Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning
Main Track

Logic-based Composition of Business Process Models

Valeria Fionda , Antonio Ielo , Francesco Ricca
Department of Mathematics and Computer Science, University of Calabria, Italy
{valeria.fionda, antonio.ielo, francesco.ricca}@unical.it

Abstract

Process mining is a family of techniques that exploit data collected from process execution to analyze and improve process efficiency, quality, and security. Over the years, many modeling languages have been proposed for process model specification, with different expressiveness, features, and computational properties. We propose a new logic-based declarative formalism, Constraint Formulae, to compose process specifications expressed in heterogeneous process modeling languages without altering their original semantics. We formalize common process mining tasks for Constraint Formulae, study their computational properties, and provide an implementation in Answer Set Programming.

1 Introduction

Process mining aims to extract valuable insights from event logs recorded by information systems to understand how a particular process works, identify areas for improvement and optimize performance (van der Aalst and et al. 2011). During the past few years, process mining found application in many different fields, including financial services (Jans et al. 2011), healthcare (Partington et al. 2015), manufacturing (ER et al. 2018), and IT (Sedrakyan, Weerdt, and Snoeck 2016). Process mining techniques can be grouped into three main categories: discovery, conformance checking, and enrichment. Discovery aims to analyze the structure of a process to identify patterns in the event log and understand the general flow of the process. Conformance checking aims at comparing an event log to a process model for identifying deviations and providing insights into the cause of these deviations. Enrichment is related to the suggestion of changes to be implemented in the process to improve efficiency, reduce costs, or improve overall performance.

In recent years several languages have been proposed to encode process models. Such languages can be broadly classified into imperative and declarative languages. Imperative languages, including Petri nets and BPMN (van der Aalst and et al. 2011), can be used to define process models that represent the precise sequence of tasks and decisions that determine the exact process execution flow. Instead, declarative languages, such as DECLARE (Pesic, Schonenberg, and van der Aalst 2007), allow to define a set of rules and constraints the process must satisfy without specifying how it should be executed.

Each modeling language has precise syntax, semantics, and computational properties that make it more suitable for use in specific circumstances. To the best of our knowledge, today there is no means to specify different parts of the same process using different modeling languages and conveniently and intuitively compose them. In this paper, we fill this gap by proposing and studying a logic-based declarative framework, specific to business process models, to compose process specifications expressed using heterogeneous process modeling languages. In more detail:

- We define the concept of *constraint formulae* as a mechanism to flexibly compose and reason about heterogeneous process specifications (Section 3);
- We recast *conformance checking*, *query checking*, and *discriminative process discovery* (van der Aalst and Carmona 2022; Chesani et al. 2022a) into the constraint formulae framework and introduce the new *trace clustering* task as a generalization of discriminative process discovery (Section 4).
- We depict a clear picture of the computational complexity of the above tasks reinterpreted in the Constraint Formulae framework (Section 5);
- We provide an encoding in Answer Set Programming (Brewka, Eiter, and Truszczynski 2011; Gelfond and Lifschitz 1991; Niemelä 1999) (Section 6) and evaluate empirically a proof-of-concept implementation (Section 7) based on CLINGO (Gebser et al. 2016).

Related Work. Discriminative process discovery is gaining momentum. Indeed, several recent papers studied the problem in the context of declarative process mining. Notable examples are NegDis (Chesani et al. 2022a) and a framework to mine CNF and DNF formulas (Chesani et al. 2022b). Differently from NegDis and CNF/DNF mining our framework does not target only the DECLARE language but it applies to constraints expressed in arbitrary modeling languages. In addition, differently from CNF/DNF mining, which learns formulas by applying a sequence of choices determined by coverage/rejection criteria over a log of negative traces, our framework finds an optimal model within a search space guided by some templates provided by an expert user thus helping results' interpretation.

