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

Process Discovery with Precedence Constraints

Gianluigi Greco [2], Antonella Guzzo [3], Luigi Pontieri[1]

RT-ICAR-CS-11-04 **Ottobre 2011**

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 with Precedence Constraints

Gianluigi Greco[1], Antonella Guzzo[2], and Luigi Pontieri[3]

Dept. of Mathematics[1] and DEIS[2], University of Calabria, 87036, Rende, Italy
ICAR-CNR[3], 87036, Rende,Italy
ggreco@mat.unical.it, guzzo@deis.unical.it, pontieri@icar.cnrt.it

Abstract. The automatic discovery of a process model out of a historical log traces can be of great value for both analysis and design tasks, and is a major goal of Process Mining approaches. A key step in discovering such a model consists in detecting a graph of causal/precedence dependencies over process activities – which can be possibly used to eventually derive more expressive control-flow (e.g., Petri-net based) models. To this end, most of current approaches exploit greedy heuristics and discard a-priori some dependencies assuming that the given log is *complete* (i.e., it covers the possible behavior of the process) – so risking to be ineffective when working with high-concurrency processes and with noisy and temporally-biased logs. Despite the usage of a-priori knowledge could improve the effectiveness, efficiency and robustness (w.r.t. incomplete/noisy data) of dependency mining algorithms, it has been given little attention so far. This paper fills the gap by proposing a *constraint-based* process discovery framework, where background knowledge can be encoded via *precedence constraints*, while the search of dependencies can be encoded as either a *constraints satisfaction* or a *constraints satisfaction optimization* problem. The computational complexity of such problems is studied deeply, and their tractability frontier is traced w.r.t. the different classes of constraints involved. The whole approach has been implemented in a prototype system, leveraging on a solid constraint programming platform, and tested on both synthesized and real log data.

1 Introduction

1.1 Mining Causal Dependencies

Process mining has recently emerged as a powerful approach to support the analysis and the design of complex business processes [40]. By analyzing a set of traces registering the sequence of tasks performed along several enactments of a transactional system, the goal of process mining[1] techniques is to (semi-)automatically derive a model explaining all the episodes recorded in it. The

[1] To be more precise, such a task should be named *process discovery*, while leaving the expression *process mining* for referring to the more general problem of analyzing the behavior of a process by taking advantage of historical log data. However, with an abuse of notation, the two expression will be used interchangeably in the paper.

"mined" model can then be used to design a detailed process schema capable of supporting forthcoming enactments, or to shed light on its actual behavior.

No matter of the specific process-oriented features being supported (such as synchronization and branching constructs, duplicate tasks, and or invisible tasks), the basic ability of any process mining algorithm is to single out the *causal dependencies* that are likely to hold among the activities occurring in the log. Indeed, process mining algorithms can be abstractly seen as sequentially carrying out two different sub-tasks: First, they analyze the log and apply some form of reasoning to learn such causal dependencies, often presented in form of *log-based ordering relations* [42, 40]. Then, they exploit the knowledge thereby acquired within mining algorithms that take into account advanced facets of process enactments [9, 26, 35, 34, 39, 22], and return process models formalized in expressive modeling languages (such as *Petri nets* [43] or *event driven process chains* EPCs [38, 23]).

In this paper, we focus on the former of the two tasks above, by elaborating techniques to mine causal dependencies from process logs. Accordingly, the output of such techniques are not (full) process models, but *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 (based on the logs) that an actual flow of information can occur from a to b. For example, the graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ in Figure 1 are two possible dependency graphs that one can associate with the traces *abcde* and *acbde*. Instead, the graph $\mathcal{G}_0$ does not properly reflect the flow associated with the trace *acbde*, where b is actually executed after c.

Dependency-graph discovery is a challenging problem in the case of concurrent processes, as traces flatten all the information related to the execution of "parallel" activities. Indeed, in the period of time elapsing between the execution of two activities, with one requiring some output produced by the other, the system may well register the execution of other activities involved over different branches of the process. Eventually, the fact that an activity always precedes another over the given traces does not necessarily witness a causal dependency between them. For example, from the fact that b precedes d in the two traces *abcde* and *acbde*, we cannot infer that b is a pre-requisite for the execution of d: In principle, b can be an activity executed over a different branch and without any causal dependency with d (as it is modeled in the graph $\mathcal{G}_2$).

As a matter of fact, several dependency graphs can in general be associated with a log given as input, and there are two complementary approaches that can be used to select the most appropriate one among them:

- On the one hand, one may design algorithms under the assumption that logs are *complete*, i.e., that they register all the possible traces for the underlying process. In this case, if an activity always precedes another, then we shall discard those dependency graphs where such activities are executed over different branches. For instance, if *abcde* and *acbde* are the only possible traces for the process, then b and d are not parallel activities and the control flow graph $\mathcal{G}_2$ has to be filtered out.

– On the other hand, one may avoid any assumption on the logs and think, instead, of exploiting some additional, semantic, knowledge that, in many cases, can be acquired by analyzing the given process from a conceptual viewpoint. For example, by a-priori knowing that b and d are parallel activities, we can discard $\mathcal{G}_1$ as a possible dependency graph, even though no trace is given where d actually occurred before b.

Log completeness has received considerable attention in the literature, and it is a crucial assumption under which a number of dependency mining methods can be proven to be correct, i.e., to be able of precisely recognizing all underlying relationships of precedence [42, 40]. In fact, dependency discovery is often carried out via heuristics approaches, for which the quality of the resulting graphs 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 to the following reasons.

Temporal bias: First, two parallel activities may appear in the same order over all log traces, 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 if the underlying process 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 that d starts.

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.

Last but not least, another, somewhat related, issue that has not given attention in process mining community is the possibility for the user to express declaratively preference/optimality criteria for the model searched. To the best of our knowledge, indeed, such criteria are typically stated in a fixed (non-parametric and implicit) way in current techniques.

1.2 Contribution

Despite the limitations illustrated above and affecting those algorithms that are based on the log-completeness assumption, it is surprising to observe that very few efforts have been spent to study the complementary approach of exploiting background knowledge in the discovery of causal precedences [14, 7, 16]. In fact, the use of background knowledge to improve the quality of the results—and even the scalability of the algorithms—has already been considered in a number of traditional data mining tasks (on relational data and on sequence data), such as

pattern mining [37, 50, 12, 27, 36, 31, 18] and clustering [46, 24, 21, 10]. However, such techniques have not found yet a counter-part in the process mining setting.

In this paper, we fill the gap by formulating a *constraint-based* process mining framework that can take advantage of possibly available background knowledge in order to circumvent the lack of raw data emerging when log completeness does not hold. In more detail:

- We propose a formal framework to specify additional properties on the dependency graphs that can be produced as output by process mining algorithms, in terms of *precedence constraints* on the available activities. Hence, we define discovery problems comprising a learning task (dependency graph mining) and a reasoning task (to check whether the resulting graphs satisfy the precedence constraints defined by the analysts).
- We show that precedence constraints are expressive enough to encode the core of the mining task. That is, we show that even the learning task can be *declaratively* formulated in terms of reasoning about precedence constraints, thus leading to a common environment where the two tasks are synergically combined and might be simultaneously carried out. Moreover, when the discovery of a dependency graph is formulated as an optimization problem with parametric costs, the analyst is allowed to have a say in the criterion used for choosing a solution among a set of possible ones, and to possibly express fine-grain preferences on structural properties of the result.
- 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 for w.r.t. them.
- We show that all the problems of interest when reasoning about precedence constraints can actually be encoded in terms of "standard" *constraints satisfaction problems*. This paves the way to reuse existing constraint programming platforms, and transparently exploit their sophisticated solution algorithms allowing them to scale over large datasets, as their recent application in data mining contexts have already demonstrated [12, 27].
- We implement all the techniques discussed in the paper and integrate them in a prototype system. Based on the above mappings, the task of reasoning about precedence constraints is delegated to a well-known constraint solver system available in the literature.
- Finally, we conducted experimental activity in order to validate the effectiveness of the proposed approach, and its scalability over large data sets.

Organization. The rest of the paper is organized as follows.

2 Preliminaries: Process Logs and Dependency Graphs

Process-aware systems usually store information on process enactments by tracing events related to the execution of the various activities. Abstracting from the specificity of the systems, we recall next a representation of process logs which is commonly adopted in the process mining literature (see, e.g., [42, 17]).

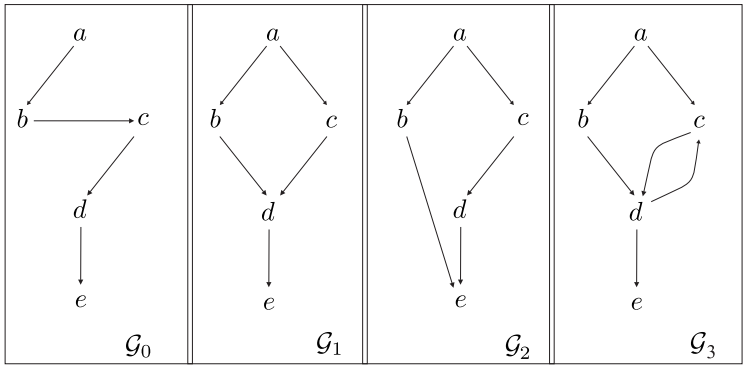

Fig. 1. Dependency graphs in the running example.

Logs of Acyclic Processes. Let $\mathcal{A}$ be an alphabet of symbols, univocally identifying the *activities* of some underlying process. A *process instance* $\mathcal{I}$ over $\mathcal{A}$ is a directed acyclic graph (V, E) with $V \subseteq \mathcal{A}$ and where a distinguished activity $a_\perp \in V$ exists from which every other activity can be reached—intuitively, $a_\perp$ is the starting activity for the enactment of $\mathcal{I}$. A trace t is a string over $\mathcal{A}$, and hence has the form $t[1]t[2]...t[n]$, with $t[i] \in \mathcal{A}$ being an activity for each $i \in \{1, ..., n\}$. If t is a topological sort of the process instance I, then t is called a *trace of* I, denoted by $\mathcal{I} \vdash t$; in this case, t does not contain multiple occurrences of the same activity, i.e., $t[i] \neq t[j]$, for each $i \neq j$. A *log* L is a multi-set of traces. For a log L, $\mathcal{A}(L)$ is the set of all the activities occurring over the traces in L. W.l.o.g., we hereinafter assume that $t[1] = a_\perp$, for each trace $t \in L$.

In process mining applications, a log L is given and the goal is to automatically derive a process model supporting the enactment of its traces. Several algorithms have been proposed to this end (see [40] and the references therein). No matter of the modeling language they adopt, one of their crucial abilities is to discover causal precedences among activities in $\mathcal{A}(L)$. Such precedences can be smoothly encoded via directed graphs as follows (cf. [1]).

Definition 1 (Support) Let L be a log, where no trace contains multiple occurrences of the same activity. Then, a graph $\mathcal{G} = (V, E)$ *supports* L, denoted by $\mathcal{G} \vdash L$, if for each trace $t \in L$, there is a subgraph $\mathcal{I}$ of $\mathcal{G}$ such that $\mathcal{I}$ is a process instance over $\mathcal{A}(L)$ and $\mathcal{I} \vdash t$. □

Example 1. Consider the trace $t_0 = abcde$ over the set of activities $\mathcal{A}(\{t_0\}) = \{a, b, c, d, e\}$. Then, it is easily checked that the graphs $\mathcal{G}_0$, $\mathcal{G}_1$, and $\mathcal{G}_2$ graphically reported in Figure 1 are such that $\mathcal{G}_0 \vdash \{t_0\}$, $\mathcal{G}_1 \vdash \{t_0\}$, and $\mathcal{G}_2 \vdash \{t_0\}$. ◁

Arbitrary Logs. A string t containing multiple occurrences of the same activity cannot be a trace of any process instance $\mathcal{I}$, as $\mathcal{I}$ is acyclic. Thus, if the underlying process involves loops, we need a mechanism to virtually unfold them.

For each trace t, let $\bar{t}$ denote the trace obtained from t by substituting, with the fresh (virtual) activity $a\langle i \rangle$, the i-th occurrence in t of any activity a.

Moreover, for a log L, let $\bar{L} = \{\bar{t} \mid t \in L\}$. Then, we say that a graph $\mathcal{G} = (V, E)$ is the *folding* of a graph $\bar{\mathcal{G}} = (\bar{V}, \bar{E})$ with $\bar{V} = \mathcal{A}(\bar{L})$ if $V = \{a \mid a\langle i\rangle \in \bar{V}\} = \mathcal{A}(L)$ and $E = \{(a, b) \mid (a\langle i\rangle, b\langle j\rangle) \in \bar{E}\}$.

Definition 2 Let L be a log. Then, a *dependency graph (CFG) $\mathcal{G}$ for L* is the folding of a graph $\bar{\mathcal{G}}$ such that $\bar{\mathcal{G}} \vdash \bar{L}$. □

Note that acyclic dependency graphs for L exist if, and only if, L contains no trace with repetitions of activities. Moreover, $\mathcal{G}$ is an acyclic dependency graph for L if, and only if, $\mathcal{G} \vdash L$.

Example 2. The graphs $\mathcal{G}_0$, $\mathcal{G}_1$, and $\mathcal{G}_2$ are dependency graphs for $\{t_0\}$. Instead, they are not dependency graphs for $\{t_1\}$, with t_1 being the trace *abcdcde*. Indeed, these graphs are acyclic while c and d occur twice in t_1. Note now that the trace $\bar{t}_1$ is the string $a\langle 1\rangle b\langle 1\rangle c\langle 1\rangle d\langle 1\rangle c\langle 2\rangle d\langle 2\rangle e\langle 1\rangle$, which is a topological sort of the acyclic graph $\bar{\mathcal{G}} = (\bar{V}, \bar{E})$ with $\bar{V} = \mathcal{A}(\{\bar{t}_1\})$ and where $\bar{E} = \{(a\langle 1\rangle, b\langle 1\rangle), (a\langle 1\rangle, c\langle 1\rangle), (d\langle 1\rangle, c\langle 2\rangle), (c\langle 2\rangle, d\langle 2\rangle), (b\langle 1\rangle, d\langle 2\rangle), (d\langle 2\rangle, e\langle 1\rangle)\}$. Thus, $\bar{\mathcal{G}} \vdash \{\bar{t}_1\}$. Eventually, note that the graph $\mathcal{G}_3$ depicted in Figure 1 is the folding of $\bar{\mathcal{G}}$. It follows that $\mathcal{G}_3$ is a dependency graph for $\{t_1\}$. ◁

3 Precedence Constraints

In this section, we formalize syntax and semantics of precedence constraints, discuss their application to the problem of mining dependency graphs, and analyze the computational complexity of the resulting framework.

3.1 Syntax and Semantics

Let $\mathcal{A}$ be a set of activities. A *precedence constraint* over $\mathcal{A}$ is an assertion aimed at expressing a relationship of precedence among some of the activities in $\mathcal{A}$. To define the syntax, we distinguish positive and negative constraints. A positive constraint π is either an expression of the form $S \to a$ (called *edge* constraint), or an expression of the form $S \rightsquigarrow a$ (called *path* constraint), where $S \subseteq \mathcal{A}$, with $|S| \geq 1$, is a non-empty set of activities and $a \in \mathcal{A} \setminus S$ is an activity. For a positive constraint π, $\neg\pi$ is a *negative* precedence constraint.

Precedence constraints are interpreted over directed graphs as follows. Let $\mathcal{G} = (V, E)$ be a directed graph such that $V \subseteq \mathcal{A}$ and $E \subseteq \mathcal{A} \times \mathcal{A}$. Then,

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

The graph $\mathcal{G}$ satisfies a set Π of precedence constraints and is called a *model* of Π, denoted by $\mathcal{G} \models \Pi$, if $\mathcal{G}$ satisfies each constraint in Π.

