation, based on the famous Strassen's method, it is $\omega = 2.8074$. However, this cost can be lowered, since it was proven that $\omega \leq 2.374$ [**?**]. Anyway, answering path queries against $\mathcal{G}_k^+$, in $O(n_a)$ time we can find the nodes $z$ and $w$ mentioned in Step 5. To this end, we simply select the activity in $\texttt{succ}(x^*)$ (resp., in $\texttt{pred}(y^*)$) with the maximal score $\mathbf{cs}_\delta(x^*, z)$ (resp., $\mathbf{cs}_\delta(w, y^*)$), among those satisfying all involved path constraints — including the existence of a path from $z$ to $a_\top$ (resp., from $a_\bot$ to $w$), which is a required property of causal successors (resp., predecessors).

Therefore, the overall cost of the loop in Figure 8 is $O((n_a^\omega + n_a^2 \times n^{\not\rightarrow}) \times m^{\not\rightarrow})$.

A cost of $O(n_a)$ is enough for accomplishing all remaining (edge update) operations in the loop. The same result holds for Step 8, where we just need to check whether all constraints are marked as satisfied. As explained previously, a $O(n_t \times l_t \times n_a)$ cost is needed for computing all bindings (while storing them in a concise form).

In conclusion, we get a total cost of $O(n_t \times l_t \times \max(l_t, n_a) + (n_a^\omega + n_a^2 \times n^{\not\rightarrow}) \times m^{\not\rightarrow})$.[6]

## 6. EXPERIMENTAL EVALUATION

The algorithms proposed in the paper have been implemented as a plug-in for the well-known process mining suite *ProM* [van Dongen et al. 2005], combining the computation schemes described in Figure 7 and Figure 8. The latter scheme is used when user-defined constraints belong to the class $\mathcal{C}[\{\not\rightarrow\}]$, whereas the former has been slightly generalized[7] in order to provide a heuristic solution approach in all remaining cases (while still being an exact solution approach over the class $\mathcal{C}[\{\rightarrow, \not\rightarrow, \rightsquigarrow\}]$).

The generalization consists of a post-processing procedure applied to the discovered causal net returned by Step 19 in Figure 7. The procedure removes "useless" edges with the intended goal of returning a more compact model, by increasing the chances of satisfying negated path constraints. Formally, it iteratively removes the edge with the lowest causal score over all the edges $(x, y)$ satisfying the following three conditions:

*(i)* none of the input (resp., output) bindings associated with $y$ (resp., $x$) coincides with $\{(x, y)\}$, i.e., the edge does not appear as a singleton binding;

*(ii)* the removal of $(x, y)$ will not violate any user-defined precedence constraint;

*(iii)* $\frac{\mathbf{cs}_\delta(x,y)}{\min\{\mathbf{cs}_\delta(x,y^*), \mathbf{cs}_\delta(x^*,y)\}} > \tau_{r2b}$, where $(x, y^*)$ (resp., $(x^*, y)$) is the outgoing edge of $x$ (resp., the incoming edge of $y$) with the highest causal score. Notice that $\tau_{r2b}$ is used

[6] This result could be improved by resorting to dynamic graph algorithms for keeping updated the transitive closure of a graph [**?**]. In particular, the overall time of the algorithm in Figure 8 would lower to $O(n_t \times l_t \times \max(l_t, n_a) + n_a^{2.374} + n_a^2 \times n^{\not\rightarrow} \times m^{\not\rightarrow})$, if using the solution in [**?**], which guarantees constant look-up times and $O(n^2)$ worst-case time for each update and $O(n_a^{2.374})$ for initializing its data structures.

[7] We also implemented a generalization of the algorithm in Figure 8, but results of experimentation evidenced that its efficacy as a heuristic was not satisfying.

Table II. Process discovery algorithms used in the experiments: legend of symbols.

| Symbol | Meaning |
|---|---|
| ILP | The ILP-based mining algorithm defined in [van der Werf et al. 2009] |
| AGNEs | The $AGNEs$ mining algorithm in [Goedertier et al. 2009] |
| $\alpha$ | The $\alpha$ mining algorithm defined in [van der Aalst et al. 2004] |
| HM | The *Heuristics* mining algorithm defined in [A.J.M.M. Weijters et al. 2006] |
| GM | The *Genetics* mining algorithm defined in [Medeiros et al. 2007] |
| Here | The COMPUTE-CN algorithm defined in Figures 7 and 8 |

here as a relative (lower) threshold to check the strength of any edge $(x, y)$, relatively to those of the best $y$'s predecessor and of the best $x$'s successor—it is similar to the "relative to best" threshold proposed in [A.J.M.M. Weijters et al. 2006]. In the tests described in the remainder of the paper, we fixed the values $\delta = 0.85$, $\tau = 0.05$, and $\tau_{r2b} = 0.5$. These values were chosen pragmatically based on a series of tests conducted on a wide range of synthesized data. Now, they are hardwired in the implementation and are transparent to the user of the plug-in.

The performances of our implementation have been compared with those of the approaches listed in Table II. Note that we considered the approaches proposed by van der Werf et al. [2009] (ILP) and Goedertier et al. [2009] (AGNEs), which we have already discussed in Section 1.2 and which can in fact incorporate a-priori knowledge on the existence of activity dependencies. Moreover, we considered four classical discovery approaches, very popular in the Process Mining community: "Heuristics Miner" [A.J.M.M. Weijters et al. 2006] (HM), "Genetic Miner" [Medeiros et al. 2007] (GM), algorithm $\alpha$ [van der Aalst et al. 2004].The performances of these methods are discussed to delineate baselines suitable for assessing the gain that can be obtained by empowering learning methods with the capability to exploit knowledge on the real structure of the process under analysis.

In order to test the competitors, we exploited their implementations available in the latest (6.3) release of the *ProM* framework [van Dongen et al. 2005], but for AGNEs and ILP for which the implementations in the version 5.2 were used. Indeed, AGNEs is only available up to this earlier version, while the implementation of ILP in the latest release lacks of the ability of improving the models discovered by expressing parallelism and direct dependency relationships (which is the feature we are interested in). Since an earlier implementation of $\alpha$ in ProM also allows users to express relationships of parallelism and direct dependencies, we have included this method too in all the tests performed in presence of background knowledge. Default settings were used for the various methods. For GM, we took the best model among 1000 different ones derived from an initial population of 100 models.

In the rest of the section, we shall illustrate results of experimental activity conducted over our method and its competitors. In particular, in Section 6.1 we consider an archetypical application scenario, in Section 6.2 we illustrate results on benchmark logs, and in Section 6.3 we present results on synthesized logs. Experiments have been performed on a dedicated machine, equipped with an Intel dual-core 3 GHz processor with 4 GB (DDR2 1033 MHz) of RAM, and running Windows 7 Professional.

### 6.1. Case Study: a Product-Recall Process

Let us consider the product recall process defined by M.T. Wynn et al. [2009], in accordance to the guidelines established by several public institutions (e.g., in Australia, New Zealand, USA, and EU). This is an archetypical application scenario, and its discussion is meant here to assess the importance of having background knowledge when an incomplete sample of the possible traces is only given at hand.

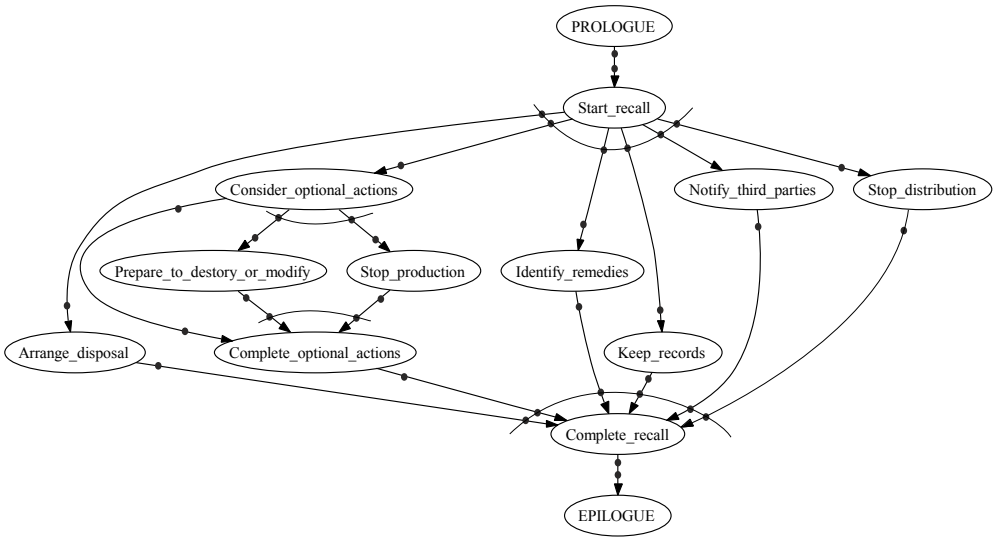



Fig. 9. Causal net of the *ProductRecall* process.

*6.1.1. Testbed Description.* The process concerns the main activities that must be performed by a recall sponsor (usually the manufacturer of a suspect product), in response to a recall incident, which can be possibly triggered by consumer complaints, supplier notifications, or failed quality tests. Specifically, the reported problem has to be investigated and a comprehensive risk analysis must be done (macro-activity `PROLOGUE`—see [M.T. Wynn et al. 2009], for details on the sequence of activities comprised in it), in order to decide whether the product should be recalled or not. The model associated with the activities occurring in the former case is reported in Figure 9: After starting the recall procedure, a case can proceed along a number of concurrent threads, including the following tasks: (i) stopping the distribution of the product, (ii) identifying remedies, (iii) arranging the disposal of items already distributed, (iv) keeping records for subsequent monitoring and analysis purposes, and (v) notifying third parties about the recall. In addition, depending on the kinds of product and of defect involved, it can be necessary to halt the production of the product and to destroy/modify other products that might have been contaminated. Once these recall actions have been completed, a sequence of finalizing activities must be performed (macro-activity `EPILOGUE`), ranging from monitoring the effectiveness of the process, to implementing changes to prevent similar problems in the future, to preparing reports for regulatory authorities and/or other third parties. Notice that, as pointed out by M.T. Wynn et al. [2009], the need of handling product recall operations, while taking care of traceability and notification issues, arises in a wide variety of real applications.

*6.1.2. Evaluation Setting.* In order to valuate the quality of findings, we contrast the set $D_{out}$ of causal dependencies discovered by the mining methods towards the set $D_{in}$ of real dependencies existing in the a-priori known process model, by resorting to the classical *F-measure* metric, defined as $\frac{2\times P\times R}{P+R}$, where $P$ (stating for *precision*) is the fraction of the dependencies in the mined model that exist in the real model, i.e., $P = |D_{out} \cap D_{in}|/|D_{out}|$, whereas $R$ (standing for *Recall*) is the fraction of real dependencies captured by the mined model, i.e., $R = |D_{out} \cap D_{in}|/|D_{in}|$.

Table III. F-measure scores obtained by algorithm COMPUTE-CN and its competitors, for different percentages of traces of process *ProductRecall* (Figure 9), without (left) and with (right) a-priori knowledge on parallelism relationships. For both cases, maximal scores on each trace percentage are written in bold.