Strictly related to our approach are the recent application of Answer Set Programming to solve conformance check-

ing, query checking and log generation of data-aware DECLARE models (Chiariello, Maggi, and Patrizi 2022). Differently from such approaches our framework is enough general to be applied to heterogeneous modeling languages, even if it could also be applied to DECLARE constraints only.

2 Preliminaries on Process Modeling

The main goal of process mining is to extract insights from *event logs* (van der Aalst and Carmona 2022). An event log stores in chronological order the *events* that occur while a process is running. Each event in the log contains information about the *activity* that was performed, the time at which it was performed, and the case identifier that allows grouping events related to the same process execution into a *trace*. Furthermore, events can refer to additional information such as the resources that were involved or the cost paid.

In the following, we assume that $\mathcal{A}$ is a given alphabet of symbols, univocally identifying a set of *activities*. A *trace* π over $\mathcal{A}$ is a finite succession $\pi[1]...\pi[l]$ with $\pi[i] \in \mathcal{A}$, for each $i \in \{1, ..., l\}$, and with $l \geq 1$ being its *length*, denoted by $len(\pi)$. A trace represents a sequence of events that occur in a particular process instance. For example, a trace can record the sequence of activities a user performs booking a flight online. A *log* $\mathcal{L}$ is a multi-set of traces, each one corresponding to a different case or process instance.

Process mining tools offer the opportunity to analyze event logs according to various perspectives (van der Aalst and Carmona 2022), such as control flow (i.e., the flow of tasks and activities), time (i.e., timing and frequency of events), resources (i.e., people, systems, and roles) or data. As control flow plays a central role in the development of process-oriented applications, in this paper we will focus on such a perspective.

3 Contraint Formulae

Let $\mathcal{A}$ be a given finite set of symbols that univocally identify the set of activities of a process, $\pi = \pi[1], \ldots, \pi[l]$ with $\pi[i] \in \mathcal{A}$ for each $i \in \{1, \ldots, l\}$ be a trace over $\mathcal{A}$, and $\mathcal{L}$ be a multi-set of traces $\{\pi_1, \ldots, \pi_n\}$ over $\mathcal{A}$.

A common task in process mining is *process discovery*, which is the problem of characterizing the process behaviors contained in a log by providing a suitable process model. To this end, in *declarative process mining*, it is common to use some Boolean conditions that can be tested over traces. Such conditions are usually organized in *templates* that define classes of properties of interest, and *constraints* that are obtained by *instantiating* a template. More formally:

Definition 1 (Constraint)**.** *A constraint c is a binary predicate over traces, that is a function $\mathcal{A}^* \mapsto \{\top, \bot\}$. Given a trace π, if $c(\pi) = \top$ we write $\pi \models c$, else if $c(\pi) = \bot$ we write $\pi \not\models c$.*

In the following, when $\pi \models c$ holds, we will also write that π satisfies c or that c holds in π.

Definition 2 (Template)**.** *A template of arity $k \in \mathbb{N}$ (a k-template) is a function that maps k-tuples of activities to constraints, $\mathcal{A}^k \mapsto (\mathcal{A}^* \mapsto \{\top, \bot\})$. Given a template t, we denote by $ar(t) = k$ the arity of t.*

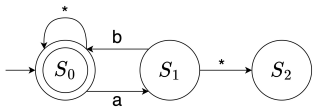


Figure 1: State machine $M_{a,b}$ accepting traces satisfying the constraint of Example 1. Edges labeled with $*$ represent transitions for any symbols in $\mathcal{A}$ for which no other explicit transitions exist.

We assume there exists a finite set $\mathcal{T}$ of templates that can be partitioned into subsets $\{\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \ldots\}$ such that $t \in \mathcal{T}_k$ if and only if $ar(t) = k$.