A foundational task in process mining, which is at the basis of a number of more elaborate mining techniques, consists of automatically building a dependency graph $\mathcal{G}$ for some log L given as input [1, 40]. In this context, precedence constraints can be naturally exploited to formalize additional requirements that dependency graphs discovered from L have to satisfy. This gives rise to the following two problems, where we explicitly distinguish the variant where the desired dependency graph is required to be acyclic.

DG-Mining**:** Given a log L and a set Π of precedence constraints over $\mathcal{A}(L)$, compute any dependency graph $\mathcal{G}$ for L such that $\mathcal{G} \models \Pi$.

Acyclic-DG-Mining**:** Given a log L and a set Π of precedence constraints over $\mathcal{A}(L)$, compute any acyclic dependency graph $\mathcal{G}$ for L such that $\mathcal{G} \models \Pi$.

Note that for $\Pi = \emptyset$, the problems above reduce to the standard ones considered in the literature [**?**].

Example 3. Consider the set $\Pi_0 = \{ \neg(\{b\} \rightsquigarrow c), \neg(\{b\} \rightsquigarrow d)\}$ of precedence constraints, stating that b and d are "parallel" activities. Then, consider the trace $t_0 = abcde$ of Example 1 and the graphs in Figure 1. Note that $\mathcal{G}_2$ is a model of Π_0, while $\mathcal{G}_0$ and $\mathcal{G}_1$ are not as they violate the constraint $\neg(\{b\} \rightarrow d)$. Thus, $\mathcal{G}_2$ is a solution to DG-Mining (on input $\{t_0\}$ and Π_0). In fact, it is also solution to Acyclic-DG-Mining. $\lhd$

3.2 Dependency Graph Mining

The problems defined above comprise a learning task (dependency graph mining) and a reasoning task (to check whether a graph satisfies precedence constraints). In fact, we next show that even the learning task can be *declaratively* formulated in terms of reasoning about precedence constraints, thus defining a common framework where the two tasks are synergically combined and might be simultaneously carried out. The basic idea is to characterize the notion of support (in Definition 1) in terms of precedence constraints.

Definition 3 (From traces to specifications) Let L be a log, and for each trace $t[1]...t[n] \in L$, let $\pi(t) = \{ \{t[1], ..., t[i-1]\} \rightarrow t[i] \mid 1 < i \leq n \}$. Then, the set $\pi(L)$ of the precedence constraints associated with L is $\bigcup_{t \in L} \pi(t)$. $\square$

Intuitively, we just state that each activity in the trace t can be directly reached be at least one of its predecessors in t. This suffices to precisely characterize the notion of dependency graph, as illustrated below. We start with the case of processes with no loops.

Proposition 4 *Let L be a log where no trace contains multiple occurrences of the same activity. Let $\mathcal{G}$ be a graph (resp., acyclic graph) over $\mathcal{A}(L)$, and Π be a set of precedence constraints over $\mathcal{A}(L)$. Then, the followings are equivalent:*

(1) $\mathcal{G}$ *is a solution to* DG-Mining *(resp.,* Acyclic-DG-Mining*) on input* L *and* Π*.*
(2) $\mathcal{G} \models \pi(L) \cup \Pi$*.*

Proof. Recall first that $\mathcal{G}$ is a solution to DG-Mining and Acyclic-DG-Mining if $\mathcal{G} \vdash L$, as the notion of folding is immaterial for logs where no trace contains multiple occurrences of the same activity (see Definition 2).

(1)⇒(2). Assume now that *(1)* holds, i.e., $\mathcal{G} \vdash L$ and $\mathcal{G} \models \Pi$. In particular, for each trace $t[1]...t[n] \in L$, there is a subgraph $\mathcal{I}$ of $\mathcal{G}$ such that $\mathcal{I} = (V, E)$ is a process instance over $\mathcal{A}(L)$ and $\mathcal{I} \vdash t$. Let $i \in \{2, ..., n\}$, and notice that there is a path from $t[1]$ in $t[i]$, by definition of process instance. It follows that there is an edge of the form $(t[j], t[i]) \in E$. If $j < i$, then we conclude that the constraint $\pi(t)$ is satisfied by $\mathcal{G}$. Otherwise, it must be the case that $j > i$. However, this is impossible as t is a topological sort of $\mathcal{I}$. Hence, $\mathcal{G} \models \pi(t)$, for each trace $t \in L$. Thus, $\mathcal{G}$ is also a model for $\pi(L)$, and hence $\mathcal{G} \models \pi(L) \cup \Pi$.

(2)⇒(1). To complete the proof, assume now that *(2)* holds. We have to show that $\mathcal{G} \vdash L$ holds. Let $\mathcal{G}$ be the graph (V, E). Let $t[1]...t[n]$ be a trace in L, and let $\mathcal{G}_t = (V_t, E_t)$ be the graph such that $V_t = \mathcal{A}(\{t\})$ and $E_t = \{(t[i], t[j]) \in E \mid i < j\}$. Of course, $\mathcal{G}_t$ is acyclic, and t is actually a topological sort of $\mathcal{G}_t$ by construction. We now claim that each activity in $t[i] \in V_t \setminus \{t[1]\}$ can be reached from $t[1]$. This is shown by induction on the index $i > 1$. In the case case where $i = 2$, $(t[1], t[2])$ must belong to $\mathcal{G}_t$ in order to satisfy the constraint $\{t[1]\} \rightarrow t[2]$ in $\pi(t)$. Then, assume that activities $t[2], ..., t[i-1]$ 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 $t[1]$ as well. It follows that $\mathcal{G}_t$ is a process instance over $\mathcal{A}(L)$ such that $\mathcal{G}_t \vdash t$, for each trace $t \in L$. That is, $\mathcal{G} \vdash L$. □

Example 4. Let Π_0 be the set of constraints defined in Example 3, and let $\pi(abcde)$ be the set of constraints associated with the trace *abcde* in Example 1. Combining the two sets of constraints in the novel set $\Pi_0' = \pi(abcde) \cup \Pi_0$, we have that $\mathcal{G}_2$ is a model of Π_0'. Thus, by Corollary 4, $\mathcal{G}_2$ is a dependency graph for $\{abcde\}$ and satisfy Π_0. ◁

In the case of arbitrary logs, the mapping is established via the concept of folding as a simple extension of the above result.

Corollary 5 *Let* L *be a log,* $\mathcal{G}$ *be a graph over* $\mathcal{A}(L)$*, and* Π *be a set of precedence constraints over* $\mathcal{A}(L)$*. Then, the followings are equivalent:*

(1) $\mathcal{G}$ *is a solution to* DG-Mining *on input* L *and* Π*.*
(2) $\mathcal{G} \models \Pi$ *and* $\mathcal{G}$ *is the folding of a graph* $\bar{\mathcal{G}}$ *such that* $\bar{\mathcal{G}} \models \pi(\bar{L})$*.*

Proof. In the light of Definition 2, we need to show that: $\mathcal{G}$ *is a folding of* $\bar{\mathcal{G}}$ *such that* $\bar{\mathcal{G}} \vdash \bar{L}$ *and* $\mathcal{G} \models L \Leftrightarrow \mathcal{G}$ *is a folding of* $\bar{\mathcal{G}}$ *such that* $\bar{\mathcal{G}} \models \pi(\bar{L})$ *and* $\mathcal{G} \models \Pi$*.* Then, the result immediately follows by applying Proposition 4 on the log $\bar{L}$. □

3.3 Complexity Analysis and Qualitative Restrictions

We now turn to study the computational complexity of the the problems DG-Mining and Acyclic-DG-Mining, which is a necessary step for developing effective algorithms for their solution. In the analysis, we take into account various qualitative properties regarding the kinds of constraint being allowed, by tracing the tractability frontier for w.r.t. them.

Let $\mathtt{S}$ be a subset of the following set of symbols $\{\rightarrow, \rightsquigarrow, \not\rightarrow, \not\rightsquigarrow\}$. Moreover, let $\mathcal{C}[\mathtt{S}]$ denote all the possible sets of constraints that can be built over an underling set $\mathcal{A}$ of activities such 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 $\mathcal{C}[\mathtt{S}]$). Eventually, let us denote by DG-Mining[$\mathtt{S}$] and Acyclic-DG-Mining[$\mathtt{S}$] the restrictions of the two problems over any set of precedence constraints Π such that $\Pi \subseteq \mathcal{C}[\mathtt{S}]$.

Then, our results can be summarized as it is stated next (see also Figure 2)—for the sake of readability, the proof is deferred to the Appendix.

Theorem 6 *The following dichotomies hold:*

- *If* $\mathtt{S} \subseteq \{\not\rightsquigarrow\}$*, then* Acyclic-DG-Mining[$\mathtt{S}$] *is feasible in* P*. Otherwise, the problem is* NP*-hard.*
- *If* $\mathtt{S} \subseteq \{\rightarrow, \rightsquigarrow, \not\rightarrow\}$*, then* DG-Mining[$\mathtt{S}$] *is feasible in* P*. Otherwise, the problem is* NP*-hard.*

Note that the decision version of the problems DG-Mining[$\mathtt{S}$] and Acyclic-DG-Mining[$\mathtt{S}$] can be easily shown to be in NP, no matter of $\mathtt{S}$. Indeed, as the size of any solution $\mathcal{G}$ (resp., and of a graph $\bar{\mathcal{G}}$ of which $\mathcal{G}$ is a folding) is polynomially bounded, we have just to prove that deciding whether $\mathcal{G}$ is actually a solution is feasible in polynomial time. In fact, by Corollary 5 (resp., Proposition **??**), this can be reduced to verify whether $\mathcal{G}$ (resp., $\bar{\mathcal{G}}$) is a model of a certain set of precedence constraints, which is of course in P.

4 Cost-Based Process Mining

In many applications, computing an arbitrary dependency graph is not enough. Indeed, one might often like to focus on those graphs that are minimal w.r.t. some (reflexive and transitive) order "$\preceq$" over the set of all the possible graphs.

Formally, a solution $\mathcal{G}$ to DG-Mining (resp., Acyclic-DG-Mining) is said $\preceq$-minimal if, for each other solution $\mathcal{G}'$ to DG-Mining (resp., Acyclic-DG-Mining), it holds that $\mathcal{G} \preceq \mathcal{G}'$. The above problems, where moreover $\preceq$-minimal solutions are to be computed, will be hereinafter denoted by Acyclic-DG-Mining$_{\preceq}$ and Acyclic-DG-Mining$_{\preceq}$, respectively.

Noticeable Examples. Let $\mathcal{A}$ be a set of activities. Assume that a *weighting function* $w : \mathcal{A} \times \mathcal{A} \mapsto \mathbb{R}$ is given, which associates a value $w(e) \in \mathbb{R}$ with each element $e \in \mathcal{A} \times \mathcal{A}$. Let $\mathcal{G} = (V, E)$ and $\mathcal{G}' = (V', E')$ be two directed graphs over $\mathcal{A}$. Then, we write $\mathcal{G} \sqsubseteq_w \mathcal{G}'$ if, and only if, $\sum_{e \in E} w(e) \leq \sum_{e' \in E'} w(e')$. Thus, we look for graphs having associated the minimum total weight for their edges.

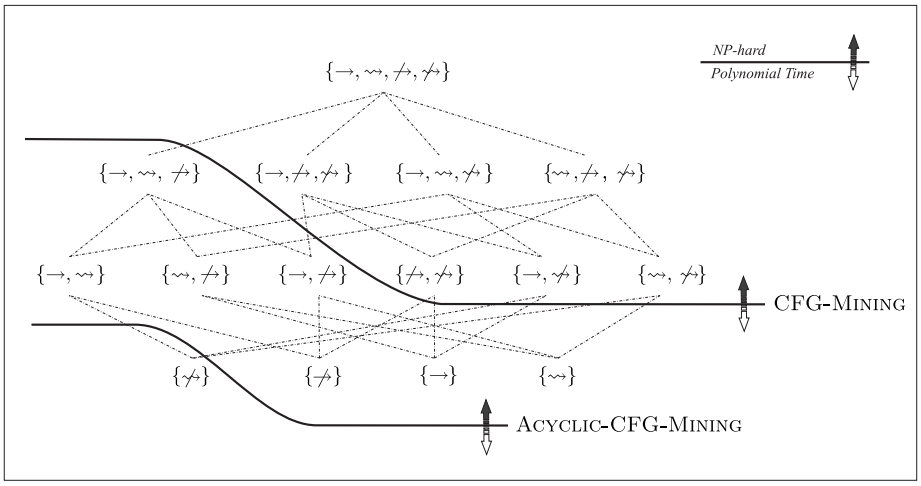

Fig. 2. Tractability Frontiers. A set $\mathtt{S} \subseteq \{\rightarrow, \rightsquigarrow, \not\rightarrow, \not\rightsquigarrow\}$ above (resp., below) the frontier means that the corresponding problem is NP-hard (resp., in P) on the class $\mathcal{C}[\mathtt{S}]$.

As a very simple weighting function, one might consider the constant function $\mathbf{1} : \mathcal{A} \times \mathcal{A} \mapsto \{1\}$ assigning unitary weight to each element in $\mathcal{A} \times \mathcal{A}$. Then, $\sqsubseteq_{\mathbf{1}}$-minimal solutions are constituted by the minimum possible number of edges.

In practice, however, functions adopted in process mining applications assign weights to the various edges based on some statistical information that can be gathered from the input log L, and then filter those edges whose weights are below some given threshold. Let $\mathbf{h} : \mathcal{A} \times \mathcal{A} \mapsto \mathbb{R}$ be a heuristic function expressing how likely is the existence of an edge, with $0 \leq \mathbf{h}(e) \leq 1$, for each $e \in \mathcal{A} \times \mathcal{A}$. Then, the weighting function induced by $\mathbf{h}$, denoted by $\mathbf{h}_{\sigma,M}$ (where $0 \leq \sigma \leq 1$ is a real number and $M > 1$ is real number), is such that:

$$\mathbf{h}_{\sigma,M}(a_i, a_j) = \begin{cases} 1 - \mathbf{h}(a_i, a_j) & \text{if } \mathbf{h}(a_i, a_j) > \sigma \\ M & \text{otherwise} \end{cases}$$

For example, inspired to [48, 47, 2], we can define the function $\mathbf{hm}_{\sigma,M}$ based on the heuristics $\mathbf{hm}(a_i, a_j) = D(a_i, a_j)/|\{t \in L \mid a_i = t[k]\}|$, where

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

and where $0 < \delta < 1$ is a real number. In words, for each trace $t[1]...t[n]$, $D(a_i, a_j)$ is incremented of a factor δ^{k-h-1} if a_i occurs $k - h$ positions before a_j, and decremented of the same factor (in absolute value) if a_i occurs $k - h$ positions before a_j. Note that both the positive and the negative factors exponentially decrease at the growing of the distances between a_i and a_j in the traces.

Example 5. Consider again the setting of Example 1. Then, it is easily seen that $\mathcal{G}_3 \not\sqsubseteq_{\mathbf{1}} \mathcal{G}_2$ and $\mathcal{G}_3 \not\sqsubseteq_{\mathbf{1}} \mathcal{G}_1$. Moreover, note that $\mathcal{G}_1 \sqsubseteq_{\mathbf{1}} \mathcal{G}_2$ and $\mathcal{G}_2 \sqsubseteq_{\mathbf{1}} \mathcal{G}_1$ hold (i.e., these graphs are equivalent w.r.t. the given order).