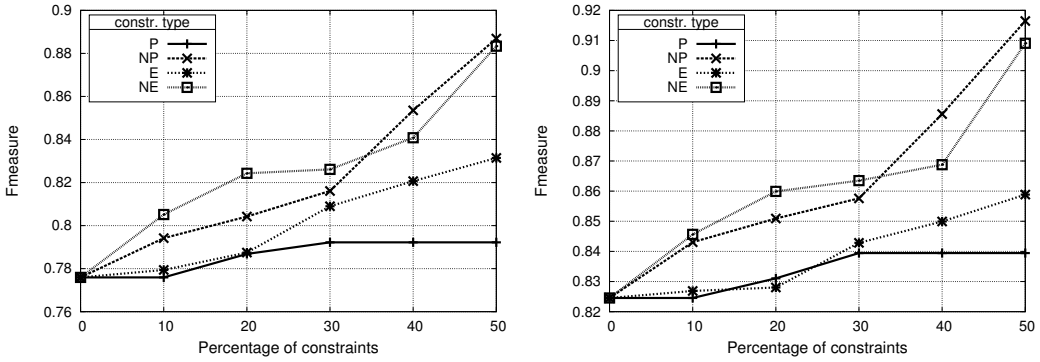
| *trace* | **without** constraints | | | | | | **with** constraints | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| % | HM | $\alpha$ | GM | ILP | AGNEs | Here | $\alpha$ | ILP | AGNEs | Here |
| 10 | 0.657 | 0.717 | 0.281 | 0.848 | 0.674 | **0.930** | 0.772 | 0.848 | 0.667 | **0.956** |
| 20 | 0.830 | 0.871 | 0.432 | 0.924 | 0.727 | **0.982** | 0.899 | 0.924 | 0.736 | **0.992** |
| 30 | 0.893 | 0.924 | 0.391 | 0.914 | 0.677 | **0.982** | 0.950 | 0.914 | 0.720 | **1.000** |
| 40 | 0.931 | 0.950 | 0.348 | 0.914 | 0.720 | **0.992** | 0.983 | 0.914 | 0.730 | **1.000** |
| 50 | 0.965 | 0.951 | 0.354 | 0.904 | 0.748 | **1.000** | 0.968 | 0.904 | 0.727 | **1.000** |
| 60 | 0.979 | 0.975 | 0.417 | 0.903 | 0.745 | **1.000** | 0.970 | 0.903 | 0.774 | **1.000** |
| 70 | 0.984 | 0.984 | 0.556 | 0.882 | 0.774 | **1.000** | 0.990 | 0.882 | 0.763 | **1.000** |
| 80 | 0.984 | 0.992 | 0.500 | 0.893 | 0.763 | **1.000** | 0.992 | 0.893 | 0.779 | **1.000** |
| 90 | **1.000** | **1.000** | 0.510 | 0.882 | 0.763 | **1.000** | **1.000** | 0.882 | 0.779 | **1.000** |
| 100 | **1.000** | **1.000** | 0.605 | 0.882 | 0.763 | **1.000** | **1.000** | 0.882 | 0.782 | **1.000** |
| *Avg* | 0.911 | 0.929 | 0.439 | 0.897 | 0.735 | **0.986** | 0.952 | 0.897 | 0.752 | **0.995** |

*6.1.3. Test with variable amounts of log traces.* Given the interest in assessing the performances of our approach in scenarios where log completeness does not hold, we first built a *complete* log $L$ for the process where each possible behavior is registered once. Then, we conducted experiments over logs that are obtained from $L$ by randomly picking $x\%$ of its traces, for $x \in \{10, 20, ..., 100\}$. In particular, 10 different logs are sampled for each $x$, and experiments are performed on each of them, so that average values will be discussed in our analysis. Table III summarizes our findings.

Let us first consider the columns reporting results for the scenario where no constraints are provided as input. It is easily seen that the performances of all methods are rather poor against very small log samples, and tend to improve when augmenting the number of input traces. Such an effect is evident in the case of the classical methods HM, $\alpha$, which get full accuracy when provided with at least $90\%$ of the log traces. Interestingly, our approach managed to reconstruct the real set of task dependencies already with $50\%$ samples of the log, hence outperforming the competitors even without being provided as input with additional background knowledge. On the one hand, this behavior is clearly due by the fact that our causal score heuristic, designed to select the edges to be included/removed from the resulting model (see Section 5), is indeed largely inspired by such earlier approaches. On the other hand, it confirms the validity of the processing scheme of Figure 7 and Figure 8 and the efficacy of the very simple post-processing method illustrated in this section. Opposed to these good news related to our method, it emerges that low quality scores are achieved when using ILP and AGNEs, which are to be seen as our more direct competitors.

The four rightmost columns in Table III report the results obtained by providing domain knowledge to the discovery methods. To this end, we assumed that the analyst knows that the activity Complete_optional_actions is parallel with each of the activities in {Identify_remedies, Keep_records, Stop_distribution, Arrange_disposal, Notify_third_parties}. Note that this knowledge can be incorporated in the two competitors AGNEs, and ILP, as well as in the $\alpha$ algorithm[8], so that we are not exploiting for the moment the richer expressiveness of our framework. By looking at the table, it emerges that our method is able of fully exploiting these constraints, by getting impressive accuracy results even on very small samples. Instead, the benefits of the background knowledge are lower for $\alpha$ and especially for AGNEs. Moreover, note that both ILP and $\alpha$ are not capable to exploit at all the given knowledge. In fact, in our experiments, we have noticed that the former method is quite often not capable to ben-

[8]To this purpose, in the latter two cases, all the above pairs of concurrent activities were added to the parallelism relation, while removing any direct succession relationship between them, derived from the log.




(a) F-measure scores on *3%* log samples (b) F-measure scores on *6%* log samples

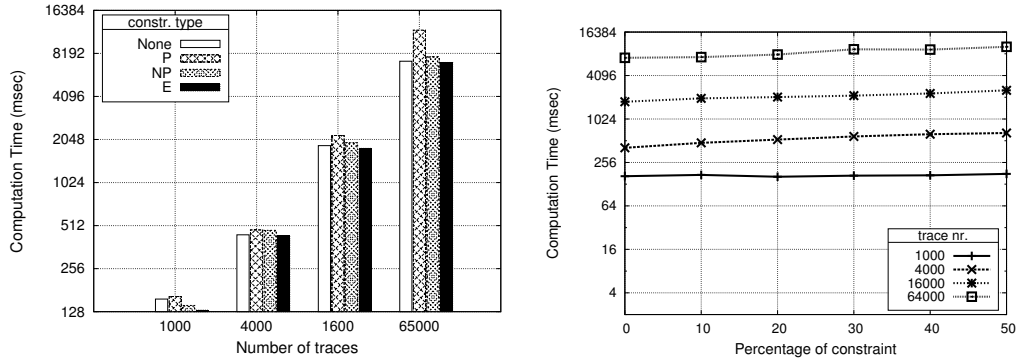
Fig. 10. F-measure scores obtained by COMPUTE-CN when varying the percentages of positive edge/path constraints and of negative edge/path constraints, for four different families of log samples (all of 16K traces), corresponding to $3\%$ (a) and $6\%$ (b) of distinct traces in the original log, respectively).

efit of knowledge about parallelism, while significant improvements can be obtained in presence of knowledge about relationships of precedences, as we shall see later.

*6.1.4. Varying the quantity and type of background knowledge.* In order to assess, in a deeper and more systematic manner, the capability of our approach to exploit a-priori knowledge for improving its performances, further tests were carried out on the same application scenario, while using different amounts of precedence constraints. As a way of simulating a non-trivial discovery setting, only very small portions of the (complete) log mentioned above were used to this end, containing $3\%$, and $6\%$, of the traces. Again, for each trace percentage $x\%$ (with $x \in \{3, 6\}$), 10 samples were generated, by randomly picking $x\%$ of the traces in the log.

Different inter-activity dependencies were extracted directly from the known control-flow model of the process, shown in Figure 9. Regarding this model as a dependency graph, four binary relations over its activities can be defined and computed trivially, each encoding some basic kind of (singleton-body) precedence constraints: edges and paths (i.e., pairs of activities, where the second one depends on the first either directly or indirectly, respectively), and the associated complementary (w.r.t. all possible activity pairs) relations of negative edges and negative paths. These relationships are denoted in the figures discussed below by E, P, NE, NP, respectively. In order to automatically generate different sets of precedence constraints, while controlling the relative amount of each type of them, a sample of elements is extracted randomly from each of the core pairwise relations above, by using some given percentage value for each of them. In fact, 5 samples are generated and results are averaged over them.

Figure 10 reports the average F-measure scores obtained by our solution approach against the different percentages of log traces, while using one of the above described kinds of pairwise constraints per time. For each kind of constraints, different amounts were considered, ranging from $0$ to $50\%$ of its whole population. It clearly emerges that the use of the background knowledge improves the performances of the algorithm. Indeed, higher F-measure scores are achieved when increasing the amount of whichever kind of constraints. However, a noticeably boost seems to be given to the accuracy when increasing the percentage of negative (edge or path) constraints, no matter of how many traces are taken as input. Conversely, lower improvements are obtained when only positive constraints are used, especially when these are expressed on paths. Intuitively, this is due to the fact that negative constraints are able to reduce the num-




(a) Computation Time vs Log Size (b) Computation Time vs Constraints' Amounts

Fig. 11. Computation time spent by algorithm COMPUTE-CN with different amounts of traces and constraints' percentages in input. A base-2 logarithmic scale is used for the vertical axis in both figures, as well as for the horizontal axis in the left-hand figure.

ber of spurious flows of executions whose presence negatively impacts on the quality of the resulting process model. This finding has been confirmed in all our tests. Hence, as a practical guideline, the users of our plug-in are encouraged to introduce as much as possible negated constraints in the specification of the mining problem.

Finally, note that Figure 10 evidences that the level of improvement is neatly higher when working on smaller log samples (hardly capturing all actual process behaviors). In fact, this is hardly surprising given that our method has very good performances even when no background knowledge is provided at all.

*6.1.5. Rate of unsatisfied constraints.* So far our algorithm has been tested in scenarios where the kinds of precedence constraints are not mixed together. Therefore, according to the results discussed in Section 5, our algorithm provides an exact solution in these cases, i.e., no constraint can be violated by the resulting process model. Hence, in order to study the efficacy of the method as a heuristic, we performed an additional series of experiments with heterogeneous combinations of precedence constraints, mixing up negative path constraints with other kinds constraints. To this end, we applied our algorithm to the same log samples built as in the previous subsection, while providing it with variable amounts of negative path constraints, and fixing the percentage of any other kind of constraints to $25\%$. Note that, given the adaptation discussed at the beginning of this section, the algorithm results a process model that still satisfies all positive edge constraints, path constraints, and negated edge constraints. However, the satisfaction of negated path constraint is just greedily enforced by the post-processing phase, and it is therefore not guaranteed. In fact, we computed the rate of negative-path constraints left unsatisfied by our approach, in correspondence of different amounts of distinct traces and of negative-path constraints provided as input (expressed as percentages w.r.t. the size of their respective populations). In the worst case, just $0.8\%$ of all negative-path constraints given as input have not been satisfied, in the average. This rate shrinks of about a half when using either $10\%$ samples or $50\%$ samples of negative paths. A similar trend is observed for trace samples with a $6\%$ of distinct process traces. The amount of violated constraints becomes negligible when bigger samples of log traces (we tested both $12\%$ or $24\%$), no matter how many negated paths are passed to the algorithm.