Definition 3 (Constraint instantiation)**.** *A constraint c is an instance of a template t if there exists a tuple $(a_1, \ldots, a_k) \in \mathcal{A}^k$, where $k = ar(t)$, such that for all $\pi \in \mathcal{A}^*$ we have that $c(\pi) = t(a_1, \ldots, a_k)(\pi)$.*

Example 1. *If we are interested in checking that activity a is always immediately followed by activity b (i.e., if $\pi[i]{=}a$ then $\pi[i{+}1]{=}b$; that is the* DECLARE *pattern* chain-response(a,b)*), we can use the finite state machine $M_{a,b}$ reported in Figure 1. Thus, we can define the constraint* chain-response$_{a,b}$ *such that $\pi \models$* chain-response$_{a,b}$ *if and only if $\pi{\in}L(M_{a,b})$. Such constraint can be generalized to whatever pair of activities (by overloading the notation) by the 2-template* chain-response*: $(x, y) \mapsto M_{x,y}$ with $x, y{\in}\mathcal{A}$. Clearly,* chain-response$_{a,b}$ *is a constraint instantiation of the template* chain-response *with activities a and b.* $\lhd$

The reasoning above can be easily generalized to different types of constraints either declarative (e.g., other DECLARE patterns or LTL$_f$-based constraints), or non-declarative (e.g., based on some imperative modeling languages), or even some procedural constraint definition (e.g., Python programs checking for some conditions).

Example 2. *Suppose we want to check whether three activities a, b and c occur in a trace π the same (unbounded) number of times. Clearly, such constraint cannot be encoded neither by a finite state machine nor by a general LTL$_f$ formula. However, it can be easily checked by a procedure $f(a, b, c)$ (written in whatever programming language) that receives the trace in input and counts the occurrences of the three activities. Such procedure can be seen as a 3-template $f : \mathcal{A}^3 \mapsto (\mathcal{A}^* \mapsto \{\top, \bot\})$.* $\lhd$

We refer to all the constraints that can be instantiated from templates in $\mathcal{T}$ as *atomic constraints*, and we assume that there exists a procedure to evaluate them over arbitrary traces. Atomic constraints are fundamental for defining *constraint formulae*. To this end, let $\mathcal{V} = \mathcal{V}_\mathcal{A} \cup \mathcal{V}_\mathcal{T}$ be a set of variables (disjoint from $\mathcal{A} \cup \mathcal{T}$) where $\mathcal{V}_\mathcal{A}$ represents the set of *activity variables* and $\mathcal{V}_\mathcal{T}$ represents the set of *template variables*. We assume each variable has associated a *domain* $\mathcal{D}$, such that $\mathcal{D}(X) \subseteq \mathcal{A}$ for each $X \in \mathcal{V}_\mathcal{A}$ and $\mathcal{D}(X) \subseteq \mathcal{T}_k$ for each $X \in \mathcal{V}_\mathcal{T}$ and some $k \in \mathbb{N}$. Similarly to $\mathcal{T}$, $\mathcal{V}_\mathcal{T}$ can be partitioned into disjoint subsets $\{\mathcal{V}_{\mathcal{T}_0}, \mathcal{V}_{\mathcal{T}_1}, \ldots\}$ where each $\mathcal{V}_{T_k}$ contains all variables $X \in \mathcal{V}_T$ s.t. $\mathcal{D}(X) \subseteq \mathcal{T}_k$.

The building blocks for defining constraints formulae are *constraint terms*. Each constraint term is a constraint in which, possibly, variables appear.

Definition 4 (Constraint term)**.** *A constraint term (c-term) is an expression $T(A_1, \ldots, A_k)$, where $T \in \mathcal{T}_k \cup \mathcal{V}_{\mathcal{T}_k}$ and $A_i \in \mathcal{A} \cup \mathcal{V}_{\mathcal{A}}$ for $i \in \{1, \ldots, k\}$. We denote the set of c-terms by $\mathcal{C}$. In particular, if $T \in \mathcal{T}$, and $A_i \in \mathcal{A}$ for all i, we say the c-term is* ground*, otherwise we say it is* non-ground*.*