Consider the log formed by the trace t_0 only. Then, the graph $\mathcal{G}_1 = (V_1, E_1)$ is such that $\sum_{e \in E_1} \mathbf{hm}_{0,M}(e) = 4 - 4 \times \delta^0 = 0$, while the graph $\mathcal{G}_2 = (V_2, E_2)$ is such that $\sum_{e \in E_2} \mathbf{hm}_{0,M}(e) = 5 - 3 \times \delta^0 - 2 \times \delta^1 = 2 \times (1 - \delta)$. It follows that $\mathcal{G}_1 \sqsubseteq_{\mathbf{hm}_{0,M}} \mathcal{G}_2$, no matter of M (and δ). $\lhd$

Complexity. Hardness results for the problems DG-Mining and Acyclic-DG-Mining immediately entail the intractability of computing solution that optimize some given minimality condition. However, it has to be addressed the question about whether focusing on $\preceq$-minimal models makes the problem any harder. The result below evidences that this is actually the case, and that intractability emerges even in absence of further constraints.

Theorem 7 DG-Mining$_{\preceq}[\emptyset]$ *and* Acyclic-DG-Mining$_{\preceq}[\emptyset]$ *are* NP*-hard, even if* $\preceq$ *is the order* $\sqsubseteq_{\mathbf{hm}_{\sigma,M}}$.

Proof. Let $S = \{e_1, ..., e_n\}$ be a set of items, and let $C = \{c_1, ..., c_m\}$ be a collection of subsets of S, i.e., $c_i \subseteq S$ for each $1 \leq i \leq m$. A hitting set for C over S is a subset $S' \subseteq S$ such that $S' \cap c_i \neq \emptyset$, for each $1 \leq i \leq m$. Computing a hitting set with minimum cardinality is a well-known NP-hard problem, even if each subset $c_i \in C$ contains exactly three elements [15].

Based on S and C, we build the log $L(S, C) = \{a_s a_{h_1} a_{h_2} a_{h_3} a_t, a_s a_{h_1} a_{h_3} a_{h_2} a_t, a_s a_{h_2} a_{h_1} a_{h_3} a_t, a_s a_{h_2} a_{h_3} a_{h_1} a_t, a_s a_{h_3} a_{h_1} a_{h_2} a_t, a_s a_{h_3} a_{h_2} a_{h_1} a_t, \mid c_i = \{h_1, h_2, h_3\} \in C\}$ over the activities $\mathcal{A}(L) = S \cup \{a_s, a_t\}$. Observe now that $\mathbf{hm}(a_s, a_t) = \delta^3$ and that $\mathbf{hm}(a_h, a_t) = (2 + 2 \times \delta + 2 \times \delta^2)/6$, for each $a_h \in S$. Let $\sigma = \delta^3$ so that $\mathbf{hm}_{\sigma,M}(a_s, a_t) = \delta^3 = M$, and define $M = 3 \times m \times (2 + 2 \times \delta + 2 \times \delta^2)/6 + 1$ so that $\mathbf{hm}_{\sigma,M}(a_s, a_t) > 3 \times m \times \mathbf{hm}_{\sigma,M}(a_h, a_t)$, for each $a_h \in S$.

Let $\mathcal{G} = (V, E)$ be a $\sqsubseteq_{\mathbf{hm}_{\sigma,M}}$-minimal dependency graph with $V = \mathcal{A}(L(S, C))$ such that $\mathcal{G} \vdash L$. By construction of $L(S, C)$ and of the weighting function, E is such that $(a_s, a_t) \notin E$ and $E \supseteq \{(a_s, a_h) \mid a_h \in S\}$. Thus, for each subset $c_i = \{h_1, h_2, h_3\} \in C$, E must contain at least an edge in the set $\{(a_{h_1}, a_t), (a_{h_2}, a_t), (a_{h_3}, a_t)\}$. No further edge is in E.

From the properties above, it immediately follows that $\{h \mid (a_h, a_t) \in E\}$ is a hitting set for C. Moreover, since $\mathbf{hm}_{\sigma,M}(a_h, a_t) = \mathbf{hm}_{\sigma,M}(a_{h'}, a_t)$, for each pair of distinct sets h and h', we also have that hitting sets with minimum cardinality one-to-one correspond with dependency graphs with minimum weight. Hence, DG-Mining$_{\preceq}[\emptyset]$ is NP-hard. Eventually, the NP-hardness of Acyclic-DG-Mining$_{\preceq}[\emptyset]$ just follows from the observation that dependency graphs associated with hitting sets having minimum cardinality are acyclic. $\square$

5 Process Mining via Constraint Programming

The complexity analysis we have conducted evidenced that polynomial time algorithms are unlikely to exists for the problem of computing models (and minimal

models) of sets of precedence constraints. This bad news calls for very sophisticated solution approaches that perform well in practice, and which possibly integrate heuristic methods to speed up the computation.

In this section, we propose to encode precedence constraints in terms of "standard" *constraints satisfaction problems* (short: CSPs) and to reuse existing constraint programming platforms to compute models of them. Indeed, such platforms have been developed to solve NP-hard problems declaratively specified in terms of CSPs, and embody sophisticated solution algorithms allowing them to scale over large datasets, as their recent application in data mining contexts have already demonstrated [12, 27].

5.1 Constraint Satisfaction Problems

Constraint programming is a declarative programming paradigm where users are just in charge of specifying the problem at hand in terms of a constraint satisfaction problem, instead of formalizing the steps needed for its solution. Constraint programming systems exploit, indeed, general search mechanisms (such as backtracking) enriched with powerful speed-up methods (such as constraint propagation techniques) in order to find a solution starting with such declarative specification [4].

Formally, a CSP *instance* is a triple $(Var, U, \mathcal{C})$, where $Var = \{X_1, ..., X_m\}$ is a finite set of variables, U is a function mapping each variable $X_i \in Var$ to a domain $U(X_i)$ of values, and $\mathcal{C}$ is a finite set of *constraints*. In particular, a constraint $C(X_{i_1}, ..., X_{i_n})$ is a Boolean function from the variables $\{X_{i_1}, ..., X_{i_n}\}$. Let θ be an *assignment* for the CSP, that is, a function mapping each variable to an element of its domain so that $\theta(X_j) \in U(X_j)$ holds for each $X_j \in Var$. We say that θ *satisfies* the constraint $C(X_{i_1}, ..., X_{i_n}) \in \mathcal{C}$ if $C(\theta(X_{i_1}), ..., \theta(X_{i_n}))$ evaluates true. Moreover, we say that θ is a solution to the instance $(Var, U, \mathcal{C})$ if it satisfies all the constraints in $\mathcal{C}$.

Various kinds of constraints are supported by constraint programming systems in the literature. To our ends, we just need to consider two kinds of constraints defined over binary domains. Let $X_1, ..., X_n$ be n variables in Var, let $U(X_i) = \{0, 1\}$, and let $w_1, ..., w_n, \gamma$ be $n + 1$ real numbers. Then,

(1) A *summation constraint* is an expression of the form $\sum_{i=1}^{n} w_i \times X_i \geq \gamma$. The constraint is satisfied by an assignment θ if $\sum_{i=1}^{n} w_i \times \theta(X_i) \geq \gamma$ holds.

(2) A *reified (summation) constraint* is an expression of the form $\sum_{i=1}^{n} w_i \times X_i \geq \theta \leftrightarrow X$, where X is a variables such that $U(X) = \{0, 1\}$. The constraint is satisfied by an assignment θ if $\sum_{i=1}^{n} w_i \times \theta(X_i) \geq \gamma$ holds if, and only, if $\theta(X) = 1$.

Example 6. Consider the problem of placing n queens on (the n rows of) a chessboard so that no queen can capture any other queen. This problem can be formalized as a constraint satisfaction problem $(Var, U, \mathcal{C})$ as follows. The set Var contains a variable $Q_{i,j}$, for each $1 \leq i, j \leq n$, such that $U(Q_{i,j}) = \{0, 1\}$. Intuitively, $Q_{i,j}$ mapped to 1 (resp., 0) denotes that a queen is places (resp., not placed) in the i-th row and the j-th column off the chessboard. The set $\mathcal{C}$ contains

Input: A set Π of precedence constraints over $\mathcal{A} = \{a_1, ..., a_n\}$, and the TYPE $\in$ {ARBITRARY, ACYCLIC} of the problem;
Output: A CSP instance $(Var, U, \mathcal{C})$;

1. **let** $Var = \{e[a_i, a_j], p[a_i, a_j]^\ell, p[a_i, a_k, a_j]^\ell \mid a_i, a_j, a_k \in \mathcal{A}, \ell \in \{1, ..., n\}\}$;
2. **let** $U(X) = \{0, 1\}$ for each $X \in Var$;
3. **let** $\mathcal{C} = \emptyset$;
4. **for each** edge constraint $S \rightarrow a_j$ in Π **do**
   $\mathcal{C} := \mathcal{C} \cup \{\sum_{a_i \in S} e[a_i, a_j] \geq 1\}$;
5. **for each** path constraint $S \rightsquigarrow a_j$ in Π **do**
   $\mathcal{C} := \mathcal{C} \cup \{\sum_{a_i \in S} p[a_i, a_j]^n \geq 1\}$;
6. **for each** negated edge constraint $\neg S \rightarrow a_j$ in Π **do**
   $\mathcal{C} := \mathcal{C} \cup \{e[a_i, a_j] = 0 \mid a_i \in S\}$;
7. **for each** negated path constraint $\neg S \rightsquigarrow a_j$ in Π **do**
   $\mathcal{C} := \mathcal{C} \cup \{p[a_i, a_j]^n = 0 \mid a_i \in S\}$;
8. **if** TYPE=ACYCLIC **then**
   $\mathcal{C} := \mathcal{C} \cup \{p[a_i, a_j]^n + p[a_j, a_i]^n \leq 1 \mid \{a_i, a_j\} \subseteq \mathcal{A}\}$;
9. **for each** pair of distinct activities a_i and a_j **do**
   $\mathcal{C} := \mathcal{C} \cup \{e[a_i, a_j] \geq 1 \leftrightarrow p[a_i, a_j]^1\}$;
   $\mathcal{C} := \mathcal{C} \cup \{e[a_i, a_k] + p[a_k, a_j]^{\ell-1} \geq 2 \leftrightarrow p[a_i, a_k, a_j]^\ell \mid a_k \in \mathcal{A}, \ell \in \{2, ..., n\}\}$;
   $\mathcal{C} := \mathcal{C} \cup \{\sum_k p[a_i, a_k, a_j]^\ell \geq 1 \leftrightarrow p[a_i, a_j]^\ell \mid \ell \in \{2, ..., n\}\}$;
10. **return** $(Var, U, \mathcal{C})$;

Fig. 3. Algorithm PCtoCSP.

the following summation constraints (where equalities are used as a syntactics shorthand, for they are equivalently rewritable in terms of inequalities):

$$\begin{cases} \sum_{i=1}^{n} Q_{i,j} = 1, \forall 1 \leq j \leq n \\ \sum_{j=1}^{n} Q_{i,j} = 1, \forall 1 \leq i \leq n \\ Q_{i,j} + Q_{i',j'} \leq 1, \forall 1 \leq i, j, i', j' \leq n \text{ such that } |i - i'| = |j - j'| \end{cases}$$

As an example, for $n = 3$ the problem does not admit solutions. Instead, for $n = 4$, a solution is the assignment θ where $\theta(Q_{i,j}) = 1$ if, and only if, $(i, j) \in \{(1, 3), (2, 1), (3, 4), (4, 2)\}$. ◁

5.2 CSP Encoding for Precedence Constraints

Now that we have introduced the framework of constraint satisfaction problems, we shall show how the satisfiability of precedence constraints can be restated in terms of CSPs.

Let Π be a set of constraints over a set $\mathcal{A} = \{a_1, ..., a_n\}$ of activities. Figure 3 illustrates an algorithm, named PCtoCSP, to encode Π into a CSP instance $(Var, U, \mathcal{C})$. The instance is such that Var contains an "edge" variable (denoted as $e[a_i, a_j]$) and $n + n^2$ "path" variables (denoted as $p[a_i, a_j]^\ell$ and $p[a_i, a_k, a_j]^\ell$, with $a_k \in \mathcal{A}$ and $\ell \in \{1, ..., n\}$) for each pair a_i and a_j of (not necessarily distinct) activities in $\mathcal{A}$. All variables are defined over the binary domain $\{0, 1\}$. In particular, $p[a_i, a_j]^\ell$ is meant to denote the existence of a path involving ℓ edges at most, while $p[a_i, a_k, a_j]^\ell$ is meant to encode the existence of a path of length ℓ at most and involving the node a_k (possibly coinciding with an endpoint).

Steps 4, 5, 6, and 7 in the algorithm in Figure 3 encode in a straightforward manner edge, path, negated edge, and negated path constraints of Π, respectively. As an example, for each edge constraint $S \rightarrow a_j$ in Π, the constraint

$\sum_{a_i \in S} e[a_i, a_j] \geq 1$ is added to $\mathcal{C}$ stating that at least one of the edge variables from an activity in S to a_j must be mapped to 1. Step 8 is responsible of enforcing acyclicity whenever the input parameter TYPE is ACYCLIC; indeed the constraint states that for each pair of distinct activities a_i and a_j, at most one path variable in $\{p[a_i, a_j]^n, p[a_j, a_i]^n\}$ can be set to 1. Finally, in step 9, it is enforced that $p[a_i, a_j]^n$ can be mapped to 1, if and only if, $e[a_i, a_j]$ is mapped to 1 or there is an intermediate activity a_k with $e[a_i, a_k] + p[a_k, a_j]^{n-1} \geq 1$. Of course, the definition is recursive, and is explicitly unfolded in the encoding as the maximum length of any path is n. The base case is for variables of the form $p[a_i, a_j]^1$, whose value coincides with that of the corresponding edge variables.

The crucial property of the transformation in Figure 3 is next pointed out. For a solution θ to the CSP $(Var, U, \mathcal{C})$ computed by PCTOCSP (on input Π and TYPE), let $\mathcal{G}_\theta = (\mathcal{A}, E)$ be the graph such that (a_i, a_j) is in E if, and only if, $\theta(e[a_i, a_j]) = 1$.

Theorem 8 *Let Π be a set of precedence constraints over $\mathcal{A}$, and let $(Var, U, \mathcal{C})$ be the* CSP *instance computed by the algorithm* PCTOCSP *on input Π and* TYPE*. Then:*

- *Let θ be a solution to $(Var, U, \mathcal{C})$. Then, $\mathcal{G}_\theta \models \Pi$. Moreover, if* TYPE=ACYCLIC*, then $\mathcal{G}$ is acyclic.*
- *Let $\mathcal{G}$ be a graph (resp., acyclic graph) such that $\mathcal{G} \models \Pi$. Then, for* TYPE=ARBITRARY *(resp.,* TYPE=ACYCLIC*), there is a solution θ to $(Var, U, \mathcal{C})$ such that $\mathcal{G} = \mathcal{G}_\theta$.*

Proof. Let θ be a solution to $(Var, U, \mathcal{C})$, and let $\mathcal{G} = (\mathcal{A}, E)$ be the graph such that (a_i, a_j) is in E if, and only if, $\theta(e[a_i, a_j]) = 1$. We have to show that $\mathcal{G} \models \Pi$. To this end, given the construction in steps 4, 5, 6, and 7, we have just to check that $\theta(p[a_i, a_j]^\ell) = 1$ with ℓ if, and only if, there is a path in $\mathcal{G}$ from a_i to a_j with ℓ edges at most. In fact, this property trivially holds for $\ell = 1$, and is guaranteed by the inductive construction of the constraints added in step 9. Finally, if TYPE=ACYCLIC, then the constraints added in step 8 guarantee that the graph $\mathcal{G}$ is acyclic.