*6.1.6. Running Time.* In order to provide the reader with a general idea of the computation times needed by our method, we performed a first series of tests on logs of different sizes. Logs were generated again with a variable level of completeness, by extracting different amounts of distinct traces out of the original (fully complete) log. First of all, for each trace percentage $x \in \{3, 6, 12, 24\}$, we randomly extracted 4 samples of (distinct) traces out of the complete log, which were subsequently used as a sort of "seeds" in order to produce a behaviorally heterogenous collection of logs. Specifically, for each log size $s \in \{1000, 4000, 16000, 64000\}$, we simply duplicated the distinct traces in each of such seed in a balanced way (i.e., all traces are replicated approximatively the same number of times), up to obtaining a collection of $s$ traces.[9] Concerning the constraints, we considered 6 distinct configurations, each corresponding to a distinct percentage $p \in \{10, 20, ..., 50\}$ for all types of pairwise constraints (i.e., E, P, NE, NP) extracted by the model in Figure 9, as explained before. For each value of $p$, 5 different heterogeneous sets of constraints were generated, containing $p\%$ constraints for each constraint type, extracted at random from their respective populations. Again, 5 different trials were performed for each of the above logs and constraints' sets.

Figure 11 reports the computation time (measured in milliseconds) when varying the amounts of traces and constraints in input. Results are presented there according to two different perspectives, namely, a curve for each constraints' percentage with the different log sizes over the horizontal axis (left) and vice-versa (right). Each value plotted in either figure has been computed by averaging the durations of all the runs performed with a specific number $s$ of traces and a particular percentage $p\%$ of constraints (for $s \in \{1000, 4000, 16000, 64000\}$ and $p \in \{10, 20, ...50\}$). Notably, whatever percentage of constraints is given as input, the computation time scales basically linearly with respect to the number of input traces. On the other hand, a positive correlation seems to exist as well between the overall running time and the quantity of constraints taken as input—if ignoring the case of 1000-sized logs, where the times measured are too low to safely infer any general significant trend of behavior. Anyway, the impact of constraints on times is negligible if compared with that of the log size.

Finally, we notice that the computation time of algorithm COMPUTE-CN is comparable to that of the standard process mining methods $\alpha$, ILP, and HM, and neatly lower than those spent by GM, AGNEs. Indeed, the latter method took about 600 times longer than COMPUTE-CN to compute a model, in the average, while the computation time of GM was about 4500 times that of COMPUTE-CN when using the default population size of 1000 (the ratio only decreased to 600:1 with 100-model populations).

### 6.2. Comparative Analysis on Benchmark Data

In order to assess the capability of our approach to discover a *good-quality* process model in a wider range of settings, we performed a series of tests on some benchmark logs, while measuring the accuracy of the each model by way of several "log-conformance" metrics, very popular in the fields of Process Mining and Business Process Analysis. Differently from the pure (edge-oriented) F-measure employed in the previous section, to contrast a discovered model to the true (a-priori known) one, these metrics allow for evaluating how much the behaviors registered in a given log comply with those allowed by the model under analysis.

*6.2.1. Testbed: Logs and Conformance Metrics.* Our experimental activities were carried out over some of the benchmark logs provided with the ProM framework [van Dongen

[9]Of course, duplicated traces might be processed more efficiently by just "weighting" each trace with the number of its occurrences in the log. Here, we avoid this trick as it is our goal to precisely stress the algorithm at the varying of the log size, in a setting where the given (very simple) process model does not allow for a sufficiently large number of distinct behaviors.

Table IV. Benchmark logs: structural characteristics and statistics.

| log name | distinct activities | pairs of \|\| activities | control-flow constructs | distinct traces | total traces |
|---|---|---|---|---|---|
| $parallel5$ | 10 | 10 | – | 109 | 300 |
| $a10skip$ | 12 | 1 | $skip$ | 6 | 300 |
| $a12$ | 14 | 2 | – | 5 | 300 |
| $a5$ | 7 | 1 | $loop$ | 13 | 300 |
| $a6nfc$ | 8 | 1 | $nf\text{-}choice$ | 3 | 300 |
| $a7$ | 9 | 4 | – | 14 | 300 |
| $a8$ | 10 | 1 | – | 4 | 300 |
| $choice$ | 12 | 0 | – | 16 | 300 |
| $driversLicense$ | 9 | 0 | – | 2 | 300 |
| $herbstFig3p4$ | 12 | 3 | $loop$ | 12 | 300 |
| $herbstFig6p18$ | 7 | 0 | $loop$ | 153 | 300 |
| $herbstFig6p36$ | 12 | 0 | $nf\text{-}choice$ | 2 | 300 |
| $herbstFig6p37$ | 16 | 36 | – | 135 | 300 |
| $herbstFig6p41$ | 16 | 4 | – | 12 | 300 |
| $herbstFig6p45$ | 8 | 4 | – | 12 | 300 |
| $l2l$ | 6 | 0 | $loop$ | 10 | 300 |
| $l2lOptional$ | 6 | 0 | $loop, skip$ | 9 | 300 |
| $l2lSkip$ | 6 | 0 | $loop$ | 8 | 300 |

et al. 2005], which have been widely used in the literature (see, e.g., [Medeiros et al. 2007; Goedertier et al. ; De Weerdt et al. 2011; Weerdt et al. 2012]) in order to evaluate process mining approaches. In particular, owing to our special interest toward incompleteness issues, we focused on the subset of those logs exhibiting the highest degree of non-determinism and concurrency. The logs include special routing constructs, such as (non free) choices, skips, and loops. Table IV summarizes their features, by reporting in particular the number of distinct activities composing the process, the number of pairs of activities belonging to different parallel branches ("parallel || pairs"), the presence of special constructs, the number of distinct activity sequences occurring in the log, and the total number of log traces.

In order to assess the capability of a process model to accurately capture the behavior recorded in a given log, several alternative *conformance* metrics have been proposed in the literature. In our analysis, we consider *precision* metrics and *recall* metrics. [10] Precision metrics attempt to estimate the amount of the "extra" (unseen and likely unwanted) behavior allowed by the model, with respect to that actually registered in the log, whereas recall metrics try to evaluate how much of the behavior recorded in a log is really captured by the model. All these metrics range over the real interval $[0, 1]$ and have been defined in the literature for Petri-net models. In order to use them with a model represented in another language (in particular, a causal net), we preliminary translated the given model into a Petri net, with the help of suitable conversion plug-ins available in the ProM framework. The actual computation of the metrics was carried out by taking advantage of the *CoBeFra* tool, recently proposed [van den Broucke et al. 2013] as a practical support to conformance analysis. A summary of all the considered metrics is reported in the first column of Table V.

*6.2.2. Results.* For each of the conformance metrics described above, Table V reports the average value obtained by applying each of the methods to each of the logs in Table IV. As a term of comparison, column $True$ reports the conformance results obtained for the process models that were actually used to generate the logs (and that are known for the given benchmark logs). Since many methods got very similar results over several metrics, a statistical testing procedure was carried out to check whether their

[10] We did not considered generalization [van der Aalst et al. a] metrics, which are meant to punish overly precise process models, which tend to prevent any likely and not explicitly forbidden behavior, if it does not occur explicitly in the log. This choice stems from the fact that, in our setting, the selected logs can be assumed to capture well enough all behavioral aspects of the respective processes.

Table V. Average conformance measures obtained, on all benchmark logs, by different workflow discovery methods — including the one proposed in this paper ($Here$). An additional column ($True$) reports the average measures computed on ground-truth process models (i.e., the one employed to generate the log). The results of the methods that were reckoned as significantly different from the best performing one (for each measure) are reported in italics.

| Metric | $True$ | AGNEs | $\alpha$ | GM | HM | ILP | Here |
|---|---|---|---|---|---|---|---|
| Fitness [Rozinat and van der Aalst 2008] | 1.000 | 0.995 | 0.988 | 1.000 | 0.995 | 1.000 | 0.997 |
| Alignment Based Fitness [Adriansyah et al. 2011] | 1.000 | 0.986 | 0.848 | 0.997 | 0.995 | 0.889 | 0.998 |
| Behavioral Recall [Goedertier et al. ] | 0.997 | 0.992 | *0.959* | 0.989 | *0.843* | 1.000 | 0.989 |
| Proper Completion [Rozinat and van der Aalst 2008] | 1.000 | 0.938 | *0.851* | 0.999 | 0.934 | 1.000 | 0.957 |
| Adv. Behav. Appropriateness [Rozinat and van der Aalst 2008] | *0.797* | *0.823* | 0.854 | *0.802* | *0.783* | 0.856 | *0.805* |
| Alignment Based Precision [van der Aalst et al. b] | 0.925 | 0.943 | 0.910 | 0.925 | 0.920 | 0.904 | 0.930 |
| Behavioral Specificity [Goedertier et al. ] | 1.000 | 0.992 | 0.978 | 0.994 | 0.991 | 1.000 | 0.994 |
| (Wtd) Behavioral Precision [De Weerdt et al. 2011] | 0.907 | 0.884 | 0.882 | 0.890 | *0.749* | 0.891 | 0.898 |
| Negative Event Precision [Goedertier et al. ] | 0.968 | 0.927 | 0.927 | 0.953 | 0.941 | 0.934 | 0.944 |

Table VI. Statistics on the "critical" sub-log extracted for each of the benchmark logs in Table IV.

| original benchmark log | distinct traces | (%) | total traces | (%) |
|---|---|---|---|---|
| herbstFig3p4 | 4 | (12%) | 48 | (16%) |
| herbstFig6p37 | 13 | (10%) | 135 | (45%) |
| herbstFig6p41 | 4 | (35%) | 68 | (23%) |
| herbstFig6p45 | 3 | (25%) | 53 | (18%) |
| parallel5 | 5 | (5%) | 9 | (3%) |

behaviors are really different. To this end, for each of the considered metrics, a paired two-tail Student's test was applied to compare the outcomes of each method with those of the best one (i.e., the one achieving the highest average value on that metrics). Notably, for each the metrics, we did not find significantly enough differences, apart from a small number of cases, which are emphasized in italics in the table—in almost all cases we could not reject (with a $95\%$ level of confidence) the null hypothesis that a method behaves identically to the best performer.

The fact that almost all existing approaches are able to reconstruct the originating models with high levels of precision and recall (for most of the considered metrics) comes with no surprise, seeing as the logs are representative enough of all the possible behaviors of their respective processes. To our ends, we observe that our method is competitive even in this standard setting where logs are basically complete. Moreover, we observe that our evaluation procedure tends in any case to disadvantage a model that was not originally built as a Petri net, because the conversion plug-ins often produce results with hidden/duplicated transitions, which are considered as a source of extra-log behaviors by certain precision metrics.[11]

As a further comment on these tests, we stress here that, in terms of computation time, our approach is in line with the most efficient process discovery techniques, and definitely faster than both AGNEs and GM. In details, the average and standard deviations values computed over the running times (in milliseconds) of each method are: $21851.2 \pm 12334.6$ for AGNEs, $341012.2 \pm 385997.7$ for GM, $95.6 \pm 73.4$ for HM, $203.1 \pm 53.4$ for ILP, and $138.2 \pm 85.3$ for our approach.