Example 3. *Consider the c-term $T(B, C)$, where $\mathcal{D}(T) = \{t_1, t_2\}$, $\mathcal{D}(B) = \{a, b\}$, $\mathcal{D}(C) = \{c\}$. The set of all possible ground c-terms we can obtain, according to the variables' domains, is $\{t_1(a, c), t_1(b, c), t_2(a, c), t_2(b, c)\}$.* $\lhd$

Constraint terms can be composed by using standard Boolean connectives ($\wedge$, $\vee$ and $\neg$) to obtain *constraint formulae* as follows:

Definition 5 (Constraint formula)**.** *We define a* constraint formula *(or simply a* formula*) inductively as:*

- *A c-term c is a formula;*
- *Given two formulae ϕ, ψ, their conjunction $(\phi \wedge \psi)$ and their disjunction $(\phi \vee \psi)$ are formulae;*
- *Given a formula ϕ, its negation $\neg\phi$ is a formula*

We denote the set of variables contained in a formula ϕ as vars(ϕ). In particular, if vars(ϕ) $= \varnothing$, we say ϕ is a ground formula, otherwise we say it is a non-ground formula. We denote the set of constraint formulae by $\mathcal{F}$.

Note that shorthands can be used to denote logical implication ($\rightarrow$), equivalence ($\leftrightarrow$), and exclusive-or ($\oplus$), as usually done in propositional logic.

Example 4. *An example of constraint formula is $\phi =$ chain-response(a,b) $\rightarrow f(a, b, c)$, based on the two atomic constraints introduced in Example 1 and Example 2. Intuitively, such formula states that whenever chain-response(a,b) holds in a trace π, then activities a, b and c must occur the same number of times in π.* $\lhd$

Given a non-ground c-term, it is possible to obtain a ground c-term by substituting each variable X with one value in $\mathcal{D}(X)$. This lead to the notion of *interpretation*, as defined below:

Definition 6 (Interpretation)**.** *An* interpretation over the domain $\mathcal{D}$ *is a function $\iota : \mathcal{V} \cup \mathcal{A} \cup \mathcal{T} \cup \mathcal{C} \cup \mathcal{F} \mapsto (\mathcal{A}^* \mapsto \{\top, \bot\}) \cup \mathcal{A} \cup \mathcal{T} \cup \mathcal{F}$ inductively defined as follows:*

- *If $a \in \mathcal{A} \cup \mathcal{T}$ then $\iota(a) = a$;*
- *If $X \in \mathcal{V}_{\mathcal{A}} \cup \mathcal{V}_{\mathcal{T}}$ then $\iota(X) \in \mathcal{D}(X)$*
- *If $T(A_1, \ldots, A_k)$ is a c-term then $\iota(T(A_1, \ldots, A_k)) = \iota(T)(\iota(A_1), \ldots, \iota(A_k))$*
- *If ϕ is a constraint formula, then its interpretation is defined inductively over the subformulas of ϕ:*
  - *If $\phi = \phi_1 \wedge \phi_2$, then $\iota(\phi) = \iota(\phi_1) \wedge \iota(\phi_2)$*
  - *If $\phi = \phi_1 \vee \phi_2$, then $\iota(\phi) = \iota(\phi_1) \vee \iota(\phi_2)$*
  - *If $\phi = \neg\psi$, then $\iota(\phi) = \neg\iota(\psi)$*

Example 5. *Consider an interpretation ι such that $\iota(X) = a, \iota(Y) = b$, and $t \in \mathcal{T}$. Then for a constraint term $t(X, Y)$, we have that $\iota(t(X, Y)) = \iota(t)(\iota(X), \iota(Y)) = t(a, b)$.* $\lhd$

Note that, the interpretation of a formula ϕ is itself a constraint – although, in general, not an atomic constraint. In fact, an interpretation maps c-terms to (atomic) constraints and such (atomic) constraints are aggregated by means of some Boolean connectives when subformulas of ϕ are inductively interpreted. Thus, in the following we overload the operator $\models$ and write $\pi \models \iota(\phi)$ whenever $\iota(\phi)(\pi) = \top$.