For the converse, let $\mathcal{G}$ be a graph such that $\mathcal{G} \models \Pi$. Consider the assignment θ such that, for each $1 \leq i, j, k \leq n$: (1) $\theta(e[a_i, a_j]) = 1$ iff (a_i, a_j) is in E; (2) $\theta(p[a_i, a_j]^\ell) = 1$ iff there is a path from a_i to a_j in $\mathcal{G}$ with length ℓ at most; and (3) $\theta(p[a_i, a_k, a_j]^\ell) = 1$ iff there is a path form a_i to a_k with length ℓ at most and passing over the node a_k. Then, it is easily seen that θ satisfies all the constraints in $\mathcal{C}$ (in particular the constraints in step 8 are satisfied if $\mathcal{G}$ is an acyclic graph, and TYPE=ACYCLIC), i.e., that θ is a solution. By construction, $\mathcal{G} = \mathcal{G}_\theta$. □

By combining Theorem 8 and Proposition 4 (on page 7), we get that PCTOCSP is an effective method to discover acyclic dependency graph supporting some given input log and conforming some given high-level specifications.

Corollary 9 *Let L be a log where no trace contains multiple occurrences of the same activity. Let $\mathcal{G}$ be a graph (resp., acyclic graph) over $\mathcal{A}(L)$, and Π be a set of precedence constraints over $\mathcal{A}(L)$. Then, the followings are equivalent:*

Input: A log L, a set Π of precedence constraints over $\mathcal{A}(L) = \{a_1, ..., a_n\}$, and the TYPE of the problem;
Output: A CSP instance $(Var, U, \mathcal{C})$;

1. **let** $(Var_1, U_1, \mathcal{C}_1)$ be the output of PCToCSP($\pi(\bar{L})$,ACYCLIC);
2. **let** $(Var_2, U_2, \mathcal{C}_2)$ be the output of PCToCSP(Π,TYPE);
3. **let** $Var = Var_1 \cup Var_2 \cup \{eProjected[a_i, a_j] \mid a_i, a_j \in \mathcal{A}\}$;
4. **let** $U(X) = \{0, 1\}$ for each $X \in Var$;
5. **let** $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$;
6. **for each** pair of distinct activities a_i and a_j **do**
   $\mathcal{C} := \mathcal{C} \cup \{e[a_i, a_j] \geq 1 \leftrightarrow eProjected[a_i, a_j]\}$;
   $\mathcal{C} := \mathcal{C} \cup \{\sum_{h,k} e[a_i\langle h\rangle, a_j\langle k\rangle] \geq 1 \leftrightarrow eProjected[a_i, a_j]\}$;
7. **return** $(Var, U, \mathcal{C})$;

Fig. 4. Algorithm DG-DiscoveryToCSP.

(1) $\mathcal{G}$ is a solution to DG-Mining *(resp.,* Acyclic-DG-Mining*) on input L and Π.*
(2) $\mathcal{G} = \mathcal{G}_\theta$ where θ is a solution to the CSP *computed by* PCToCSP *on input $\pi(L) \cup \Pi$ with* TYPE=ARBITRARY *(resp.,* ACYCLIC*).*

In order to deal with arbitrary dependency graphs, i.e., not necessarily acyclic ones, we need a slight extension to the algorithm in Figure 3. This extension is the algorithm DG-DiscoveryToCSP illustrated in Figure 4.

The algorithm receives as input a log L, a set Π of precedence constraints and a TYPE. It produces a CSP instances that is obtained as the union of the instances $(Var_1, U_1, \mathcal{C}_1)$ and $(Var_2, U_2, \mathcal{C}_2)$ produced by PCToCSP on input $(\pi(\bar{L})$,ACYCLIC) and (Π,TYPE), respectively—see steps 1-5. Eventually, the set $\mathcal{C}$ of constraints is enriched by defining $e[a_i, a_j] \in Var_2$, for each pair of activities a_i, a_j, as the "projection" of the corresponding variables in Var_2, i.e., $e[a_i, a_j] = 1$ if, and only if, there is a pair of indices h and k such that $e[a_i\langle h\rangle, a_j\langle k\rangle] = 1$, where $e[a_i\langle h\rangle, a_j\langle k\rangle] \in Var_2$ (step 6).

As with the algorithm PCToCSP, for a solution θ to the CSP $(Var, U, \mathcal{C})$ computed by DG-DiscoveryToCSP, let $\mathcal{G}_\theta = (\mathcal{A}(L), E)$ be the graph such that (a_i, a_j) is in E if, and only if, $\theta(e[a_i, a_j]) = 1$. Moreover, let $\bar{\mathcal{G}}_\theta = (\bar{\mathcal{A}}(\bar{L}), \bar{E})$ be the graph such that $(a_i\langle h\rangle, a_j\langle k\rangle)$ is in $\bar{E}$ if, and only if, $\theta(e[a_i\langle h\rangle, a_j\langle k\rangle]) = 1$. Then, the following is easily established.

Proposition 10 *Let L be a log, let $\mathcal{G}$ be a graph over $\mathcal{A}(L)$, and let Π be a set of precedence constraints over $\mathcal{A}$. Then, the followings are equivalent:*

(1) *$\mathcal{G}$ is a solution to* DG-Mining *(resp.,* Acyclic-DG-Mining*) on input L and Π.*
(2) *$\mathcal{G} = \mathcal{G}_\theta$ where θ is a solution to the* CSP *computed by* DG-DiscoveryToCSP *on input L, Π, and* TYPE=ARBITRARY *(resp.,* ACYCLIC*).*

Proof. Note that the constraints in step 6 guarantees that $\mathcal{G}_\theta$ is the folding of the graph $\bar{\mathcal{G}}_\theta$, for each solution θ. Thus, the result follows from the correctness of algorithm PCToCSP (cf. Theorem 8) guaranteeing that $\bar{\mathcal{G}}_\theta \models \pi(\bar{L})$, and by the application of Corollary 5. □

5.3 Constraint Satisfaction Optimization Problems

In some cases, among all the possible solutions to a CSP instance at hand, we are interested in the one optimizing some given criterium. This gives rise to a *constraint satisfaction optimization problem* (short: CSOP) instance, that is, a quadruple $(Var, U, \mathcal{C}, f)$ where $(Var, U, \mathcal{C})$ is the underlying CSP instance, and where f is a function assigning an integer $f(\theta)$ to each solution θ to $(Var, U, \mathcal{C})$. A solution θ to the CSOP instance is then a solution to the CSP instance such that $f(\theta) \leq f(\theta')$, for each other CSP solution θ'.

In several constraint programming systems, the function f can be specified in terms of a linear minimization constraint—again, we next restrict ourselves to the special case of binary domains. Let $X_1, ..., X_n$ be n variables in Var, let $U(X_i) = \{0, 1\}$, and let $w_1, ..., w_n$ be n real numbers. Then, a *linear minimization constraint* is an expression of the form $\sum_{i=1}^{n} w_i \times X_i$. Under this function, the cost of a CSP solution θ is the value $f(\theta) = \sum_{i=1}^{n} w_i \times \theta(X_i)$.

By using linear minimization constraints on top of the rewriting in Figure 4, we may compute $\sqsubseteq$-minimal solutions rather than arbitrary solutions. The correctness is a simple consequence of Proposition 10 and of the semantics of linar minimization constraints.

Corollary 11 *Let L be a log, let $\mathcal{G}$ be a graph over $\mathcal{A}(L)$, let Π be a set of precedence constraints over $\mathcal{A}$, and let $w : \mathcal{A}(L) \times \mathcal{A}(L) \mapsto \mathbb{R}$ be a weighting function. Then, the followings are equivalent:*

(1) *$\mathcal{G}$ is a $\sqsubseteq$-minimal solution to* DG-Mining$_{\sqsubseteq}$ *(resp.,* Acyclic-DG-Mining$_{\sqsubseteq}$*) on input L and Π.*

(2) *$\mathcal{G} = \mathcal{G}_\theta$ where θ is to the* CSOP *$(Var, U, \mathcal{C}, \sum_{a_i, a_j} w((a_i, a_j)) \times e[a_i, a_j]$, and where $(Var, U, \mathcal{C})$ is the* CSP *computed by* DG-DiscoveryToCSP *on input L, Π, and* Type=arbitrary *(resp.,* acyclic*).*

5.4 Implementation Issues and System Prototype

The whole approach discussed in this section has been implemented in a system prototype. Figure 5 offers a high-level view of the prototype, where the points of interaction with the user and the main functional modules are emphasized.

Basically, the system receives two kinds of inputs: a process log and a set of user-specified precedence constraints. In addition to these fundamental information, users are allowed to set a number of parameters to tune both pre-processing steps and the actual search method of a dependency graph model.

Log data are first processed by the *Constraint Extractor* module in order to translate the given traces into a set of precedence constraints (cf. Proposition 10 and Corollary 11), and by the *Dependency Evaluator* module, which supports the computation of the weighting functions of Section 4. The set of precedence constraints can then be refined with the help of the *Constraint Filter* module implementing a number of methods, possibly guided by the weighting function, which are meant to pragmatically restrict the space of admissible dependency

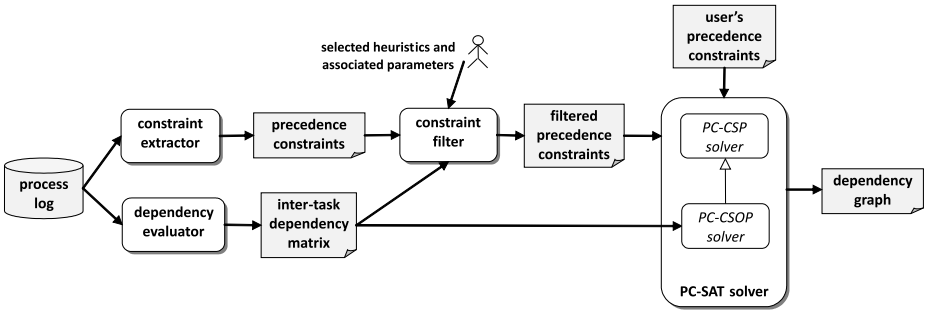

Fig. 5. Conceptual architecture of the system prototype.

graphs. The weighting function and the (possibly refined) set of constraints is then provided as input to the *PC-Sat solver* module, which reformulates the discovery problem as a standard CSP or CSOP instance (cf. Section 5) via the submodules, *PC-CSP solver* and *PC-COSP solver*, respectively, and which is in charge of actually solving them. Of course, these modules takes as an additional input all user-defined precedence constraints, and return the dependency graph.

All modules mentioned above have been implemented in Java, with the exception of the *PC-SAT solver* module, which has been developed in C++.

Solution Algorithms. The computation of the dependency graph that satisfy all given precedence constraints is carried out by *PC-SAT solver* module, which is thus the core of the proposed architecture. This module leverages on the popular C++ constraint programming library *Gecode*[2], by exploiting its constraint solving infrastructure. As a basic solution scheme, *PC-CSP* uses backtracking, while *PC-CSOP* uses a branch-and-bound approach. During the exploration, solution algorithms alternate branching steps, where a value is assigned to some variables as in standard search methods, and constraint propagation steps (which is a peculiarity of constraint programming), where different constraints can be iteratively applied as to shrink the space of the possible dependency graphs and propagate the consequences of choices made in the previous steps. The *PC-SAT solver* module exploits standard Gecode's propagators for all kinds of constraints required in our framework. Instead, ad-hoc branchers have been defined. Indeed, we first branch on the edge variables before considering path variables; and, among these latter, variables of the form $p[a_i, a_j]^\ell$ are always before those having the form $p[a_i, a_k, a_j]^\ell$.

Heuristics. In order to pragmatically reduce the size of the search space and speed-up the computation, a number of heuristics have been made available to the users. We distinguish three kinds of heuristics.

Reducing Redundancy: Users can reduce the level of redundancy in the set of precedence constraints generated, based on two redundancy management

[2] http://www.gecode.org/

policies, which found on different notions of constraint subsumption. According to the first notion, a precedence constraint of the form $S \rightarrow a$ is filtered out if there is another precedence constraint $S' \rightarrow a$ such that $S' \supset S$. In a more caution perspective, instead, we filter out $S \rightarrow a$ if there is additionally a constraint $S'' \rightarrow a$ such that $S'' \subset S$. Notice that this latter (weaker) notion of redundancy allows for handling effectively the presence of skip-like control flow structures, where some synchronizing (i.e. join) activity a can be activated either by an activity in $S \setminus S''$ or, optionally, by an activity in $S \cap S''$.

Closed World Assumption: In order to reduce the size of the search space, users can ask for introducing automatically further constraints, based on a sort of Closed World Assumption (CWA), where an edge (x, y) is not permitted to appear in the model if activity y never follows activity x (directly or indirectly) in any trace of the log. In the case of unfolding, CWA constraints are expressed over real activities, rather than on their unfolded versions.

Constraint Size As a crucial parameter for the efficiency of the solution algorithm is the maximum number of elements occurring in the body of each constraint, a number of heuristics are implemented in order to possibly refine the computation of the set $\pi(t)$ of the precedence constraints associated with the trace $t[1]...t[n]$ (see Definition 3). Specifically, let $\{t[1], ..., t[i-1]\} \rightarrow t[i]$ be a constraint in $\pi(t)$. Then,

- A maximal horizon H can be fixed over the number of predecessor activities that can be involved in a causal relationship. Thus, we remove from the body $\{t[1], ..., t[i-1]\}$ each activity $t[j]$ such that $j < i - H$.
- Two thresholds $\sigma_{abs} \geq 0$ and $\sigma_{r2b} \geq 0$ can be defined acting as lower bounds for the dependency score associated with a predecessor activity. The former is an absolute threshold. Thus, we remove any activity $t[j]$ such that $\mathbf{hm}(t[j], t[i]) < \sigma_{abs}$. The latter is relative to the best score associated with any predecessor activity—subscript *r2b* here stands for "relative to best", like the heuristics defined in [2]. Thus, we remove any activity $t[j]$ such that $\frac{\arg\max_{1<k<i} \mathbf{hm}(t[k],t[i])}{\mathbf{hm}(t[j],t[i])} < \sigma_{r2b}$.
- The maximum number K_{top} of activities that occur in the body can be specified. Thus, the set $\{t[1], ..., t[i-1]\}$ is filtered by just keeping the K_{top} elements with the highest associated dependency score (w.r.t. $t[j]$).

Finally, a series of templates are offered to the user in order to easily formulate some general constraints on the structure of the dependency graph model. These include the specification of maximal/minimal bounds on the total number of edges, and on the degree of each node.

6 Experimental Results

This section is meant to illustrate a series of experiments that were conducted, with the help of the implemented system prototype, in order to assess the validity and the applicability of our approach. In particular, in order to provide the

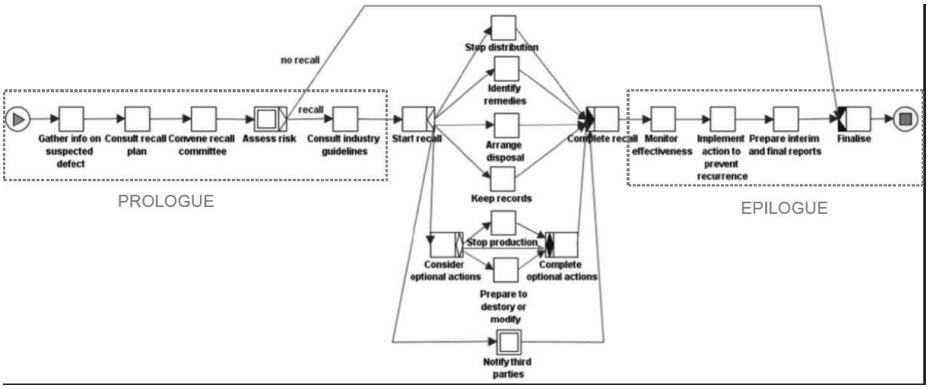

Fig. 6. Workflow schema for a Product-Recall process.

reader with some intuitive evidence of the practical usefulness of our approach, we consider, as a representative application scenario, a process concerning the recall of industry products not satisfying either safety requirements or quality standards.