*6.2.3. Tests on "critical" sublogs.* The quality results presented in the previous section clearly demonstrate that each of the analyzed state-of-the-art methods (as well as, comfortingly, our approach) is really effective in rediscovering the structure of a pro-

[11]This explains why even the collection of ground-truth models has been given lower precision scores than the native Petri-net models discovered with ILP or AGNEs when using the *Advanced Behavioral Appropriateness* and *Alignment Based Precision* metrics. On the other hand, even a low recall score was assigned to the a-priori models by the *Behavioral Recall* metrics.

Table VII. Results on"critical" samples without and with background knowledge. For each row, **the best average score** is in bold and underlined, while the results of all methods that were **not recognized as significantly different** from the best performer (for the same metrics and setting) are in bold.

| | **without** constraints | | | | | | **with** constraints | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Metric | AGNEs | $\alpha$ | GM | HM | ILP | *Here* | $\alpha$ | AGNEs | ILP | *Here* |
| Fitness | **0.909** | **0.852** | **0.950** | **0.882** | **0.941** | **0.951** | 0.865 | 0.958 | 0.969 | **1.000** |
| Behavioral Recall | **0.913** | **0.792** | **0.948** | **0.881** | **0.945** | **0.939** | 0.806 | 0.966 | 0.962 | **1.000** |
| Alignment Based Fitness | 0.867 | **0.744** | **0.948** | **0.834** | **0.945** | **0.948** | 0.799 | 0.919 | 0.962 | **1.000** |
| Proper Completion | 0.095 | 0.102 | **0.523** | 0.081 | **0.459** | **0.543** | 0.340 | 0.457 | 0.564 | **1.000** |
| Adv. Behav. Appropriateness | **0.667** | **0.711** | **0.901** | **0.580** | **0.828** | **0.910** | 0.788 | 0.723 | **0.848** | **0.928** |
| Alignment Based Precision | **0.976** | **0.957** | **0.966** | **0.956** | **0.963** | **0.965** | **0.967** | **0.976** | **0.966** | **0.972** |
| Behavioral Specificity | **0.913** | **0.792** | **0.948** | **0.881** | **0.945** | **0.949** | 0.806 | 0.966 | 0.962 | **1.000** |
| (Wtd) Behavioral Precision | **0.669** | **0.701** | **0.794** | **0.615** | **0.794** | **0.776** | 0.764 | 0.843 | **0.872** | **0.966** |
| Negative Event Precision | 0.708 | **0.746** | **0.827** | 0.658 | **0.833** | **0.822** | 0.815 | 0.872 | **0.897** | **0.976** |
| F-measure | 0.740 | 0.730 | **0.817** | 0.733 | **0.833** | **0.836** | 0.806 | 0.929 | 0.833 | **1.000** |

cess whenever they are provided with a rich enough sample of process traces, capable to encompass the diverse kinds of admitted behaviors. In order to test all discovery methods in a more challenging setting, we extracted a *critical* sample out of each log, constituting an incomplete collection of traces, covering a limited portion of the whole variety of the behaviors in the original log. An empirical iterative procedure was devised to this purpose, where increasing amounts of traces are randomly removed from a given benchmark log, until we register a loss of $15\%$ in the F-measure of the models discovered with all tested methods. In order to further emphasize the effect of log incompleteness, we focused our attention on a subset of the logs in Table IV, which allowed all the tested discovery methods to perfectly rediscover the associated model (i.e., and hence achieve a maximal value of 1 over both F-measure and all of the recall metrics). The size of each of these sub-logs is reported in the Table VI, in terms of the number of traces and of distinct activity sequences that appear in the sub-log.

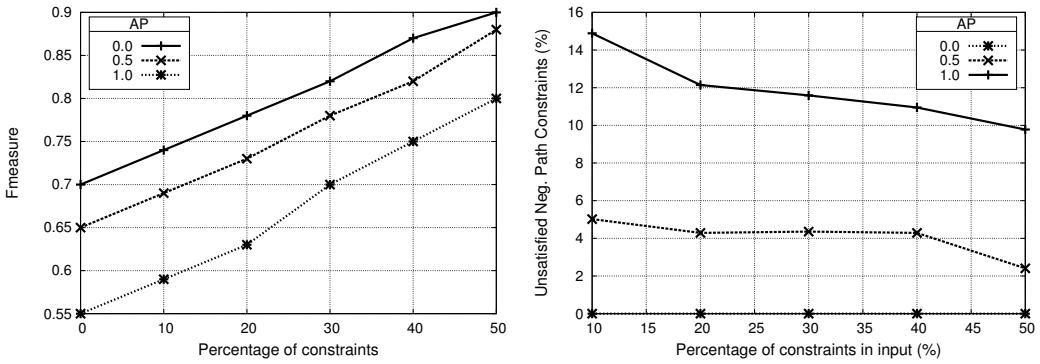
Given the incompleteness of the resulting logs, experiments have been conducted also in the presence of background knowledge. In this case, for each critical sub-log, we considered all possible combinations of three constraints at most, extracted from the given true models, by reporting the results obtained in the best scenario. Note that here we exploit the full expressiveness of our approach. In fact, it emerged that the best results are obtained in the presences of negative path constraints only.

Results for these tests are reported in Table VII, where the precision and recall metrics are computed by evaluating the quality of the models discovered from the sub-logs built as above w.r.t. the whole behavior of the traces registered in the original benchmark log. Moreover, the F-measure values are also reported (see Section 6.1.2). As above, a two-tail Student's test was carried out to assess the significance of the performance differences. The results obtained in absence of background knowledge further confirm the validity of our approach and, in particular, the effectiveness of its underlying (causal-score driven) heuristics for pruning useless/unlikely edges. The advantage of using additional background knowledge clearly emerges. In particular, our method perfectly reconstructs the true models of all benchmark logs (see the lowermost row), and achieves maximal recall without incurring into overgeneralization.

### 6.3. Further Tests on Synthesized Data

A final series of tests was conducted on different synthesized logs, in order to assess the effectiveness of the proposed approach against logs following different data distributions, as far as concerns the structure of the processes producing them.

*6.3.1. Generation of process models.* The data used in this experimentation were produced with the help of a random log generator (based upon the one described in [Burattin and Sperduti 2011]), which allows for both constructing a process model randomly,




(a) Accuracy w.r.t. true dependencies (b) Rate of unsatisfied neg. path constraints

Fig. 12. Results on synthesized log data, with different degrees of parallelism (AP): accuracy of discovered process models (in terms of F-measure w.r.t. real activity dependencies), and rates of unsatisfied (negative path) constraints. Both measures are reported for different amounts of a-priori constraints of all types (expressed as percentages w.r.t. the sizes of their respective population).

and generating a log of random execution traces out of it. The synthesis of the model is carried out via an incremental block-oriented procedure, where an initially empty model is extended iteratively by adding a new subprocess, until a given number of elementary activities is obtained. Each subprocess can be either an elementary activity, a sequence of activities, or a more complex control-flow block. The whole procedure is governed by a combination of gaussian and multinomial probability distributions, and it can be controlled by way of a number of parameters, which are listed below:

— *Task Number* (*TN*): mean of a gaussian distribution[12] determining the number of elementary activities to be generated;
— *Branching Factor* (*BF*): mean of a gaussian distribution determining the number of branches in any branching structure;
— *Singleton Probability* (*SP*): probability that any current subprocess will be instantiated with just one activity;
— *Fork Probability* (*FP*): probability that the subprocess is a "fork" structure (i.e., a number of subprocesses that can be executed in parallel or that are mutually exclusive), given that it is not an elementary activity;
— *AND Probability* (*AP*): probability that the branches of any fork structure are in parallel, rather than in mutual exclusion.

In order to prevent models from having an excessively unbalanced shape, one further parameter can be set, named *Maximum Nesting Depth* (*MND*), stating the maximum level of nesting of any activity with respect to all the control-flow blocks it is enclosed within—precisely, whenever the nesting level of a subprocess equals *MND*, the subprocess is forced to take the form of a single activity (i.e., it cannot give rise to a complex control-flow block). In our experiments, we fixed $SP = 0.1$, $FP = 0.5$, $TN = 25$, and $MND = 4$, while we simulated three different levels of concurrency, hinged on different values of parameter $AP \in \{0.0, 0.5, 1.0\}$. For each configuration, we generated 10 different schemas, and produced 10 logs for each of them, by simulating 1000 random enactments of the schema (as in [Burattin and Sperduti 2011]). In this way, 100 different logs were eventually obtained for each level of concurrency.

[12]The standard deviation of all gaussian distributions is set to a third of the mean.

*6.3.2. Results.* Figure 12 reports the results of tests performed by applying our approach, while providing it with heterogenous sets of precedence constraints, and hence expressing partial information on real activity relationships. As in Section 6.1.6, these constraints were randomly extracted from the model which generated the log, according to different levels of coverage for the entire population of constraints that can be derived from any model, expressed in terms of a percentage $p \in \{10, 20, ..., 50\}$. In more details, for each log $L$ and each value of $p$, we randomly generated 5 sets of constraints, by sampling $p\%$ pair from each of the binary relations (of the form E, P, NE, NP) extracted from the a-priori known model of $L$.

Each value in the figure was computed by averaging all the results obtained when using (i) any log of workflows generated with the give value of $AP$, and (ii) any constraint set covering $p\%$ of each (positive/negative direct/indirect) precedence relation. Clearly, higher levels of concurrency (i.e., higher values of $AP$) lead to lower accuracy results, as far as concerns the recognition of real activity dependencies. This result comes with no surprise, seeing as the relatively low number of random traces (namely 1000) available in each log hardly suffice to rediscover the structure of models with many parallel branches of activities. However, providing increasing amounts of constraints definitely helps to improve the average accuracy of the resulting models.

As a further effectiveness indicator (connected with the capability of our approach to really satisfy the constraints it was provided with), we finally present, in the right side of Figure 12, the rate of unsatisfied negative path constraints, i.e., the percentage of constraints of this kind that were taken as input in some run of the algorithm, but were not fulfilled by the causal net discovered in the same run.[13] Again, these results are reported for different values of both $p$ and $AP$. It is good news that the rates of unsatisfied negative path constraints are always relatively small, especially in the case of low concurrency levels. In fact, all the models that were induced from concurrency-free logs do not violate any constraint at all. Higher violation rates affects the models extracted from choice-free logs ($AP = 1$), i.e., logs produced by a model where all the branches going out of a fork node are always executed concurrently. However, even in such an extreme case, the rate is quite low, reaching at most $15\%$, precisely when $p\% = 10\%$. Therefore, the absolute number of constraints violated is still very small, and only represents a $0.15\%$ of all possible constraints associated with the underlying models. Moreover, note that increasing the amount of input constraints seems to help our approach to reduce violation rates. This can be explained by observing that, as the availability of more background knowledge makes each discovered model more similar to the true one, it is more likely that the former will eventually satisfy a higher amount of all the a-priori constraints associated with the latter.

## 7. DISCUSSION AND CONCLUSION

Current research is rather active in proposing process mining techniques supporting increasingly expressing modeling languages. However, most of the approaches proposed in the literature mine the causal dependencies that hold over the activities by completely ignoring the prior knowledge that in many cases is available to the analyst. This paper moves a systematic step to fill the gap, by proposing and analyzing a constraint-based framework for process mining, where background knowledge can be encoded in the form of precedence constraints. The computational complexity of the framework has been studied, and the whole approach has been implemented in a prototype system, which has been tested on different log data.