Figure 6 shows a workflow model, defined in [25] for such a product recall process. The workflow is specified in the *YAWL* modelling language [49] and it is a variant of one of the official case studies (named "YAWL4ProductRecall") of the homonymous workflow management system. By the way, like most workflow modelling languages, YAWL allows for specifying the nature of both fork and synchronization nodes, in addition to formulate basic precedences between activities in the form of edges. Notice that all fork nodes and join nodes in the model of Figure 6 are AND-split and AND-join, respectively.

As pointed out in [25], the need of handling product recall operations, while taking care of traceability and notification issues, arises in a wide variety of real applications. In particular, the workflow model in Figure 6 was built following the guidelines for consumer goods safety recalls established by the bi-national Government agency "Food Standards Australia New Zealand", and it was validated against similar guidelines by US and EU public institutions.

The model describes the main activities to be undertaken by a recall sponsor (usually the manufacturer of a suspect product), in response to a recall incident, possibly triggered, e.g., by consumer complaints, supplier notifications, or failed quality tests. The problem reported has to be investigated carefully and a comprehensive risk analysis must be done, in order to decide whether the product should be recalled or not. In the first case, after consulting industry guidelines, a recall 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 all these recall actions have been completed, a sequence of activities must to be performed, which ends with the completion of the case. These activities range from monitoring the effectiveness of the recall process, to implementing suitable changes to prevent similar problems in the future, to the preparation of reports for regulatory authorities and/or other third parties.

In order to study the effectiveness of our approach in rediscovering activity dependencies from log data, the above workflow model, deployed in system YAWL, was executed a few hundreds of times with the help of 12 voluntary undergraduate students — notice that in such a simulation setting all the workflow tasks were set as manual activities.

Since we are mainly interested in testing our approach against difficult process discovery settings —where the behavior of the analyzed process is highly non-deterministic and is hardly captured in the given sample of log traces— our analysis is restricted only to cases that really involved recall actions (so excluding those where the skipping edge from activity `Assess_risk` to the final one is followed). Moreover, for the sake of readability, a simplified version of the schema in Figure 6 is being considered from now on, where the initial and final (sequential) phases in the process are replaced with two higher level activities, named `PROLOGUE` and `EPILOGUE`, respectively — as graphically illustrated in Figure 6. This simplification is meant to help the reader focus on the most variable part of the recall process, where several activities can be performed concurrently, and can interleave in a great number of ways.

In the following, two different simulation scenarios are discussed in detail, which, despite their simplicity, can help appreciate the benefits of using the technique proposed in the paper when analyzing biased or incomplete log data. The first scenario, referred to in the following as `Scenario 1`, corresponds to an idealistic process management setting, where: (i) the executions of different process instances do not overlap over the time (i.e., a new case can start only after all previous ones have been completed) and (ii) there is no restriction to the assignment of activities to workers (i.e., each activity can be executed by any worker). These two assumptions were removed in the second simulation scenario (named `Scenario 2`), where both workers and activities were partitioned into 5 skill-based classes (for activity assignment purposes) while admitting the concurrent execution of multiple process instances.

Using background knowledge to deal with temporal bias (time shifts) A first kind of violation of the log completeness assumption mentioned above happens when any two mutually parallel activities x and y appear in the same order in all log traces, simply because one of them always finishes after the other, due to a different duration of x and y or of some of their respective predecessor activities. Interestingly, such a situation was empirically found to happen in the first simulation scenario (Scenario 1) considered in our experimentation, where the process was enacted in a sort of clean-room setting, where the working force

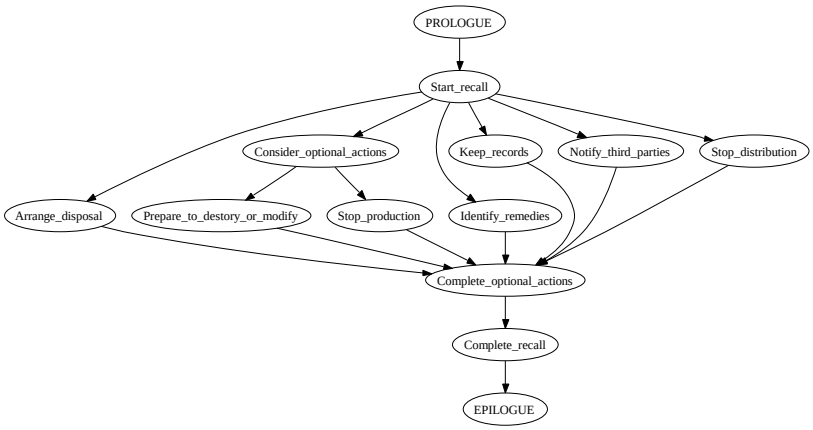

Fig. 7. Dependency graph discovered for the process in Figure 6 out of a temporally-biased log.

available in the organization is always superior to the workload of the product recall process.

Figure 7 shows the optimal (w.r.t. the weighting function **hm**) dependency graph found by our system against a log of 100 traces, produced by enacting the model in Figure 6 according to this setting — recall that the edge from `Assess_risk` to `Finalize` was disabled in both the experiment scenarios considered here. As a matter of facts, it is easy to see that, in this discovered model, activity `Complete Optional Activities` is deemed as dependent on any of the activities `Keep records`, `Identifiy remedies`, `Notify third parties`, `Stop distribution` and `Arrange disposal` — which all appears as predecessors of the former. This is in neat contrast with the real model of the process (cf. Figure 6 where all these activities are in parallel with each other and independent of each other, and it descends from the fact that in the first simulation setting activity `Complete Optional Activities` finished always later than other ones in the other branches of the main forking structure.

Incidentally, the same result was obtained with other classical process mining tools. In particular, Figure 8 shows the control-flow models found by two process discovery plugins available in the popular process mining framework *ProM* [45]: *Heuristics Miner*, implementing the approach in [2], and *Alpha Miner*, implementing the α-algorithm proposed in [42].

Notably, the dependencies in the original control-flow model can be rediscovered by simply augmenting the set of log-driven constraints with the ones in Figure 9, expressing a-priori knowledge on the absence of dependencies between the activity `Complete Optional Actions` and all of the other ones mentioned right above here. This claim is confirmed by Figure 10, which shows the result obtained by our system prototype when using, as input, the same log as before, together with the user-specified constraints reported right above here.

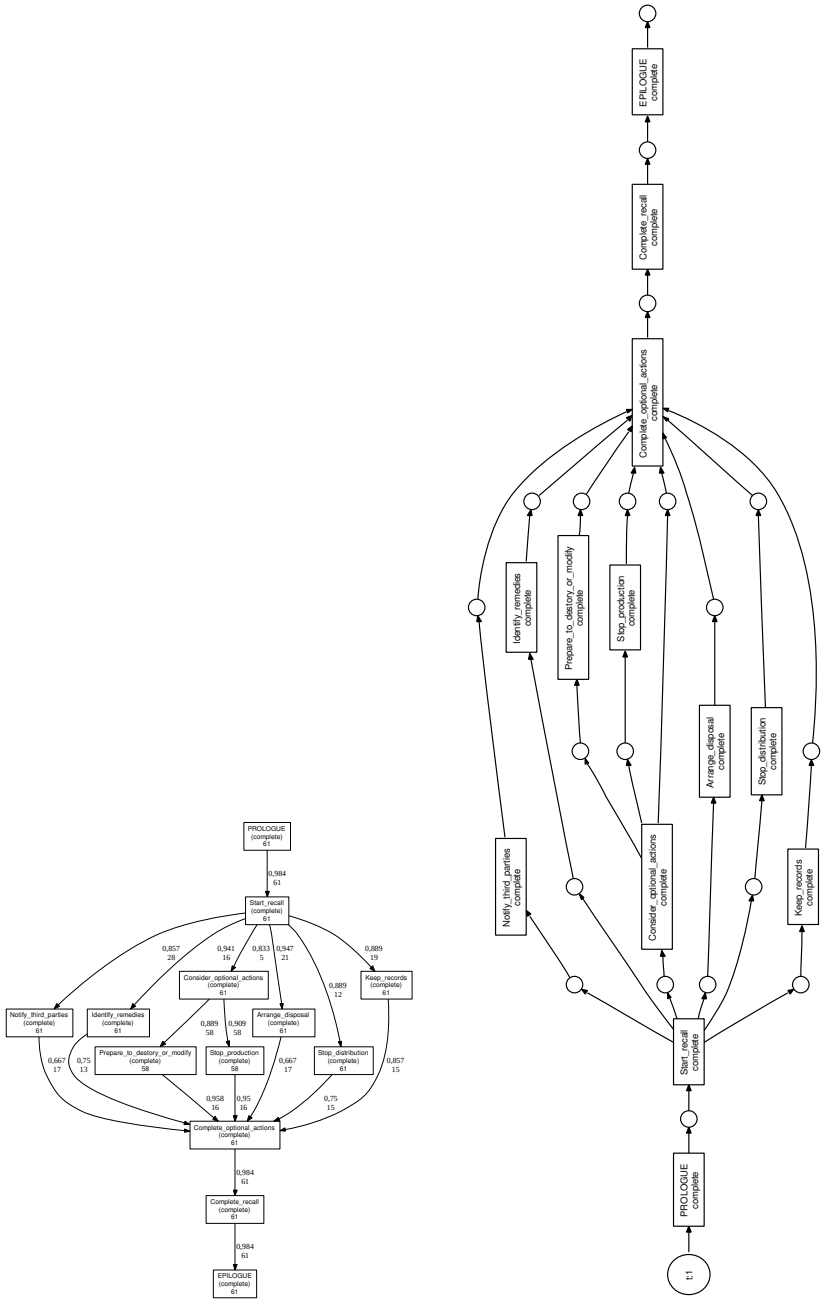

Fig. 8. Control-flow models found in the first simulation scenario for the product recall process, by two classical process discovery algorithms: ProM's plugins *Heuristics Miner* [2] (left) and *alpha* [42] (right).

Identify Remedies $\not\rightsquigarrow$ Complete Optional Actions
Keep Records $\not\rightsquigarrow$ Complete Optional Actions
Stop Distribution $\not\rightsquigarrow$ Complete Optional Actions
Notify Third Parties $\not\rightsquigarrow$ Complete Optional Actions

Fig. 9. Some user-given constraints for improving the quality of the model in Figure 7.

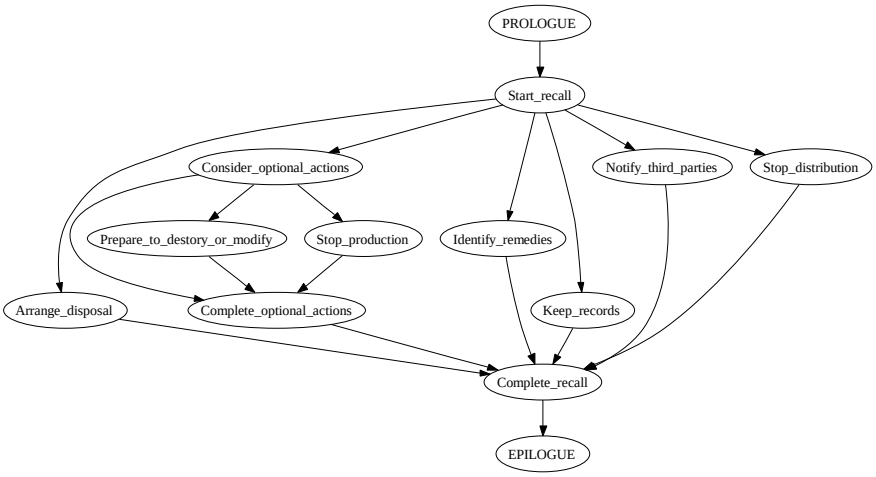

Fig. 10. Dependency graph found in the first simulation scenario with the additional expert-driven constraints in Figure 9.

Using background knowledge to deal with log scarcity In general, log completeness does not hold in any case where the traces in the given log does not capture all the differences in behavior that may occur in the population of all possible process enactments (specially, as far as concerns the presence of a minimal set of different interleavings for mutually concurrent activities). The possibility of exploiting expert-driven constraints, offered by our framework, can be a precious facility in such a situation, which may be quite likely to happen when analyzing the logs of complex real-world processes with many activities and many parallel execution branches.

For the sake of presentation, let us consider a simplified version of the process in Figure 6, where the whole block of activities `Consider Optional Actions`, `Prepare to Destroy or Modify`, `Stop production` and `Complete Optional Actions` is disabled. Even though this latter workflow model does not exhibit a high degree of behavior variability, any process discovery technique might well fail in rediscovering it when provided with an incomplete sample of the process instances. In order to simulate such a situation, we randomly extracted 20 log traces from the traces generated according to `Scenario 2`, and removed from them any occurrence of the "abstracted" activities (namely, `Consider Optional Actions`, `Stop production`, `Complete Optional Actions` and `Prepare to Destroy or Modify`).

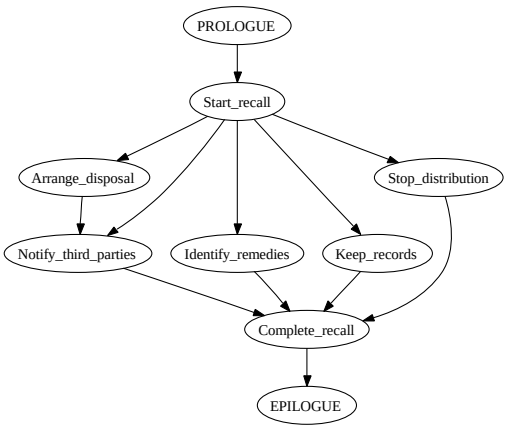

Fig. 11. Dependency graph found against a "small" (20 traces), incomplete, log of the process in Figure 6

The dependency graph discovered out of this small set of traces (still by solving an optimization CSP problem is shown in Figure 11. Clearly this model fails to represent the actual behavior of the original process, due to the wrong dependence between activities `Notify Third Parties` and `Arrange Disposal` , actually belonging to concurrent branches in the process. By the way, the very same set of inter-activity precedence relationships appear in the models found with other process mining tools (including ProM's plugins Heuristics Miner and Alpha Miner mentioned above).

Again, the lack of a complete log can be counterbalanced in this case by providing the CSP solver with a minimal amount of background information. In fact, in this case it was sufficient to provide the system with the sole constraint `Arrange Disposal` $\rightarrow$ `Complete Recall` in order to make it rediscover exactly the activity dependencies in the original process model (cf. Figure 6) — apart from the block of activities that has been disabled in this second simulation scenario.

Analyzing the capability of the approach to deal with incomplete logs In order to assess the capability of our approach to deal with incomplete log data in a deeper and more complete way, we extracted a number of random samples, with different cardinalities, out of the total collection of (236) traces produced in both simulation scenarios.

In order to have a quantitative evaluation for the quality of findings, we can contrast the set D_{out} of causal dependencies discovered towards the set D_{in} of real dependencies existing in the a-priori known process model, by resorting to the classical *F-measure* metrics, 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 really exist in the real model, i.e. $P = \frac{|D_{out} \cap D_{in}|}{|D_{out}|}$, whereas R (standing for *Recall*) is the fraction of real dependencies captured by the mined model, i.e. $R = \frac{|D_{out} \cap D_{in}|}{|D_{in}|}$.

Table 1 summarizes the results obtained by our approach, against different subsets of the log traces, compared with those of two popular process discovery methods [2, 42]. Specifically, in correspondence of each percentage value $p\%$ (with $p = 10, 20, \cdots, 100$), the figure reports the following measures (all averaged over 10 different samples with a $p\%$ of the traces): edge-oriented *F-measure*, total number of dependencies found, and computation time.

Notably the approach proposed in the paper manages to achieve outstanding scores for the *F-measure* over all the samples, and ensures a satisfactory trade-off between precision and recall in the detection of causal dependencies. In particular, the method appears to overcome the competitor ones over smaller logs, where traditional completeness assumption are hardly satisfied. By a finer grain analysis, we can see that, while ensuring apparently good effectiveness results on these logs, the *alpha* algorithm tends to derive from them an overwhelming set of dependencies w.r.t. the real ones. By converse, our approach keeps staying always very close to the actual number (i.e., 19) of dependencies.

	F-measure			*Size*			*Time*—		
trace%	*HM*	*AM*	*Ours*	*HM*	*AM*	*Ours*	*HM*	*AM*	*Ours*
10	0.657	0.731	**0.885**	16, 6	26, 3	18, 5	34, 9	157, 3	1238, 6
20	0.830	0.869	**0.971**	18, 1	24, 1	18, 9	79, 1	216, 9	2670, 3
30	0.893	0.924	**0.981**	18, 4	21, 7	18, 9	112, 2	283, 8	1282, 2
40	0.931	0.943	**0.989**	18, 8	21, 4	19, 0	154, 3	344, 7	5982, 2
50	0.965	0.968	**0.989**	18, 9	20, 3	19, 0	196, 9	413, 1	3576, 6
60	0.979	0.968	**0.995**	19, 2	20, 3	19, 0	233, 1	539, 7	2561, 3
70	0.984	0.990	**0.995**	19, 0	19, 4	19, 0	269, 7	541, 9	2136, 9
80	0.984	0.992	**0.995**	19, 0	19, 3	19, 0	316, 2	617, 5	3370, 6
90	1.000	1.000	1.000	19, 0	19, 0	19, 0	345, 9	678, 9	3472, 7
100	1.000	1.000	1.000	19, 0	19, 0	19, 0	423, 2	710, 9	2597, 1
Average	0.911	0.929	**0.975**	18, 5	21, 3	18, 9	210, 0	444, 7	3108, 8

Table 1. Results obtained by the proposed approach (*Ours*) and by two competitor techniques, for different percentages of traces used as input— *HM* and *AM* here denote the two ProM's plugins *Heuristics Miner* [2] and *Alpha Miner* [42], respectively.

Before leaving this section, it can be interesting to look closer at the results of the tests summarized above, performed against 20% of the given log, for which our approach managed to rediscover a fully complete and correct set of dependencies. A rather imprecise picture of the relationships between process activities is obtained instead by analyzing this incomplete log with classical algorithms [2, 42], as shown in Figure 12.

7 Related Work

7.1 Process Discovery and Dependency Mining

Several process discovery approaches have been proposed in the literature, which both differ in the language used for modelling workflows and in the specific algorithms used to discover it. For example, processes are intuitively represented in [1, 48, 47, 2, 17] via pure directed graphs, which only express precedence relationships. More expressive representations are used instead in other proposals,

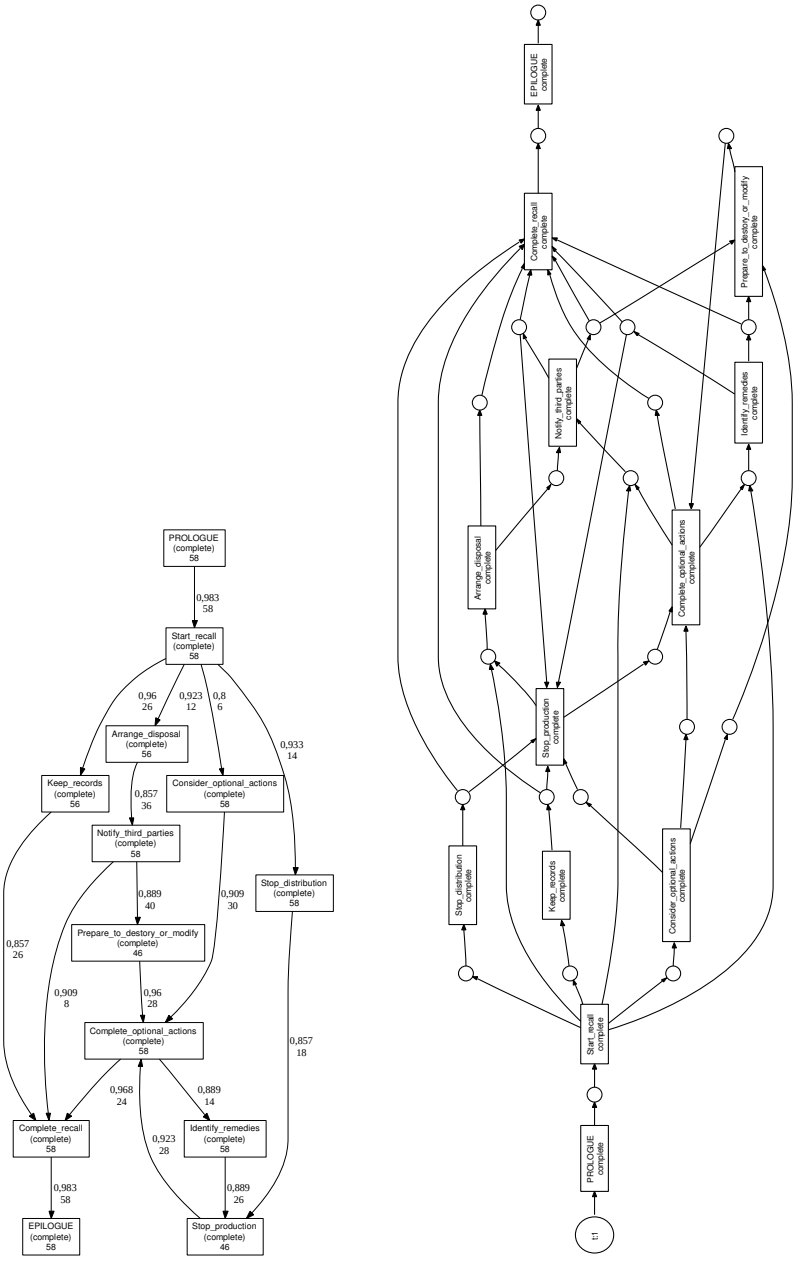

Fig. 12. Control-flow models induced for the product recall process from a 20% sample, by ProM's plugins *Heuristics Miner* [2] (left) and *Alpha Miner* [42] (right).

ranging from expression tree models [33] and block-structured workflow models [19, 20], to special classes of Petri-nets [41, 40, 42, 11].

Despite the variety of approaches proposed in the literature, the discovery of a control-flow model is typically founds on preliminary finding a set of log-driven relations between activities (expressing precedence, parallelism and mutual exclusion relationships), which can be eventually used to infer a more expressive process model. In particular, estimating causal dependencies among the activities is a fundamental sub-problem, particularly challenging in the case of concurrent processes, and in presence of noise.

When mining such dependencies, most of the approaches in the literature rely (implicitly or explicitly) on some suitable notion of *log completeness*, in that they assume that the given log traces are sufficient to recognize all activity dependencies, and to eventually reconstruct the actual structure of a process. In particular, some classical works, including [42], assume that all pairs of activities that are directly linked by a causal dependency, in the unknown process model, must appear consecutively in one log trace at least. Notably, under this assumption, the approach in [42] was proven to be correct, i.e., it precisely reconstructs the process model, provided that the model enjoys certain properties (namely, it is a so-called *structured workflow (SWF)* net). In more details, a basic binary relation, denoted by $>$, is derived from the log to represent immediate-succession: given two activities a, b, $a > b$ holds iff a occurs immediately before b in some log trace – incidentally, the log-completeness notion in [42] amounts to assume that any actual activity dependency is reflected in relation $>$. This relation is used as a basis for deriving causal dependencies —a is deemed as causing b if $a > b$ and not $b > a$ — and for eventually detecting branching and choice patterns – assuming that both a and b depend on the same activity, they are mutually concurrent if both $a > b$ and $b > a$, whereas they are mutually exclusive if neither $a > b$ nor $b > a$. Clearly, such an approach is very likely to fail in real-world scenarios, where the logs often are incomplete (so that no evidence is provided for some pairs of mutually dependent activities) and/or noisy (so that fake dependencies may be derived from erroneous records).

Both issues are faced heuristically in [48, 47, 2], by introducing some degree of fuzziness in the estimation of log-driven ordering relationships. Basically, the approach relies on calculating simple frequency measures, prior to eventually derive these relationships. In particular, for each pairs of activities, say a and b, a causality counter $\#(a \rightarrow b)$ is computed by considering all the (not necessarily consecutive) occurrences of these activities in each log trace. More specifically, every time a appears before b, with n intermediate activities, a contribution δ^n is added to $\#(a \rightarrow b)$ and subtracted from $\#(b \rightarrow a)$. Any causality counter $\#(a \rightarrow b)$ is then normalized by dividing it by the minimum between the frequencies of activities a and b in the entire log. Clearly, using such an exponential weighting scheme is meant to give more relevance to closer occurrence pairs, in the estimation of causal dependencies between activities – i.e. the closer two activities occur in log traces, the more likely they depend on each other, provided that they (almost) always appear in the same order. In order to find reliable causal dependencies against noisy logs, two significance thresholds N and θ can be used. Specifically, a causal link between two any activities a and b is eventu-

ally created if: *(i)* the corresponding causality scores is higher than N; and : *(ii)* there are at least θ traces where a directly precedes b and the opposite does not hold — by the way, the latter criterion implicitly relies on the log completeness notion of [42]. Strong causal dependencies can be directly shown to the user, in the form of a dependency graph model (named "Heuristics Net"), or can be exploited to derive a WF net, similarly to[42].

A variant of this approach is presented in [2], where the user can require the dependency graph to be connected ("All-activities-connected" heuristics) — i.e., any activity, but starting and final ones, has at least one predecessor and at least one successor. Specifically, for each not-starting activity a, the approach in [2] selects the best predecessor, as the one having the highest dependency score towards a; similarly, if a is not final, a dependency link is also drawn from a to the activity a' such that the dependency score (a, a') is maximal. In order to allow for an activity to have multiple predecessors/successors, a further heuristics is introduced, which consists in selecting also a predecessor/successor with a lower dependency score, provided that it is close enough to the maximum — a "relative-to-best" threshold is used to this end, as an upper bound to the relative distance between the score of a candidate and the maximal one.

It is worth noticing that our approach can take advantage of heuristics-based log-driven dependency scores like those in [48, 47, 2] — function **hm** in Section 4 implements indeed the same exponential weighting scheme. However, these scores are only used as a guide in the search of an optimal dependency graph by way of constraint programming techniques. Moreover, in order to cope with noisy data, our approach can also employ analogous relevance thresholds — namely, σ_{abs} and σ_{r2b}, which are similar to parameters N and relative-to-best in [2], respectively. These thresholds are meant to possibly remove unlikely dependencies from log-driven precedence constraints, and prune portions of the search space. Such a pruning can be made stronger by introducing negative CWA constraints, ensuring that, if an activity never precedes another in the log traces, there will not be a path from the former to the latter in the resulting dependency graph. By the way, such a (optional) completeness assumption is looser than the one used in [42] – which prevents an edge between two activities that do not directly follows one another in some trace.

7.2 Declarative (constraint-based) process modelling

Activity dependencies (both control-flow and data-flow) are a fundamental kind of information used in all process modeling frameworks, which can be used directly by constraint-based process modelling and process management environments. In the last years there has been a surge of interest towards declarative constraint-based process modelling frameworks, which can turn out particularly suitable for the management of loosely structured collaboration processes[29], as a flexible and intuitive alternative to the usage of traditional procedural (e.g., workflow-oriented) process models. Clearly, in such a context, the possibility of extracting a control-flow model out of historical log data is of great value, in

that it gives a more precise and complete picture of the typical ways of executing such lowly-structured processes. However, it can be important as well to take advantage of background knowledge encoded in constraint-oriented models when learning a new model by way of process discovery techniques. Due to space limitation, only a few constraint-based modelling approaches are described in the following.

In [32] a process modelling framework is presented which provides constraints for specifying ordering, fork, exclusion and inclusion requirements on the execution of process activities. Here authors put emphasis on the declarative modelling paradigm as a means for achieving a high level of flexibility. In particular, the process is represented by combining basic constraints, encoding control-flow dependencies, and partially specified blocks, called "pockets" of flexibility. Such a pocket consists of activities, sub-processes and a set of both order and inclusion constraints, which can be fully specified/implemented at run-time by human end-users. The approach supports verification techniques for the detection of redundant and conflicting constraints, and has been implemented as a part of a workflow management system prototype named *Chameleon*.

A declarative constraint-based language, named *ConDec* (CONstraintDEClarative), has been proposed in [30] in order to support the modeling, enactment, and verification of declarative process specifications. A ConDec model is composed by a set of activities, representing atomic units of work, and of a set of mandatory and/or optional constraints, specifying relationships among activities. Several constraint types are available as templates, which include: existence constraints, relation constraints, negation constraints and choice constraint. Existence constraints are cardinality constraints on activities, specifying how many times an activity can be executed in any process instance. Relation constraints allows to specify ordering and co-occurrence dependencies between activities. Negation constraints are the negative version of relation constraints, which can be used, e.g., to state that the occurrence of some activities exclude some other ones. Choice constraints finally allow for expressing the necessity to execute some activities as a set of alternative choices. All these constraints are formally specified as Linear Temporal Logic (LTL) formulas – LTL is a temporal logic providing, besides classical logical operators, several temporal operators (always, eventually, until and next time). As a result, a ConDec model is a graph-based model, which can be translated into a combination of LTL expressions, each corresponding to a single constraint. A variant of ConDec, named *DecSerFlow*, has been developed for web service domain [28]. Both ConDec and DecSerFlow languages are supported by *Declare* [29] , a workflow management system for designing and enacting processes based on declarative specifications.

CLIMB (Computational Logic for the verIfication and Modeling of Business processes and choreographies) language [8] is another declarative language for specifying and verifying a process model. CLIMB is a subset of the Social Constrained IFF Framework (SCIFF) language, a FOL-based language originally conceived for heterogeneous Multi-Agent Systems. Here, rule-based constraints are used again as a basic means for expressing desired and forbidden execution

patterns. Constraints are imposed on activities in terms of Integrity Constraints (ICs in short), a sort of reactive rules which can mention, in the body, the occurrence of activities (i.e., events) and constraints on their associated variables, in the style of Constraint Logic Programming (CLP). A model in this language is a set of logical ICs in the form of implications.

7.3 Declarative and/or constraint-based process discovery

An open challenge for Process Mining community is to make Process Discovery declarative, allowing the user to easily and declaratively encode background knowledge and to control the inductive bias and language bias of the learning algorithms.

Recently, some works related to the field of Inductive Logic Programming (ILP) show how process discovery can be formulated as an ILP learning task. For example, this is done in [14] by combining ILP classifier induction techniques with partial order planning. Here, the activities of a business process are seen as planning operators with pre-conditions and post-conditions. Given the current definition of activities' pre-conditions and post-conditions, a plan for achieving the business goal is generated and presented to the user, who is in charge of specifying whether each activity in the plan can be really executed. In this way the system collects positive and negative examples for activities executions (events), which can be then used in the learning phase. By interplaying planning and learning techniques, a process model is discovered eventually.

In [7, 5, 3], some ILP-based process discovery approaches are presented, where a partial model of process behavior is discovered, consisting of a set of ICs (as a sort of behavioral rules). In particular, the approach proposed in [7] takes as input a set of execution traces, previously labeled as compliant or not, and produces a set of SCIFF rules which correctly classify them. By ensuring a mapping of SCIFF rules to a declarative process languages such as DecSerFlow or ConDec, these rules are then used to build a model in the corresponding graphical notation (as a possible support to a declarative specification of a business process). The whole process of induction plus translation has been implemented in the *DecMiner* plug-in of ProM. In [5], the discovered process model is a constraint-based probabilistic process model, expressed in Markov Logic [13] — roughly speaking, this is an extension to first-order logics where each formula can be associated with a weight, and can be seen as a set of soft constraints on possible worlds, so that, if a world violates one formula, it is less probable but not impossible. The approach consists in three steps: learning a SCIFF theory (as in [7]), translating it into Markov Logic formulas, and learning formulas' weights through the discriminative weight learning algorithm of [13].