[13] As discussed before, the model returned by our algorithm is always guaranteed to satisfy all other kinds of constraints.

Note that the use of constraints is currently gaining attention in a different context, yet still related to process management, namely in the context of developing *declarative* approaches to support business automation. In fact, traditional modeling languages, such as Petri nets or causal nets, are *procedural* ones for they explicitly represent all the allowed behavior of the process, according to a "closed world" assumption. Opposed to this approach, recent research focused on developing declarative models where any possible enactment is allowed unless a constraint (expressed in some suitable formal logic) is known to hold and explicitly forbids it (see, e.g., [van der Aalst et al. 2009; Sadiq et al. 2005; Reichert et al. 2009]). Following a number of earlier attempts to use logic-based languages for the specification of business processes (see, e.g., [Bonner 1999; Senkul et al. 2002; Dourish et al. 1996; Joeris 2000; Wainer and de Lima Bezerra 2003; Lu et al. 2006; Attie et al. 1996]), *Declare* [Pesic et al. 2009] is nowadays the most solid platform adopting a declarative perspective for process modeling, where the semantics of each constraint is provided in terms of an associated Linear Temporal Logic (short: LTL) formula, whose (finite) models are precisely the set of all the allowed traces [Pesic et al. 2007; Pesic et al. 2010].

Interestingly, the problem of automatically inferring a process model from a given log available at hand has been considered within these declarative frameworks for process management, too. In particular, techniques specifically designed to infer Declare constraints have been presented by Maggi et al. [2011] and Maggi et al. [2012] and subsequently enhanced by [Maggi et al. 2013] in order to discover data-aware models. Another approach has been proposed by Di Ciccio and Mecella [2012], where constraints are eventually expressed in terms of *regular expressions* rather than in terms of LTL formulas. By sharing the spirit of the above proposals but focusing on a slightly different problem, Chesani et al. [2009] proposed a methodology to analyze a log whose traces are labeled as compliant or non-compliant, and whose goal is to learn a classification model defined as a set of rules/constraints expressed in the SCIFF [Alberti et al. 2008] language (eventually mapped in the Declare notation). Other approaches to build rule-based classification models have been proposed, for instance, by Bellodi et al. [2010] and Ferreira and Ferreira [2006].

By looking at the above body of literature, it clearly emerges that the use of constraints in these works is completely different from ours. Indeed, the role of constraints in such declarative frameworks is to specify the set of traces that are allowed, while in our approach they are instead used to specify the set of those possible process models that are of interest to the analyst. In fact, our perspective is the traditional one where the analyst wants to end up with a procedural model, and where constraints are used to prune the search space of all the models that can be the result of a mining phase. Accordingly, we adopted a language to specify topological constraints on the process model, rather than a language (such as LTL) tailored to define constraints on traces.

Even though the two approaches are completely orthogonal, we stress however that the research reported in this paper might have also an impact in the context of developing mining approaches for declarative process models. Indeed, in this setting, the idea of incorporating a-priori knowledge in the mining phase has been largely unexplored, and constitutes a promising avenue of further research. Currently, analysts are provided with the rules induced via learning methods and might refine them to incorporate their knowledge in a post-processing phase. Instead, it would be interesting to define approaches where analysts can a-priori formalize their knowledge and where rule/constraints adhering with it are automatically inferred from the available log.

Finally, to delineate another avenue for further research, we stress that logs have been viewed in the paper as multi-set of traces, by disregarding in particular the data involved over the various activities. Therefore, it is natural to look for extensions of

the proposed mining techniques capable to deal with context information (about, e.g., parameters and functional features of the activities) and process ontologies.

## References


Arya Adriansyah, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. 2011. Conformance Checking Using Cost-Based Fitness Analysis. In *EDOC*. 55–64.

R. Agrawal, D. Gunopulos, and F. Leymann. 1998. Mining process models from workflow logs. In *Proc. 6th Intl. Conf. on Extending Database Technology (EDBT'98)*. 469–483.

A.J.M.M. Weijters, W.M.P. van der Aalst, and A.K. Alves de Medeiros. 2006. *Process Mining with the HeuristicsMiner Algorithm*. Technical Report. Eindhoven University of Technology, Eindhoven.

Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. 2008. Verifiable agent interaction in abductive logic programming: The SCIFF framework. *ACM Trans. Comput. Logic* 9 (2008), 29:1–29:43. Issue 4.

Nikolas Anastasiou, Tzu-Ching Horng, and William Knottenbelt. 2011. Deriving generalised stochastic Petri net performance models from high-precision location tracking data. In *Proceedings of the 5th International ICST Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS '11)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 91–100. http://dl.acm.org/citation.cfm?id=2151688.2151700

Paul C. Attie, Munindar P. Singh, E. Allen Emerson, Amit P. Sheth, and Marek Rusinkiewicz. 1996. Scheduling workflows by enforcing intertask dependencies. *Distributed Systems Engineering* 3, 4 (1996), 222–238.

Elena Bellodi, Fabrizio Riguzzi, and Evelina Lamma. 2010. Probabilistic Declarative Process Mining. In *KSEM*. 292–303.

A. Bonner. 1999. Workflow, Transactions, and Datalog. In *Proc. of the 18th ACM Symposium on Principles of Database Systems (PODS'99)*. 294–305.

Andrea Burattin and Alessandro Sperduti. 2011. PLG: A Framework for the Generation of Business Process Models and Their Execution Logs. In *Business Process Management Workshops*. Lecture Notes in Business Information Processing, Vol. 66. Springer Berlin Heidelberg, 214–219.

C. W. K. Chen and D. Y. Y. Yun. 2003. Discovering Process Models from Execution History by Graph Matching. In *Proc. 4th Intl. Conf. on Intelligent Data Engineering and Automated Learning (IDEAL'03)*. 887–892.

Federico Chesani, Evelina Lamma, Paola Mello, Marco Montali, Fabrizio Riguzzi, and Sergio Storari. 2009. Exploiting Inductive Logic Programming Techniques for Declarative Process Mining. *T. Petri Nets and Other Models of Concurrency* 2 (2009), 278–295.

Sandra de Amo and Daniel A. Furtado. 2007. First-order temporal pattern mining with regular expression constraints. *Data & Knowledge Engineering* 62 (September 2007), 401–420. Issue 3.

A. K. A de Medeiros, B. F. van Dongen, W. M. P. van der Aalst, and A. J. M. M. Weijters. 2004. *Process Mining: Extending the $\alpha$-algorithm to Mine Short Loops*. Technical Report. University of Technology, Eindhoven. BETA Working Paper Series, WP 113.

Luc De Raedt, Tias Guns, and Siegfried Nijssen. 2008. Constraint programming for itemset mining. In *Proceeding of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '08)*. ACM, New York, NY, USA, 204–212.

J. De Weerdt, M. De Backer, J. Vanthienen, and B. Baesens. 2011. A robust F-measure for evaluating discovered process models. In *Proc. of 2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2011)*. 148–155.

R. Dechter. 1992. Constraint networks. *Encyclopedia of Artificial Intelligence* (1992), 276–285.

Claudio Di Ciccio and Massimo Mecella. 2012. Mining Constraints for Artful Processes.. In *BIS (Lecture Notes in Business Information Processing)*, Witold Abramowicz, Dalia Kriksciuniene, and Virgilijus Sakalauskas (Eds.), Vol. 117. Springer, 11–23.

Paul Dourish, Jim Holmes, Allan MacLean, Pernille Marqvardsen, and Alex Zbyslaw. 1996. Freeflow: Mediating Between Representation and Action in Workflow Systems. In *Proceedings of the CM Conference on Computer Supported Cooperative Work (CSCW '96)*. 190–198.

Hugo M. Ferreira and Diogo R. Ferreira. 2006. An Integrated Life Cycle for Workflow Management Based on Learning and Planning. *Int. J. Cooperative Inf. Syst.* 15, 4 (2006), 485–505.

M.R. Garey and D.S. Johnson. 1979. *Computers and Intractability. A Guide to the Theory of* NP*-completeness*. Freeman and Comp., NY, USA.

Stijn Goedertier, David Martens, Jan Vanthienen, and Bart Baesens. Robust Process Discovery with Artificial Negative Events. *J. Mach. Learn. Res.* (????).

Stijn Goedertier, David Martens, Jan Vanthienen, and Bart Baesens. 2009. Robust Process Discovery with Artificial Negative Events. *Journal of Machine Learning Research* 10 (2009), 1305–1340.

Gianluigi Greco, Antonella Guzzo, and Luigi Pontieri. 2007. An Information-Theoretic Framework for Process Structure and Data Mining. *IJDWM* 3, 4 (2007), 99–119.

Gianluigi Greco, Antonella Guzzo, and Luigi Pontieri. 2012. Process Discovery via Precedence Constraints. In *ECAI*. 366–371.

G. Greco, A. Guzzo, L. Pontieri, and D. Saccà. 2006. Discovering Expressive Process Models by Clustering Log Traces. *IEEE Trans. on Knowledge and Data Engineering* 18, 8 (2006), 1010–1027.

Tias Guns, Siegfried Nijssen, and Luc De Raedt. 2011. Itemset mining: A constraint programming perspective. *Artif. Intell.* 175 (2011), 1951–1983. Issue 12-13.

M. Hammori, J. Herbst, and N. Kleiner. 2006. Interactive workflow mining – requirements, concepts and implementation. *Data & Knowledge Engineering* 56, 1 (2006), 41–63.

J. Herbst and D. Karagiannis. 2000. Integrating Machine Learning and Workflow Management to Support Acquisition and Adaptation of Workflow Models. *Journal of Intelligent Systems in Accounting, Finance and Management* 9 (2000), 67–92.

J. Herbst and D. Karagiannis. 2003. Workflow mining with InWoLvE. *Computers in Industry. Special Issue: Process/Workflow Mining* 53, 3 (2003), 245ñ–264.

Haiyang HU, Jianen XIE, and Hua HU. 2011. A Novel Approach for Mining Stochastic Process Model from Workflow Logs. *Journal of Computational Information Systems* 7, 9 (2011), 3113–3126.

Gregor Joeris. 2000. Decentralized and Flexible Workflow Enactment Based on Task Coordination Agents. In *2nd Intl. BiConference Workshop on Agent-Oriented Information Systems (AOIS00+CAiSE00*. iCue Publishing, 41–62.

G. Keller, M. Nüttgens, and A. W. Scheer. 1992. *Semantische Processmodellierung auf der Grundlage Ereignisgesteuerter Processketten (EPK)*. University of Saarland, Saarbrücken. Heft 89 (in German) pages.

Ruopeng Lu, Shazia Sadiq, Vineet Padmanabhan, and Guido Governatori. 2006. Using a Temporal Constraint Network for Business Process Execution. In *Seventeenth Australasian Database Conference (ADC2006) (CRPIT)*, Gillian Dobbie and James Bailey (Eds.), Vol. 49. ACS, Hobart, Australia, 157–166.

Fabrizio Maria Maggi, R. P. Jagadeesh Chandra Bose, and Wil M. P. van der Aalst. 2012. Efficient Discovery of Understandable Declarative Process Models from Event Logs. In *CAiSE*. 270–285.

Fabrizio Maria Maggi, Marlon Dumas, Luciano Garcia-Banuelos, and Marco Montali. 2013. Discovering Data-Aware Declarative Process Models from Event Logs. In *BPM*. 81–96.