AGNEs (standing for Artificial Generation of Negative Events)[16] is another declarative process mining algorithm which makes use of an ILP classification-oriented learner, with the final aim of building a a Petri-net model. The algorithm consists in four steps: First, a sort of basic temporal constraints (based on frequent patterns and association rules) are extracted from the input log, in

order to capture local dependency, non-local dependency, and parallelism relationships between activities; notably, some of such constraints can be provided directly by domain experts, as a form of background knowledge. Second, the input log and the temporal constraints induced at the previous step are used to generate negative examples, based on some threshold-driven generalization operators, which reason on possible variants of a given activity sequence (owing to the presence of parallel branches and/or of cycles). More precisely, for each prefix of a given log trace, negative events are generated, stating which activities are not allowed to be executed as subsequent step in the trace. Third, by using all input log traces log (as positive examples) together with the artificial negative events described above, a logic program is induced which allows for predicting whether for a given activity a is allowed to occur or not, at a given position of a given sequence. To this aim the multi-relational induction algorithm *TILDE* is employed, which constitutes a first-order generalization of popular for decision tree induction algorithm *C4.5*. As a final step, the logic program produced by TILDE is transformed into a Petri net.

A last approach which is somewhat related to ours, and to the general family of constraint-based process mining, is the one presented in [44], where language-based region theory is applied to process discovery – regarding each log trace as a word and the log itself as a prex-closed language. Under this perspective, a Petri-net model can be searched for the language (whose words correspond to firing sequences of the net), while regarding each language region as a possible place of the net. Starting with the most liberal (and overgeneralized) net, with as many transitions as the process activities and no place, a more refined workflow model can be obtained by iteratively adding a new place to restricting the allowed behavior. A place (and its associated edges) can be chosen in a greedy way, by requiring that is as expressive as possible, i.e., it has a minimum number of incoming edges and a maximum number of outgoing edges. The insertion of a new place can be then faced by solving a system of linear inequations, under Integer Linear Programming. In order to curb the growth of the discovered model [3], the search of places is guided by causal dependencies derived from the log (by using the metrics in [42]). More precisely, at each step, a single log-driven dependency is considered, which expresses a causal linking between two activities, say a and b. Then, an optimal place is searched for, by solving the Integer Linear Programming problem mentioned above, complemented with the additional constraint that the place is linked to a (via an incoming edge) and to b (via an outgoing edge). The approach makes use of an off-the-shelf LP solver, performing a branch-and-bound search. Notably, in [44] the user is allowed to constrain the general structure of the process model, by restricting the search to one among a pre-defined set of Petri-net classes (including SWF-nets, Marked Graphs, Free-Choice nets, Pure nets, Elementary nets). Beside offering the possibility of choosing among a subset of the Petri net classes mentioned right above, the ProM plugin implementing the approach allows the user to enforce

[3] In principle, the number of places may be exponential in the number of transitions (i.e., activities)

finer grain constraints by manually modifying the basic activity dependencies extracted from the log — prior to deriving a novel (refined) workflow model.

7.4 Distinctive features of the proposed approach

A few major aspects of our approach are summarized next:

- According to the general spirit of declarative process mining, our approach allows the user to formulate high-level prior knowledge in the form of positive/negative dependencies (either direct or indirect) for single pairs of activities [4] — but he/she is not required to manually provide the system with a large enough collection of negative examples (as in the classification-oriented learning frameworks like [7, 5, 3, 14]),
- The final result of our approach is a sort of global process model, showing, in a concise and summarized way, the dependencies that are likely to link the activities in a given process log. Moreover, this basic information on activity relationships may be reused (in the place of count-based log-ordering relations) within consolidated approaches to the construction of a more expressive control-flow model (such as, e.g., the Petri-net oriented ones in [40, 42]) This differentiates our approach from the ones in [7, 5, 3], where the aim is to induce a set of constraints, as a sort of business rules capturing parts of the behavior of the process analyzed.
- A number of tunable heuristics (like, e.g., in [48, 47, 16]) – namely frequency threshold σ_{abs}, *all-activities-connected* heuristics and associated relative-to-best threshold (σ_{r2b}) – can be exploited to cope with the presence of noise in the input log.
- **(?)** Some degree of control on inductive bias (and hence on the level of generalization w.r.t. the input log) is given to the user, who can possibly ask for the automated addition of negative CWA constraints. It is worth noticing that this latter kind of artificially generated constraints follows a looser notion of log-completeness than classical ones(cf.[42]). Moreover, since such constraints are not extracted from the log directly, but are derived from positive log-driven constraints (which can abstract over log contents via window-based and threshold-based heuristics), the aggressiveness degree in the pruning of search space can be controlled by the user. Notably, such a feature is absent in classical approaches, while a similar goal is pursued in [16] by allowing the user for controlling somewhat the generation of negative examples.
- A key distinguishing aspect of our approach is that the user can directly act on the optimality criterion, used in the search of a process model, by providing the reasoner with a collection of (possibly semantic/domain -driven) dependency weights between the activities. To the best of our knowledge, such a capability lacks in all existing process discovery approaches, where

[4] By the way, the constraint language currently used does not allow for concisely specifying mutual exclusion or co-occurrence constraints

the optimality criterion is stated in a fixed (non-parametric) manner, and often encoded, implicitly and approximatively, into greedy search heuristics, and the user is not even allowed to express any preference criteria.

8 Conclusion

Current research is rather active in proposing mining techniques supporting even richer modeling languages. However, very few efforts have been spent to analyze the foundational problem of dependency graph discovery. In fact, the vast majority of process discovery approaches extract activity dependencies by resorting to greedy heuristics, and adopt log completeness assumption to restrict the search space, so risking to be ineffective against incomplete, noisy, and/or temporally-biased logs. In order to make the mining of a dependency graph more effective, efficient and robust, we have proposed a *constraint-based* process discovery framework, where a-priori knowledge can be encoded in the form of *precedence constraints*, and the search of dependencies can be stated as a *constraints satisfaction* problem or a *constraints satisfaction optimization* problem. The computational complexity of these problems has been studied in details w.r.t. different types of constraints, and the tractability frontier of them both has been identified. The whole approach has been implemented in a prototype system, which has been tested on different log data. Preliminary results confirmed the validity of the approach and the opportunity of investigating on the development of declarative process discovery tools, capable of taking full advantage of background knowledge and of user preferences/guidance in order to improve both scalability and quality of results.

As to future work, we observe that the whole framework proposed in the paper is essentially *propositional*, for it assumes a simplification of the schema and of the enactments in which many real-life details are omitted. This is a standard assumption in current research in process mining. Therefore, an interesting avenue for further research is to extend make the automatic extraction of dependency weights more semantics-/domain- oriented by both exploiting context information (about, e.g., parameters and functional features of the activities) and/or pre-existing process ontologies. Moreover, we plan to investigate on extending our constraint-based framework towards more expressive control-flow models and execution constraints.

Acknowledgements

We would like to thank Andrea Burattin and Alessandro Sperduti (?) for providing us with the source code of their process log generator [6]. ►**Se mostriamo gli esperimenti su dati sintetici**◄

References

1. R. Agrawal, D. Gunopulos, and F. Leymann. Mining process models from workflow logs. In *Proc. 6th Intl. Conf. on Extending Database Technology (EDBT'98)*, pages 469–483, 1998.
2. A.J.M.M. Weijters, W.M.P. van der Aalst, and A.K. Alves de Medeiros. Process mining with the heuristicsminer algorithm. Technical report, Eindhoven University of Technology, Eindhoven, 2006.
3. Marco Alberti, Marco Gavanelli, Evelina Lamma, Fabrizio Riguzzi, and Sergio Storari. Learning specifications of interaction protocols and business processes and proving their properties. *Intelligenza Artificiale*, 5(1):71–75, 2011.
4. Krzysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, New York, NY, USA, 2003.
5. Elena Bellodi, Fabrizio Riguzzi, and Evelina Lamma. Probabilistic declarative process mining. In *KSEM*, pages 292–303, 2010.
6. Andrea Burattin and Alessandro Sperduti. Plg: A framework for the generation of business process models and their execution logs. In *Business Process Management Workshops*, volume 66 of *Lecture Notes in Business Information Processing*, pages 214–219. Springer Berlin Heidelberg, 2011.
7. Federico Chesani, Evelina Lamma, Paola Mello, Marco Montali, Fabrizio Riguzzi, and Sergio Storari. Exploiting inductive logic programming techniques for declarative process mining. *T. Petri Nets and Other Models of Concurrency*, 2:278–295, 2009.
8. Federico Chesani, Paola Mello, Marco Montali, and Paolo Torroni. Modeling and verifying business processes and choreographies through the abductive proof procedure sciff and its extensions. *Intelligenza Artificiale*, 5(1):101–105, 2011.
9. H. Davulcu, M. Kifer, C.R. Ramakrishnan, and I.V. Ramakrishnan. Logic based modeling and analysis of workflows. In *Proc. of the 17th ACM Symposium on Principles of Database Systems (PODS'98)*, pages 25–33, 1998.
10. Sandra de Amo and Daniel A. Furtado. First-order temporal pattern mining with regular expression constraints. *Data & Knowledge Engineering*, 62:401–420, September 2007.
11. A. K. A de Medeiros, B. F. van Dongen, W. M. P. van der Aalst, and A. J. M. M. Weijters. Process mining: Extending the α-algorithm to mine short loops. Technical report, University of Technology, Eindhoven, 2004. BETA Working Paper Series, WP 113.
12. Luc De Raedt, Tias Guns, and Siegfried Nijssen. Constraint programming for itemset mining. In *Proceeding of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 204–212, New York, NY, USA, 2008. ACM.
13. Pedro Domingos, Stanley Kok, Daniel Lowd, Hoifung Poon, Matthew Richardson, and Parag Singla. Markov logic. In *Probabilistic Inductive Logic Programming*, pages 92–117, 2008.
14. Hugo M. Ferreira and Diogo R. Ferreira. An integrated life cycle for workflow management based on learning and planning. *Int. J. Cooperative Inf. Syst.*, 15(4):485–505, 2006.
15. M.R. Garey and D.S. Johnson. *Computers and Intractability. A Guide to the Theory of* NP*-completeness*. Freeman and Comp., NY, USA, 1979.
16. Stijn Goedertier, David Martens, Jan Vanthienen, and Bart Baesens. Robust process discovery with artificial negative events. *Journal of Machine Learning Research*, 10:1305–1340, 2009.

17. G. Greco, A. Guzzo, L. Pontieri, and D. Saccà. Discovering expressive process models by clustering log traces. *IEEE Trans. on Knowledge and Data Engineering*, 18(8):1010–1027, 2006.
18. Tias Guns, Siegfried Nijssen, and Luc De Raedt. Itemset mining: A constraint programming perspective. *Artif. Intell.*, 175:1951–1983, 2011.
19. J. Herbst and D. Karagiannis. Integrating machine learning and workflow management to support acquisition and adaptation of workflow models. *Journal of Intelligent Systems in Accounting, Finance and Management*, 9:67–92, 2000.
20. J. Herbst and D. Karagiannis. Workflow mining with InWoLvE. *Computers in Industry. Special Issue: Process/Workflow Mining*, 53(3):245ñ–264, 2003.
21. Lei Jia, Renqing Pei, and Dingyu Pei. Tough constraint-based frequent closed itemsets mining. In *Proceedings of the 2003 ACM symposium on Applied computing*, SAC '03, pages 416–420, New York, NY, USA, 2003. ACM.
22. Mohan Kamath and Krithi Ramamritham. Correctness issues in workflow management. *Distributed Systems Engineering*, 3(4):213–221, 1996.
23. G. Keller, M. Nüttgens, and A. W. Scheer. *Semantische Processmodellierung auf der Grundlage Ereignisgesteuerter Processketten (EPK)*. University of Saarland, Saarbrücken, 1992.
24. Dan Klein, Sepandar D. Kamvar, and Christopher D. Manning. From instance-level constraints to space-level constraints: Making the most of prior knowledge in data clustering. In *Proceedings of the Nineteenth International Conference on Machine Learning*, ICML '02, pages 307–314, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.
25. M.T. Wynn, C. Ouyang, A.H.M. ter Hofstede, and C.J. Fidge. Workflow support for product recall coordination. Technical report, BPMcenter.org, 2009.
26. P. Muth, J. Weifenfels, M.Gillmann, and G. Weikum. Integrating light-weight workflow management systems within existing business environments. In *Proc. 15th IEEE Int. Conf. on Data Engineering (ICDE'99)*, pages 286–293, 1999.
27. Siegfried Nijssen, Tias Guns, and Luc De Raedt. 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, pages 647–656, New York, NY, USA, 2009. ACM.
28. Maja Pesic, Dragan Bosnacki, and Wil M. P. van der Aalst. Enacting declarative languages using ltl: Avoiding errors and improving performance. In *SPIN*, pages 146–161, 2010.
29. Maja Pesic, Helen Schonenberg, and Wil M. P. van der Aalst. Declare demo: A constraint-based workflow management system. In *BPM (Demos)*, 2009.
30. Maja Pesic, M. H. Schonenberg, Natalia Sidorova, and Wil M. P. van der Aalst. Constraint-based workflow models: Change made easy. In *OTM Conferences (1)*, pages 77–94, 2007.
31. Vid Podpečan, Miha Grčar, and Nada Lavrač. Semi-supervised constrained clustering: an expert-guided data analysis methodology. In *Proceedings of the 11th Pacific Rim international conference on Trends in artificial intelligence*, PRICAI'10, pages 219–230, Berlin, Heidelberg, 2010. Springer-Verlag.
32. Shazia Wasim Sadiq, Maria E. Orlowska, and Wasim Sadiq. Specification and validation of process constraints for flexible workflows. *Inf. Syst.*, 30(5):349–378, 2005.
33. G. Schimm. Mining most specific workflow models from event-based data. In *Proc. of Int. Conf. on Business Process Management*, pages 25–40, 2003.
34. H. Schuldt, G. Alonso, C. Beeri, and H. Schek. Atomicity and isolation for transactional processes. *ACM Trans. Database Syst.*, 27(1):63–116, 2002.

35. P. Senkul, M. Kifer, and I.H. Toroslu. A logical framework for scheduling workflows under resource allocation constraints. In *Proc. 28th Int. Conf. on Very Large Data Bases (VLDB'02)*, pages 694–702, 2002.
36. Arnaud Soulet and Bruno Crémilleux. Mining constraint-based patterns using automatic relaxation. *Intelligent Data Analysis*, 13:109–133, 2009.
37. Anthony K. H. Tung, Raymond T. Ng, Laks V. S. Lakshmanan, and Jiawei Han. Constraint-based clustering in large databases. In *Proceedings of the 8th International Conference on Database Theory*, ICDT '01, pages 405–419, London, UK, 2001. Springer-Verlag.
38. W. M. P. van der Aalst, J. Desel, and E Kindler. On the semantics of EPCs: A vicious circle. In *Proc. EPK 2002: Business Process Management using EPCs*, pages 71–80, 2002.
39. W. M. P. van der Aalst, A. Hirnschall, and H. M. W. Verbeek. An alternative way to analyze workflow graphs. In *Proc. 14th Intl. Conf. on Advanced Information Systems Engineering*, pages 534–552, 2002.
40. W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow mining: A survey of issues and approaches. *Data & Knowledge Engineering*, 47(2):237–267, 2003.
41. W. M. P. van der Aalst and K. M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.
42. W. M. P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(9):1128–1142, 2004.
43. W.M.P. van der Aalst. The application of petri nets to worflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.
44. J. M. E. M. van der Werf, B. F. van Dongen, C. A. J. Hurkens, and A. Serebrenik. Process discovery using integer linear programming. *Fundamenta Informaticae*, 94:387–412, 2009.
45. B. van Dongen, A. de Medeiros, H. Verbeek, A. Weijters, and W. van der Aalst. The prom framework: A new era in process mining tool support. In Gianfranco Ciardo and Philippe Darondeau, editors, *Applications and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 1105–1116. Springer Berlin / Heidelberg, 2005.
46. Kiri Wagstaff, Claire Cardie, Seth Rogers, and Stefan Schrödl. Constrained k-means clustering with background knowledge. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, pages 577–584, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.
47. A. J. M. M. Weijters and W. M. P. van der Aalst. Rediscovering workflow models from event-based data using Little Thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, 2003.
48. A.J.M.M. Weijters and W.M.P. van der Aalst. Process mining: Discovering workflow models from event-based data. In *Proceedings of the 13th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2001)*, pages 283–290, 2001.
49. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.
50. Unil Yun and John J. Leggett. Wfim: Weighted frequent itemset mining with a weight range and a minimum weight. In *Proceedings of the Eleventh SIAM International Conference on Data Mining*, SDM '05, pages –1–1, 2005.

A Proof of Theorem 6

We start the proof with the Acyclic-DG-Mining problem. In this case, the only tractability result is for a rather narrow class of constraints.

Theorem 12 Acyclic-DG-Mining$[\{\not\rightsquigarrow\}]$ *is feasible in polynomial time.*

Proof. Let L be a log and Π be a set of precedence constraints. We have to decide whether there is an acyclic dependency graph $\mathcal{G}$ for L such that $\mathcal{G} \models \Pi$, i.e., whether there is a graph $\mathcal{G}$ such that $\mathcal{G} \vdash L$ and $\mathcal{G} \models \Pi$. If $L = \emptyset$, then a graph with no edges is a trivial solution. On the other hand, if L contains some trace with multiple occurrences of the same symbol, then no solution exists at all. Thus, assume that $L \neq \emptyset$ is a log for an underlying acyclic process, and let $a_\perp$ be the starting activity of each trace in L. We distinguish two cases.

If Π contains a negated path constraint of the form $\neg(S \rightsquigarrow a)$, with $a_\perp \in S$, then Acyclic-DG-Mining admits no solution. Indeed, let t be a trace containing a, and assume by contradiction that $\mathcal{G}$ is a solution. Then, there is a subgraph $\mathcal{I}$ of $\mathcal{G}$ that is a process instance over $\mathcal{A}(L)$ (and is such that $\mathcal{I} \vdash t$). This is impossible, since a cannot be reached by $a_\perp$ in $\mathcal{G}$ and, hence, in $\mathcal{I}$. Otherwise, i.e., if Π contains no such constraint, let us consider the dependency graph $\mathcal{G} = (V, E)$ such that $E = \{a_\perp\} \times (V \setminus \{a_\perp\})$. Of course, $\mathcal{G}$ is acyclic and we trivially have that $\mathcal{G} \vdash L$ and $\mathcal{G} \models \Pi$, no matter of L and Π. □

The following two hardness results can be established even if the given log L contains no trace. Hence, the intrinsic difficulty is just confined in the problem of finding an acyclic model for a given set of precedence constraints.

Theorem 13 Acyclic-DG-Mining$[\{\rightarrow\}]$ *and* Acyclic-DG-Mining$[\{\rightsquigarrow\}]$ *are* NP*-hard, even if the input log L contains no trace.*

Proof. Recall that deciding whether a Boolean formula in conjunctive normal form $\Phi = c_1 \wedge \ldots \wedge c_m$ over the variables $X_1, \ldots, X_n$ is satisfiable, i.e., deciding whether there exists a truth assignment to the variables making each clause c_j true, is an NP-hard problem. The problem remains NP-hard, even if each clause contains at most three distinct (positive or negated) variables [15]. Thus, in the following, any arbitrary clause c_j is assumed to be of the form $t_{j,1} \vee t_{j,2} \vee t_{j,3}$, where $t_{j,i}$ $(1 \leq i \leq 3)$ is either a variable (e.g., X_h) or a negated variable (e.g., $\neg X_h$), and where $t_{j,1}$, $t_{j,2}$, and $t_{j,3}$ are not necessarily distinct.

Based on Φ, we build the set $\mathcal{A}(\Phi)$ consisting of the clauses and the variables in Φ (viewed as activities). Formally, $\mathcal{A}(\Phi) = \{c_1, ..., c_m, \} \cup \{t_{j,1}, t_{j,2}, t_{j,3} \mid 1 \leq j \leq m\}$. 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 constraint $\{t_{j,1}, t_{j,2}, t_{j,3}\} \rightarrow c_j$;
- For each pair of clauses c_j and $c_{j'}$ such that $t_{j,i} = \neg t_{j',i'}$ for any two indices $1 \leq i, i' \leq 3$, $\Pi(\Phi)$ contains the constraints $\{c_j\} \rightarrow t_{j',i'}$ and $\{c_{j'}\} \rightarrow t_{j,i}$;
- No further constraint is in $\Pi(\Phi)$.

We now claim that: *Φ is satisfiable $\Longleftrightarrow$ there is an acyclic graph $\mathcal{G}$ such that $\mathcal{G} \models \Pi(\Phi)$.*

($\Rightarrow$) Assume that Φ is satisfiable, and let σ be a satisfying truth assignment for the variables $X_1, ..., X_n$. Consider the graph $\mathcal{G} = (\mathcal{A}, E)$ such that:
- For each clause c_j, if $t_{j,i}$ evaluates true in σ, then $(t_{j,1}, c_j) \in E$;
- For each pair of clauses c_j and $c_{j'}$ such that $t_{j,i} = \neg t_{j',i'}$ for any two indices $1 \leq i, i' \leq 3$, the two edges $(c_j, t_{j',i'})$ and $(c_{j'}, t_{j,i})$ are both in E;
- No further edge is in E.

It is immediate to check that $\mathcal{G}$ satisfies all the constraints in $\Pi(\Phi)$. Moreover, we note that $\mathcal{G}$ is acyclic. Indeed, if an edge of the form $(t_{j,i}, c_j)$ occurs in E, then we are guaranteed that there is no clause $c_{j'}$ with an index $1 \leq i' \leq 3$ such that $t_{j,i} = \neg t_{j',i'}$ and $(t_{j',i'}, c_{j'}) \in E$. In fact, this follows from the fact that σ is a satisfying assignment and by construction of E.

($\Leftarrow$) Assume that $\mathcal{G}$ is an acyclic graph satisfying all the constraints in $\Pi(\Phi)$, and let us build a truth assignment σ for Φ. Consider any pair of clauses c_j and $c_{j'}$ such that $t_{j,i} = \neg t_{j',i'}$ for any two indices $1 \leq i, i' \leq 3$. Since $\mathcal{G} \models \Pi(\Phi)$, $\mathcal{G}$ either contains the edge from $t_{j,i}$ to c_j, or it contains the edge from $t_{j',i'}$ to $c_{j'}$. Assume that $t_{j,i} = X_h$. Then, in the former case, we set $\sigma(X_h)$ to true (thereby satisfying c_j), and in the latter case to false (thereby satisfying $c_{j'}$). Note that σ is well defined. Moreover, since for each clause c_j, $\Pi(\Phi)$ contains the constraint $\{t_{j,1}, t_{j,2}, t_{j,3}\} \rightarrow c_j$, the truth assignment σ eventually satisfies all the clauses of Φ.

From this claim and since the reduction is feasible in polynomial time it follows that Acyclic-DG-Mining[$\{\rightarrow\}$] is NP-hard, even if the input log contains no trace. To conclude the proof, we just notice that the salient properties of the reduction are not altered if we replace each edge constraint in $\Pi(\Phi)$ with the analogous path constraint. Hence, Acyclic-DG-Mining[$\{\rightsquigarrow\}$] is NP-hard. □

In the case of negated edge constraints, the hardness can be given only for non-empty logs, for otherwise the graph with no edges is a trivial solution.

Theorem 14 Acyclic-DG-Mining[$\{\not\rightarrow\}$] *is* NP*-hard.*

Proof. The line of the proof is to show that any set of positive edge constraints can be encoded via the constraints associated with a suitably defined log plus a set of negated edge constraints. Then, the result will follow from Theorem 13.

Let $\Pi = \{\{b_i^1, ..., b_i^{k_i}\} \rightarrow a_i \mid i \in \{1, ..., m\}\} \subseteq \mathcal{C}[\{\rightarrow\}]$ be a set of edge constraints over a set $\mathcal{A}$ of activities. Let $a_\perp \notin \mathcal{A}$ be a fresh activity, and for each constraint $\{b_i^1, ..., b_i^{k_i}\} \rightarrow a_i$, let $c_i \notin \mathcal{A}$ be a fresh activity associated to it. Based on Π, we build a log $L(\Pi)$ with traces $t_1, ..., t_m$ such that $t_i = a_\perp c_i b_i^1, ..., b_i^{k_i} a_i$, for each $i \in \{1, ..., m\}$. Moreover, consider the set Π' of negated edge constraints including $\{a_\perp\} \not\rightarrow a$, for each activity $a \notin \{c_1, ..., c_m\}$, and $\{c_i\} \not\rightarrow a$, for each $a \notin \{b_i^1, ..., b_i^{k_i}\}$ and each $i \in \{1, ..., m\}$.

We now claim that: *there is an acyclic graph $\mathcal{G}$ such that $\mathcal{G} \models \Pi \Leftrightarrow$ there is an acyclic graph $\mathcal{G}'$ such that $\mathcal{G}' \vdash L(\Pi)$ and $\mathcal{G}' \models \Pi'$.*

($\Rightarrow$) Assume that $\mathcal{G}$ is an acyclic model of Π. Let $\mathcal{G}'$ be a graph over the activities in $\mathcal{A} \cup \{a_\perp, c_1, ..., c_m\}$ defined as follows. The subgraph of $\mathcal{G}'$ induced over

$\mathcal{A}$ coincides with $\mathcal{G}$. Moreover, $\mathcal{G}'$ contains the edges $(a_\perp, c_i)$, for each $i \in \{1, ..., m\}$, and the edges (c_i, b_i^j) for each $i \in \{1, ..., m\}$ and $j \in \{1, ..., k_i\}$. By construction, $\mathcal{G}' \models \Pi'$. It remains to show that $\mathcal{G}' \vdash L(\Pi)$, or equivalently (by Proposition 4) $\mathcal{G}' \models \pi(L)$. To this end, consider a trace t_i with $i \in \{1, ..., m\}$, and note that $\pi(t_i)$ consists of the constraints: (i) $\{a_\perp\} \to c_i$, (ii) $\{a_\perp, c_i, b_i^1, ..b_i^{j-1}\} \to b_i^j$, for each $1 \leq j \leq k_i$, and (iii) $\{a_\perp, c_i, b_i^1, ..b_i^{k_i}\} \to a_i$. Note that (i) and (ii) are satisfied by the edges originating from the nodes $a_\perp$ and c_i. Moreover, observe that (iii) combined with the negated edge constraints in Π' imply the constraint $\{b_i^1, ..b_i^{k_i}\} \to a_i$, which is in fact a constraint in the original set Π. This constraint is satisfied by $\mathcal{G}$, and thus by construction by $\mathcal{G}'$ as well. It follows that $\mathcal{G}' \vdash L(\Pi)$.

($\Leftarrow$) Assume there is an acyclic graph $\mathcal{G}'$ such that $\mathcal{G}' \vdash L(\Pi)$ and $\mathcal{G}' \models \Pi'$. As $\mathcal{G}'$ is a model of Π', we can again notice that each constraint (iii) $\{a_\perp, c_i, b_i^1, ..b_i^{k_i}\} \to a_i$ is actually equivalent to $\{b_i^1, ..b_i^{k_i}\} \to a_i$. Let $\mathcal{G}$ be the subgraph of $\mathcal{G}'$ induced over the nodes in $\mathcal{A}$. As $\mathcal{G}'$ satisfies $\{b_i^1, ..b_i^{k_i}\} \to a_i$, for each $i \in \{1, ..., m\}$, and since such constraints are defined over $\mathcal{A}$, it follows that $\mathcal{G} \models \Pi$.

In the light of the claim above, the result follows by the NP-hardness of Acyclic-DG-Mining$[\{\to\}]$ (cf. Theorem 13). □

We can now turn to the DG-Mining problems, where arbitrary dependency graphs are considered. In this case, the counterpart of Theorem 12 can be established over a larger class of constraints.

Theorem 15 DG-Mining$[\{\to, \rightsquigarrow, \not\to\}]$ *is feasible in polynomial time.*

Proof (Sketch). Let Π be a set of constraints in $\mathcal{C}[\{\to, \rightsquigarrow, \not\to\}]$ and L be a log. We start by building a graph $\mathcal{G} = (V, E)$ over the nodes involved in Π and L and where an edge (a, a') is in E if, and only if, there is no negated edge constraint $S \not\to a'$ with $a \in S$. Then, we check whether all the other (edge and path) constraints are satisfied by $\mathcal{G}$. If this is not the case, then there is not solution at all. Otherwise, it remains just to check whether $\mathcal{G}$ is the folding of a graph $\bar{\mathcal{G}}$ such that $\bar{\mathcal{G}} \vdash L$. This can be carried out by trying to simulate the enactment of $t = t[1]....t[n]$ over $\mathcal{G}$. For each activity $a \neq a_\perp$, initialize a variable $w_a = 0$, and let $w_{a_\perp} = 1$. We incrementally process each $1 < i \leq n$, and if there is an edge $(a, t[i])$ such that $w_a - w_{t_i} > 0$, then we increment w_{t_i} by 1; otherwise, we stop the process with a failure. It can be shown that if no failure occurred over all the possible traces, then $\mathcal{G}$ is the folding of a graph $\bar{\mathcal{G}}$ such that $\bar{\mathcal{G}} \vdash L$. □

The picture is now easily completed concerning the NP-hardness results, as negated path constraints can be used to enforce acyclicity.

Theorem 16 DG-Mining$[\{\to, \not\rightsquigarrow\}]$, DG-Mining$[\{\rightsquigarrow, \not\rightsquigarrow\}]$, *and* DG-Mining$[\{\not\to, \not\rightsquigarrow\}]$ *are* NP*-hard.*

Proof. We exhibit a reduction to the Acyclic-DG-Mining[$\{\rightarrow\}$] (resp., Acyclic-DG-Mining[$\{\rightsquigarrow\}$], Acyclic-DG-Mining[$\{\not\rightarrow\}$]) problem. Let Π be a set of constraints in $\mathcal{C}[\{\rightarrow\}]$ (resp., $\mathcal{C}[\{\rightsquigarrow\}]$, $\mathcal{C}[\{\not\rightarrow\}]$), and consider the problem of deciding whether there is an acyclic graph $\mathcal{G}$ such that $\mathcal{G} \models \Pi$.

Based on Π, we build the set Π' of constraints including all the constraints in Π, plus the novel constraint $\{a\} \not\rightsquigarrow a$, for each activity a. Of course, Π' belongs to $\mathcal{C}[\{\rightarrow, \not\rightsquigarrow\}]$ (resp., $\mathcal{C}[\{\rightsquigarrow, \not\rightsquigarrow\}]$, $\mathcal{C}[\{\not\rightarrow, \not\rightsquigarrow\}]$). Moreover, the role of the fresh constraints is just to enforce the acyclicity of the desired graph. Indeed, $\mathcal{G} \models \Pi'$ if, and only if, $\mathcal{G} \models \Pi$ and $\mathcal{G}$ is acyclic. The result then follows from Theorem 13 and Theorem 14. □