Fabrizio Maria Maggi, Arjan J. Mooij, and Wil M. P. van der Aalst. 2011. User-guided discovery of declarative process models. In *CIDM*. 192–199.

A. K. Medeiros, A. J. Weijters, and W. M. Aalst. 2007. Genetic process mining: an experimental evaluation. *Data Mining and Knowledge Discovery* 14 (2007), 245–304. Issue 2.

M.T. Wynn, C. Ouyang, A.H.M. ter Hofstede, and C.J. Fidge. 2009. *Workflow Support for Product Recall Coordination*. Technical Report. BPMcenter.org.

Siegfried Nijssen, Tias Guns, and Luc De Raedt. 2009. Correlated itemset mining in ROC space: a constraint programming approach. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '09)*. ACM, New York, NY, USA, 647–656.

Maja Pesic, Dragan Bosnacki, and Wil M. P. van der Aalst. 2010. Enacting Declarative Languages Using LTL: Avoiding Errors and Improving Performance. In *SPIN*. 146–161.

Maja Pesic, Helen Schonenberg, and Wil M. P. van der Aalst. 2009. DECLARE Demo: A Constraint-based Workflow Management System. In *BPM (Demos)*.

Maja Pesic, M. H. Schonenberg, Natalia Sidorova, and Wil M. P. van der Aalst. 2007. Constraint-Based Workflow Models: Change Made Easy. In *OTM Conferences (1)*. 77–94.

Manfred Reichert, Stefanie Rinderle-Ma, and Peter Dadam. 2009. Transactions on Petri Nets and Other Models of Concurrency II. Springer-Verlag, Berlin, Heidelberg, Chapter Flexibility in Process-Aware Information Systems, 115–135. DOI:http://dx.doi.org/10.1007/978-3-642-00899-3_7

A. Rozinat and W. M. P. van der Aalst. 2008. Conformance checking of processes based on monitoring real behavior. *Inf. Syst.* 33, 1 (March 2008), 64–95. DOI:http://dx.doi.org/10.1016/j.is.2007.07.001

Shazia Wasim Sadiq, Maria E. Orlowska, and Wasim Sadiq. 2005. Specification and validation of process constraints for flexible workflows. *Inf. Syst.* 30, 5 (2005), 349–378.

Thomas J. Schaefer. 1978. The complexity of satisfiability problems. In *Proceedings of the tenth annual ACM symposium on Theory of computing (STOC '78)*. ACM, New York, NY, USA, 216–226. DOI:http://dx.doi.org/10.1145/800133.804350

G. Schimm. 2003. Mining Most Specific Workflow Models from Event-Based Data. In *Proc. of Int. Conf. on Business Process Management*. 25–40.

P. Senkul, M. Kifer, and I.H. Toroslu. 2002. A logical Framework for Scheduling Workflows Under Resource Allocation Constraints. In *Proc. 28th Int. Conf. on Very Large Data Bases (VLDB'02)*. 694–702.

S.K.L.M. van den Broucke, J. De Weerdt, J. Vanthienen, and B. Baesens. 2013. A comprehensive benchmarking framework (CoBeFra) for conformance analysis between procedural process models and event logs in ProM. In *Proc. of 2013 IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2013)*. 254–261.

W.M.P. van der Aalst. 1998. The Application of Petri Nets to Worflow Management. *Journal of Circuits, Systems, and Computers* 8, 1 (1998), 21–66.

Wil van der Aalst, Arya Adriansyah, and Boudewijn van Dongen. Replaying history on process models for conformance checking and performance analysis. *Wiley Int. Rev. Data Min. and Knowl. Disc.* (????).

Wil van der Aalst, Arya Adriansyah, and Boudewijn van Dongen. Replaying history on process models for conformance checking and performance analysis. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 2, 2 (????).

Wil van Der Aalst, Arya Adriansyah, and Boudewijn van Dongen. 2011. Causal nets: a modeling language tailored towards process discovery. In *Proceedings of the 22nd international conference on Concurrency theory (CONCUR'11)*. Springer-Verlag, Berlin, Heidelberg, 28–42. http://dl.acm.org/citation.cfm?id=2040235.2040239

Wil M. P. van der Aalst. 2011. *Process Mining: Discovery, Conformance and Enhancement of Business Processes* (1st ed.). Springer Publishing Company, Incorporated.

W. M. P. van der Aalst, J. Desel, and E Kindler. 2002. On the Semantics of EPCs: A Vicious Circle. In *Proc. EPK 2002: Business Process Management using EPCs*. 71–80.

Wil M. P. van der Aalst, Maja Pesic, and Helen Schonenberg. 2009. Declarative workflows: Balancing between flexibility and support. *Computer Science - R&D* 23, 2 (2009), 99–113.

W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. 2003. Workflow Mining: A Survey of Issues and Approaches. *Data & Knowledge Engineering* 47, 2 (2003), 237–267.

W. M. P. van der Aalst and K. M. van Hee. 2002. *Workflow Management: Models, Methods, and Systems*. MIT Press.

W. M. P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. 2004. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 16, 9 (2004), 1128–1142.

J. M. E. M. van der Werf, B. F. van Dongen, C. A. J. Hurkens, and A. Serebrenik. 2009. Process Discovery using Integer Linear Programming. *Fundamenta Informaticae* 94 (2009), 387–412. Issue 3-4.

B. van Dongen, A. de Medeiros, H. Verbeek, A. Weijters, and W. van der Aalst. 2005. The ProM Framework: A New Era in Process Mining Tool Support. In *Applications and Theory of Petri Nets 2005*, Gianfranco Ciardo and Philippe Darondeau (Eds.). Lecture Notes in Computer Science, Vol. 3536. Springer Berlin / Heidelberg, 1105–1116.

J. Wainer and F. de Lima Bezerra. 2003. Constraint-Based Flexible Workflows. In *Proceedings of the 9th International Workshop on Groupware: Design, Implementation, and Use (CRIWG 2003)*, Vol. 2806. Springer-Verlag, Berlin, 151 – 158.

Jochen De Weerdt, Manu De Backer, Jan Vanthienen, and Bart Baesens. 2012. A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs. *Information Systems* 37, 7 (2012), 654 – 676.

A.J.M.M. Weijters and W.M.P. van der Aalst. 2001. Process Mining: Discovering Workflow Models from Event-Based Data. In *Proceedings of the 13th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2001)*. 283–290.

A. J. M. M. Weijters and W. M. P. van der Aalst. 2003. Rediscovering workflow models from event-based data using Little Thumb. *Integrated Computer-Aided Engineering* 10, 2 (2003), 151–162.

Lijie Wen, Jianmin Wang, Wil M. Aalst, Biqing Huang, and Jiaguang Sun. 2009. A novel approach for process mining based on event types. *J. Intell. Inf. Syst.* 32, 2 (April 2009), 163–190. DOI:http://dx.doi.org/10.1007/s10844-007-0052-1

## A. PROOFS IN SECTION 2.2

PROOF OF THEOREM 2.7. *(if part).* Let $\mathcal{C} = \langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ be a causal net such that $\mathcal{C} \vdash L$. Let $t$ be any trace in $L$. Let $\mathcal{G}_t = (V_t, E_t)$ be the subgraph of $\mathcal{G} = (V, E)$ such that $V_t = \{t[1], ..., t[len(t)]\}$ and $E_t = \{(t[i], t[j]) \in E \mid 1 \leq i < j \leq len(t)\}$. Hence, $\mathcal{G}_t$ is the subgraph of $\mathcal{G}$ induced over the activities occurring in $t$, where we keep those edges of $E$ that conform with the ordering of the activities in $t$. Since $L$ is linear, $\mathcal{G}_t$ is acyclic and $t$ is a topologic sort of it. According to Definition 2.5, to conclude, it is then sufficient to prove that $\mathcal{G}_t$ is a dependency graph. In fact, $a_\perp$ and $a_\top$ have no ingoing and outgoing edges, respectively, by construction of $\mathcal{G}_t$ and since $\mathcal{G}$ is a dependency graph. The final requirement to be checked is now that, for each activity $a \in V \setminus \{a_\perp, a_\top\}$, it must be the case that $a$ occurs in some path from $a_\perp$ to $a_\top$. In particular, since $\mathcal{G}_t$ is acyclic, we can equivalently check that each activity $a \in V \setminus \{a_\top\}$ (resp., $a \in V \setminus \{a_\perp\}$) has at least one outgoing (resp., ingoing) edge.

Let $\sigma$ be a valid binding sequence such that $\sigma_j = \langle t[j], ib_j, ob_j \rangle$ is the $j$-th binding activity of $\sigma$, for each $j \in \{1, ..., len(t)\}$. Note that $\sigma$ exists, since $\mathcal{C} \vdash L$. Consider first an activity $a \in V \setminus \{a_\top\}$, i.e, $a = t[j]$ where $j \in \{1, ..., len(t) - 1\}$. As $\sigma$ is a binding sequence, $ob_j \in \mathcal{O}(t[j])$ holds and hence we have that $ob_j \neq \emptyset$. In particular, there is an edge $(t[j], y) \in ob_j$ such that $(t[j], y) \in S^\sigma_j = S^\sigma_{j-1} \cup ob_j \setminus ib_j$, because $\sigma$ is valid so that $ib_j \subseteq S^\sigma_{j-1}$. Moreover, again because $\sigma$ is valid, $S^\sigma_{len(t)} = \emptyset$ holds, and hence there is an index $i \in \{j + 1, ..., len(t)\}$ such that $(t[j], y) \in ib_i$. It follows that $y = t[i]$ and thus $(t[j], t[i])$ belongs to $E$ (hence, to $E_t$ since $j < i$). So, we have shown that each activity $a \in V \setminus \{a_\top\}$ has at least one outgoing edge.

We conclude by claiming that, for each $j \in \{2, ..., len(t)\}$, there is an index $i \in \{1, ..., j - 1\}$ such that $(t[i], t[j])$ is in $E$ (hence, in $E_t$). In order to prove the claim, let again $\sigma$ be a valid binding sequence such that $\sigma_j = \langle t[j], ib_j, ob_j \rangle$ is the $j$-th binding activity of $\sigma$, for each $j \in \{1, ..., len(t)\}$. By definition of binding sequence, we know that $S^\sigma_j \subseteq S^\sigma_{j-1} \cup ob_j$ holds, for each $j \in \{1, ..., len(t)\}$. In particular, since $\sigma$ is valid, $S^\sigma_0 = \emptyset$ holds, and hence $S^\sigma_j \subseteq \bigcup_{h=1}^{j} ob_h$, for each $j \in \{1, ..., len(t)\}$. Recall now from the definition of causal net that $ob_j \in \mathcal{O}(t[j])$ implies $ob_j \subseteq \{(t[j], y) \mid (t[j], y) \in E\}$. Therefore, we conclude that:

$$S^\sigma_j \subseteq \bigcup_{h=1}^{j} \{(t[h], y) \mid (t[h], y) \in E\}, \text{for each } j \in \{1, ..., len(t)\}. \tag{1}$$

Let us exploit again the fact that $\sigma$ is valid, in order to derive that $ib_j \subseteq S^\sigma_{j-1}$ holds, for each $j \in \{1, ..., len(t)\}$. Let now $j$ be an index in the set $\{2, ..., len(t)\}$. Observe that, by definition of causal net, $ib_j \in \mathcal{I}(t[i])$ implies that $ib_j \subseteq \{(x, t[j]) \mid (x, t[j]) \in E\}$, with $ib_j$ being in particular non empty. Therefore, by looking again at Equation 1 above and recalling that $ib_j \subseteq S^\sigma_{j-1}$, we can eventually conclude that an index $i \in \{1, ..., j - 1\}$ exists such that $(t[i], t[j])$ occurs in $E$.

*(only-if part).* Assume that, for each trace $t \in L$, there is a subgraph $\mathcal{G}_t$ of $\mathcal{G} = (V, E)$ such that $\mathcal{G}_t$ is an acyclic dependency graph and $t$ is a topologic sort of $\mathcal{G}_t$. For each trace $t$ in $L$ and for each index $j \in \{1, ..., len(t)\}$, define the input binding $ib_{t,j} = \{(x, t[j]) \mid (x, t[j]) \in E, x \in \{t[1], ..., t[j - 1]\}\}$ and the output binding $ob_{t,j} = \{(t[j], y) \mid (t[j], y) \in E, y \in \{t[j + 1], ..., t[len(t)]\}\}$. Note that, for each $j \in \{2, ..., len(t)\}$, $ib_{t,j} \neq \emptyset$ holds. Indeed, as $\mathcal{G}_t$ is a dependency graph, for each $j \in \{2, ..., len(t)\}$, $t[j]$ has at least one ingoing edge $(x, t[j])$ in $\mathcal{G}_t$. Moreover, since $t$ is a topologic sort of $\mathcal{G}_t$, we actually have that $x \in \{t[1], ..., t[j - 1]\}$. Similarly, it can be seen that, for each $j \in \{1, ..., len(t) - 1\}$, $ob_{t,j} \neq \emptyset$ holds. Finally, it is immediate to check that $ib_{t,1} = ob_{t,len(t)} = \emptyset$.

Now, consider the functions $\mathcal{I}$ and $\mathcal{O}$ such that, for each $a \in V$,

— $\mathcal{I}(a) = \bigcup_{t \in L, j | t[j]=a} ib_{t,j} \cup I_a$, where $I_a = \{(x, a) \mid (x, a) \in E\}$;
— $\mathcal{O}(a) = \bigcup_{t \in L, j | t[j]=a} ob_{t,j} \cup O_a$, where $O_a = \{(a, y) \mid (a, y) \in E\}$.

Since $\mathcal{G}$ is a dependency graph and given the definition of $I_a$ and $O_a$, for each $a \in V$, it is immediate to check that $\mathcal{C} = \langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ is a causal net. It remains to show that $\mathcal{C} \vdash L$, i.e., that for each trace $t$ in $L$, there is a valid binding sequence $\sigma$ such that $\langle t[j], ib_j, ob_j \rangle$ is the $j$-th binding activity of $\sigma$, for each $j \in \{1, ..., len(t)\}$. Let $t[1]...t[n]$ be a trace, and consider the sequence $\sigma = \langle t[1], ib_{t,1}, ob_{t,1} \rangle, ..., \langle t[n], ib_{t,n}, ob_{t,n} \rangle$. We have to show that $S^\sigma_n = \emptyset$ and $ib_{t,j} \subseteq S^\sigma_{j-1}$, for each $j \in \{1, ..., n\}$.

To prove the result, we first claim that, for each $j \in \{0, 1, ..., n\}$, $S^\sigma_j = \bigcup_{i=1}^{j} \{(t[i], t[i']) \mid (t[i], t[i']) \in E, i' > j\}$. In fact, the base case holds because $S^\sigma_0 = \emptyset$, by definition of the state of a causal net. Now, assume that the property holds up to the index $h < j$. We have to show that $S^\sigma_{h+1} = \bigcup_{i=1}^{h+1} \{(t[i], t[i']) \mid (t[i], t[i']) \in E, i' > h + 1\}$. To this end, recall that $S^\sigma_{h+1} = S^\sigma_h \cup ob_{t,h+1} \setminus ib_{t,h+1}$, so that $S^\sigma_{h+1} = \bigcup_{i=1}^{h} \{(t[i], t[i']) \mid (t[i], t[i']) \in E, i' > h\} \cup ob_{t,h+1} \setminus ib_{t,h+1}$ holds, by inductive hypothesis. Eventually, the result derives by the above expression and the fact that $ob_{t,h+1} = \{(t[h + 1], y) \mid (t[h + 1], y) \in E, y \in \{t[h + 2], ..., t[n]\}\}$ and $ib_{t,h+1} = \{(x, t[h + 1]) \mid (x, t[h + 1]) \in E, x \in \{t[1], ..., t[h]\}\}$.

Armed with the above property, we can now resume the proof. First, we have to show that $S^\sigma_n = \emptyset$. In fact, we have that $S^\sigma_n = \bigcup_{i=1}^{n} \{(t[i], t[i']) \mid (t[i], t[i']) \in E, i' > n\}$, which coincides with the empty set as $n$ is the length of $t$. Second, we have to show that $ib_{t,j} \subseteq S^\sigma_{j-1}$, for each $j \in \{1, ..., n\}$. To this end, recall that $ib_{t,j} = \{(x, t[j]) \mid (x, t[j]) \in E, x \in \{t[1], ..., t[j - 1]\}\}$, and eventually just check that $ib_{t,j} \subseteq \bigcup_{i=1}^{j-1} \{(t[i], t[i']) \mid (t[i], t[i']) \in E, i' > j - 1\} = S^\sigma_{j-1}$. □

PROOF OF THEOREM 2.10. *(if part).* Let $\mathcal{C} = \langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ be a causal net such that $\mathcal{C} \vdash L$, with $\mathcal{G} = (N, E)$. This means that, for each trace $t$ in $L$, there is a binding sequence $\sigma$ such that $\sigma$ is valid w.r.t. $\mathcal{C}$, and the $j$-th element of $\sigma$ has the form $\langle t[j], ib_j, ob_j \rangle$, for each $j \in \{1, ..., len(t)\}$. Let $S^\sigma_j$ denote the state of $\mathcal{C}$ at the $j$-th step of $\sigma$, and consider the sequence $\bar{\sigma}$ having the same length $len(t)$ as $\sigma$ and whose $j$-th element $\langle \bar{a}_j, \bar{ib}_j, \bar{ob}_j \rangle$, for each $j \in \{1, ..., len(t)\}$, is obtained from $\langle t[j], ib_j, ob_j \rangle$ as follows:

- $\bar{a}_j$ is the virtual activity $t[j]^{\langle k \rangle}$, where $k$ is the number of occurrences of the activity $t[j]$ in $t[1]...t[j]$, i.e., the number of binding activities defined over $t[j]$ in the first $j$ elements of $\sigma$;
- For each output binding $(t[j], y) \in ob_j$, let $\alpha_y$ denote the number of binding activities occurring up to the $j$-th step of $\sigma$ where $(t[j], y)$ occurs as an element of the output binding. Note that, since $\sigma$ is a valid sequence w.r.t. $\mathcal{C}$ (and since bindings are sets, i.e., multiple occurrences are not allowed), we are guaranteed about the existence of $\alpha_y$ binding activities where these output bindings are consumed, i.e., where they are taken as input. Let $next(j, y)$ be the index of the $\alpha_y$-th activity binding of this kind, hence, in particular with $(t[j], y) \in ib_{next(j,y)}$, and note that $next(j, y) > j$. Then, we define $\bar{ob}_j = \{(\bar{a}_j, \bar{a}_{next(j,y)}) \mid (t[j], y) \in ob_j\}$.
- For each input binding $(x, t[j]) \in ib_j$, let $\beta_y$ denote the number of binding activities occurring up to the $j$-th step of $\sigma$ where $(x, t[j])$ is taken as input. Note that, since $\sigma$ is a valid sequence w.r.t. $\mathcal{C}$, we are guaranteed about the existence of $\beta_y$ binding activities where these input bindings are produced. Let $prev(j, x)$ be the index of the $\beta_y$-th activity binding of this kind, hence, in particular with $(x, t[j]) \in ob_{prev(j,x)}$, and note that $prev(j, x) < j$. Then, we define $\bar{ib}_j = \{(\bar{a}_{prev(j,x)}, \bar{a}_j) \mid (x, t[j]) \in ib_j\}$.

Define now $\bar{\mathcal{G}}_t = (\bar{V}_t, \bar{E}_t)$ as the graph where $\bar{V}_t = \mathcal{A}(\{unfold(t)\})$ and where $\bar{E}_t = \bigcup_{j=1}^{len(t)} (i\bar{b}_j \cup o\bar{b}_j)$. We claim that $\bar{\mathcal{G}}_t$ is a dependency graph. Indeed, note first that $\bar{\mathcal{G}}_t$ is acyclic because each edge in $\bar{E}_t$ has the form $(\bar{a}_i, \bar{a}_j)$, with $i < j$. In particular, $t[1]^{\langle 1 \rangle}$ and $t[len(t)]^{\langle 1 \rangle}$ play the role of the starting and the terminating activity, respectively. Then, consider an activity $\bar{a}_j \in \bar{V}_t \setminus \{t[1]^{\langle 1 \rangle}\}$ (resp., $\bar{a}_j \in \bar{V}_t \setminus \{t[len(t)]^{\langle 1 \rangle}\}$). Note that $\bar{a}_j$ has at least one ingoing (resp., outgoing) edge in $\bar{E}_t$, because there is at least an edge of the form $(x, t[j])$ (resp., $(t[j], y)$) in $ib_j$ (resp., $ob_j$), by the fact that $\mathcal{C} = \langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ is a causal net (hence, $ib_j \neq \emptyset$ and $ob_j \neq \emptyset$ hold) and that $\sigma$ is a binding sequence. Since $\bar{\mathcal{G}}_t$ is acyclic, the above properties entail that $\bar{a}_j$ occurs in a path from $t[1]^{\langle 1 \rangle}$ to $t[len(t)]^{\langle 1 \rangle}$.

Consider now the functions $\bar{\mathcal{I}}_t$ and $\bar{\mathcal{O}}_t$ such that $\bar{\mathcal{I}}_t(\bar{a}_j) = i\bar{b}_j$ and $\bar{\mathcal{O}}_t(\bar{a}_j) = o\bar{b}_j$, for each $\bar{a}_j \in \bar{V}_t$. Since $\bar{\mathcal{G}}_t$ is a dependency graph and given the construction of its edges, we derive that $\langle \bar{\mathcal{G}}_t, \bar{\mathcal{I}}_t, \bar{\mathcal{O}}_t \rangle$ is a causal net (over $\mathcal{A}(\{unfold(t)\})$). Moreover, we claim that $\bar{\sigma}$ is valid w.r.t. $\langle \bar{\mathcal{G}}_t, \bar{\mathcal{I}}_t, \bar{\mathcal{O}}_t \rangle$. To prove the claim, note first that, given the above construction for $\bar{\sigma}$, if $i = prev(j, x)$ (resp., $i = next(j, y)$), then $x = t[i]$ (resp., $y = t[j]$) and $j = next(i, t[j])$ (resp., $j = prev(i, t[j])$). Then, consider the state $S_j^{\bar{\sigma}}$ of $\langle \bar{\mathcal{G}}_t, \bar{\mathcal{I}}_t, \bar{\mathcal{O}}_t \rangle$ at the $j$-th step of $\bar{\sigma}$, and observe that the following properties hold.

— For each $j \in \{1, ..., len(t)\}$, $i\bar{b}_j \subseteq S_{j-1}^{\bar{\sigma}}$. Indeed, consider an element $(\bar{a}_{prev(j,x)}, \bar{a}_j)$ in $i\bar{b}_j$, let $i = prev(j, x)$, and recall that $(\bar{a}_{prev(j,x)}, \bar{a}_j) = (\bar{a}_i, \bar{a}_{next(i,t[i])})$. Therefore, $(\bar{a}_{prev(j,x)}, \bar{a}_j)$ occurs in $o\bar{b}_i$ and, hence, in $S_i^{\bar{\sigma}} = S_{i-1}^{\bar{\sigma}} \cup o\bar{b}_i \setminus i\bar{b}_i$. Eventually, as $\bar{a}_j$ occurs only at the $j$-th step of $\bar{\sigma}$ and $i < j$, we have that $(\bar{a}_{prev(j,x)}, \bar{a}_j) \in S_{j-1}^{\bar{\sigma}}$.

— $S_{len(t)}^{\bar{\sigma}} = \emptyset$. Indeed, assume by contradiction that $(\bar{x}, \bar{y})$ occurs in $S_{len}^{\bar{\sigma}}(t)$. Then, there is an index $j$ such that $\bar{x} = \bar{a}_j$ and $(\bar{a}_j, \bar{y})$ occurs in $o\bar{b}_j$. By construction, we therefore have that $(\bar{x}, \bar{y}) = (\bar{a}_j, \bar{a}_{next(j,y)})$. However, by letting $i = next(j, y)$, we can write that $(\bar{a}_j, \bar{a}_{next(j,y)}) = (\bar{a}_{prev(i,t[j])}, \bar{a}_i) \in i\bar{b}_i$. Since $i > j$, it follows that $(\bar{x}, \bar{y}) \notin S_i^{\bar{\sigma}}$, as $\bar{x}$ occurs only at the $j$-th step of $\bar{\sigma}$. We conclude that $(\bar{x}, \bar{y}) \notin S_{len(t)}^{\bar{\sigma}}$. Contradiction.

By putting together the above results, we have so far shown that $\langle \bar{\mathcal{G}}_t, \bar{\mathcal{I}}_t, \bar{\mathcal{O}}_t \rangle$ is a causal net with $\langle \bar{\mathcal{G}}_t, \bar{\mathcal{I}}_t, \bar{\mathcal{O}}_t \rangle \vdash \{unfold(t)\}$, for each $t$ in $L$. It follows that we can apply Theorem 2.7 on $\langle \bar{\mathcal{G}}_t, \bar{\mathcal{I}}_t, \bar{\mathcal{O}}_t \rangle$, and we derive that $\bar{\mathcal{G}}_t \vdash_a \{unfold(t)\}$ holds, for each $t$ in $L$.

Finally, consider the graph $\bar{\mathcal{G}}_t$ and note that conditions (1) and (2) in Definition 2.8 are trivially satisfied, by construction of its edges. Indeed, all edges outgoing from $\bar{a}_j$, with $j \in \{1, ..., len(t)\}$, have the form $(\bar{a}_j, \bar{a}_{next(j,y)})$, where $(t[j], y) \in ob_j$. In particular, $(\bar{a}_j, \bar{a}_{next(j,y)})$ is univocally determined by $y$, so that $(\bar{a}_j, \bar{a}_{next(j,y)}) \neq (\bar{a}_j, \bar{a}_{next(j,y')})$ implies that $y \neq y'$ and hence $\bar{a}_{next(j,y)}$ and $\bar{a}_{next(j,y')}$ are virtual activities built from different true activities. A similar line of reasoning applies to the edges incoming into $\bar{a}_j$. Moreover, $fold(\bar{\mathcal{G}}_t)$ is a subgraph of $\mathcal{G}$. Indeed, $\bar{E}_t$ is the union of all bindings in $\langle \bar{\mathcal{G}}_t, \bar{\mathcal{I}}_t, \bar{\mathcal{O}}_t \rangle$, and any element in these bindings is of the form $(x^{\langle i \rangle}, y^{\langle j \rangle})$, where $(x, y)$ occurs in a binding in $\mathcal{I}$ or $\mathcal{O}$ and, hence, is an edge in $E$. Therefore, according to Definition 2.8, we have shown that $\mathcal{G} \vdash L$ holds.

*(only-if part).* Assume that $\mathcal{G} \vdash L$ holds, with $\mathcal{G} = (V, E)$. Thus, for each trace $t \in L$, there is a graph $\bar{\mathcal{G}}_t = (\bar{V}_t, \bar{E}_t)$ such that $fold(\bar{\mathcal{G}}_t)$ is a subgraph of $\mathcal{G}$, $\bar{\mathcal{G}}_t \vdash_a \{unfold(t)\}$, and the following two conditions hold:

(1) there is no pair of edges $(x^{\langle i \rangle}, y^{\langle j \rangle})$, $(x^{\langle i \rangle}, y^{\langle j' \rangle})$ in $\bar{\mathcal{G}}_t$ such that $j \neq j'$, and
(2) there is no pair of edges $(x^{\langle i \rangle}, y^{\langle j \rangle})$, $(x^{\langle i' \rangle}, y^{\langle j \rangle})$ in $\bar{\mathcal{G}}_t$ such that $i \neq i'$.

By Theorem 2.7 applied on the fact that $\bar{\mathcal{G}}_t \vdash_a \{unfold(t)\}$ holds, we derive that there is a causal net $\bar{\mathcal{C}}_t = \langle \bar{\mathcal{G}}_t, \bar{\mathcal{I}}_t, \bar{\mathcal{O}}_t \rangle$ such that $\bar{\mathcal{C}}_t \vdash \{unfold(t)\}$ holds. This means that there is

binding sequence $\bar{\sigma}$ that is valid w.r.t. $\bar{\mathcal{C}}$ and where the $j$-th binding activity $\langle \bar{a}_j, \bar{ib}_j, \bar{ob}_j \rangle$ of $\bar{\sigma}$, for each $j \in \{1, ..., len(t)\}$, is such that $\bar{a}_j$ is the symbol $t[j]^{\langle k \rangle}$, with $k$ being the number of occurrences of $t[j]$ in $t[1]...t[j]$.

Let $\mathcal{I}_t$ (resp., $\mathcal{O}_t$) be the function such that, for each node $z$ in $fold(\bar{\mathcal{G}}_t)$, $\mathcal{I}_t(z) = \{ib_z \mid$ exists $j$ s.t. $\bar{ib}_z \in \bar{\mathcal{I}}_t(z^{\langle j \rangle})\}$ (resp., $\mathcal{O}_t(z) = \{ob_z \mid$ exists $i$ s.t. $\bar{ob}_z \in \bar{\mathcal{O}}_t(z^{\langle i \rangle})\}$) where $ib_z$ (resp., $ob_z$) is the binding obtained from $\bar{ib}_z$ (resp., $\bar{ob}_z$) by stripping off the instantiation numbers of the virtual symbols. Similarly, define $\sigma$ as the sequence obtained from $\bar{\sigma}$ by stripping off the instantiation numbers of the virtual symbols. Note that because of the conditions (1) and (2) above, and since bindings are defined over the edges of $\bar{\mathcal{G}}_t$, $|\bar{ib}_z| = |ib_z|$ and $|\bar{ob}_z| = |ob_z|$ hold, for each activity $z$. Therefore, if $S^{\bar{\sigma}}_j$ is the state of $\bar{\mathcal{C}}_t$ at the $j$-th step of $\bar{\sigma}$, then the state $S^{\sigma}_j$ of $\mathcal{C}_t$ at the $j$-th step of $\sigma$ can be obtained from $S^{\bar{\sigma}}_j$ by just stripping off the instantiation numbers of the symbols it contains. Hence, since $\bar{\sigma}$ is valid w.r.t. $\bar{\mathcal{C}}_t$, we can conclude that $\sigma$ is valid w.r.t. $\mathcal{C}_t = \langle fold(\bar{\mathcal{G}})_t, \mathcal{I}_t, \mathcal{O}_t \rangle$. This witness that $\mathcal{C}_t \vdash \{t\}$ holds.

Let now $\bar{\mathcal{G}} = (\bar{V}, \bar{E})$ be the graph such that $\bar{V} = \bigcup_{t \in L} \bar{V}_t$ and $\bar{E} = \bigcup_{t \in L} \bar{E}_t$, and let $\mathcal{I}$ and $\mathcal{O}$ be the functions such that $\mathcal{I}(z) = \bigcup_{t \in L} \mathcal{I}_t(z)$ and $\mathcal{O}(z) = \bigcup_{t \in L} \mathcal{O}_t(z)$, for each $z$ in $fold(\bar{\mathcal{G}}) = (V_f, E_f)$. Note that $fold(\bar{\mathcal{G}})$ is a subgraph of $\mathcal{G}$, and that $\mathcal{C} = \langle fold(\bar{\mathcal{G}}), \mathcal{I}, \mathcal{O} \rangle$ is a causal net such that $\mathcal{C} \vdash L$. Therefore, in the case where $fold(\bar{\mathcal{G}}) = \mathcal{G}$, then we have derived that there is a causal net $\mathcal{C} = \langle \mathcal{G}, \mathcal{I}, \mathcal{O} \rangle$ such that $\mathcal{C} \vdash L$. To conclude the proof, the only remaining case to be analyzed is when $fold(\bar{\mathcal{G}})$ is a proper subgraph of $\mathcal{G}$. In this case, consider the functions $\mathcal{I}'$ and $\mathcal{O}'$ such that, for each $z \in V \cap V_f$, $\mathcal{I}'(z) = \mathcal{I}(z) \cup I_z$ and $\mathcal{O}'(z) = \mathcal{O}'(z) \cup O_z$, and for each $z \in V \setminus V_f$, $I'(z) = I_z$ and $\mathcal{O}'(Z) = O_z$, where $I_z = \{(x, z) \mid (x, z) \in E\}$ and $O_z = \{(z, y) \mid (z, y) \in E\}$. As $\mathcal{G}$ is a dependency graph, by construction of $\mathcal{I}'$ and $\mathcal{O}'$, we trivially have that $\langle \mathcal{G}, \mathcal{I}', \mathcal{O}' \rangle$ is a causal net, which generalizes the behavior of $\mathcal{C} = \langle fold(\bar{\mathcal{G}}), \mathcal{I}, \mathcal{O} \rangle$ over the nodes and the edges in $\mathcal{G}$ that do not occur in $fold(\bar{\mathcal{G}})$. Since $\mathcal{C} \vdash L$, we conclude that $\langle \mathcal{G}, \mathcal{I}', \mathcal{O}' \rangle \vdash L$. Indeed, if $\sigma$ is a sequence valid w.r.t. $\mathcal{C}$, then it is also valid w.r.t.$\langle \mathcal{G}, \mathcal{I}', \mathcal{O}' \rangle$. □

PROOF OF THEOREM 2.12. In the case where conditions (1) and (2) in Definition 2.8 are not guaranteed to hold, by inspecting the proof of the only-if part of Theorem 2.10, it can be checked that the structure $\mathcal{C}$ built there over $\mathcal{G}$ is still such that $\mathcal{C} \vdash L$ holds. The only difference is that $\mathcal{C}$ might be a possibly extended causal net. □