It should be clear by now that constraint formulae allow to mix heterogeneous formalisms to model process behavior. In our setting, process models consist of a constraint formula ϕ (or, equivalently, a conjunction of multiple constraint formulae $\phi_1, \ldots, \phi_n$ - the conjunction of formulae is still a formula). Furthermore, if Φ is a finite set of constraint formulae, whenever $\pi \models \bigwedge_{\phi \in \Phi} \phi$, we will write $\pi \models \Phi$.

4 Process Mining via Constraints Formulae

In this section we introduce some relevant problems in process mining, namely, *conformance checking*, *query checking*, *variant discrimination* and *trace clustering*, by recasting them in the constraint formulae framework.

Conformance Checking. *Conformance checking* is the problem of comparing an execution trace to a process model in order to identify any deviations from the expected behavior (i.e., the problem of deciding if a trace is *conformant* or not to a model). In our setting, process models are represented as constraint formulae and, thus, conformance checking translates to checking if a trace π is accepted by a (set of) constraint formula(e):

Problem 1 (CK - Conformance Checking)**.** *Given a trace π and a ground constraint formula ϕ,* CK*(π, ϕ) is the problem of deciding if $\pi \models \phi$.*

Clearly, the above problem can be easily extended to event logs (that is, a multiset of traces) instead of a single execution trace by checking if $\pi \models \phi$ holds for each π in the log.

Query Checking. Given an event log $\mathcal{L}$ and a non-ground constraint formula ϕ, an interesting problem in process mining is to compute an interpretation ι such that $\pi \models \iota(\phi)$ for each $\pi \in \mathcal{L}$. However, event logs obtained by observing real business processes are usually affected by data quality problems such as the presence of anomalous traces and noise (e.g., erroneous data recordings) (Koschmider et al. 2022). Clearly, both anomalies and noise can have a negative impact on the query checking problem (e.g., an interpretation that allows to model all the behaviors reported in the log might not exist) and in practical applications may be convenient to reason about a *relaxed* version of the problem that leverages the notion of *support*:

Definition 7 (Support)**.** *Let ϕ be a ground formula and $\mathcal{L}$ an event log. The* support *of ϕ in the log $\mathcal{L}$ is the fraction of traces that model ϕ wrt the size of the log, defined as follows:*

$$S(\phi, \mathcal{L}) = \frac{1}{|\mathcal{L}|} \cdot \sum_{\pi \in \mathcal{L}, \pi \models \phi} \#_{\mathcal{L}}(\pi)$$

where $\#_{\mathcal{L}}(\pi)$ is the number of occurrences of π in $\mathcal{L}$.

Problem 2 (QC - Query Checking)**.** *Given an event log $\mathcal{L}$, a non-ground constraint formula ϕ and a support threshold $s \in [0, 1]$,* QC*$(\mathcal{L}, \phi, s)$ is the problem of computing an interpretation ι such that $S(\iota(\phi), \mathcal{L}) \geq s$.*

Discriminative process discovery. In real business processes, it is quite common to have different classes of process behaviors that correspond to different variations of the process that can occur in practice. In this setting, it is of practical interest to determine *what distinguishes* and *characterizes* the different variations, that usually require *divergent* process models to be explained. Such a problem is referred to as *discriminative process discovery* and it has been recently studied for DECLARE models (Chesani et al. 2022a), by using ASPRIN (Brewka et al. 2015) in order to compute a DECLARE model which accepts all traces in a log and reject all traces in another log that is optimal wrt some preference criteria. Here, we formally characterize discriminative process discovery, not limited to DECLARE formulas.

As in the case of query checking, data quality issues can have a negative impact on the possibility of identifying a constraint formula that perfectly characterizes two logs (in some cases, such a formula might not exist). For this reason, hereafter, we will rely on some *support threshold*.

Definition 8 (s-Separates)**.** *Let $\langle \mathcal{L}, \mathcal{L}' \rangle$ be a pair of logs and $s \in [0,1]$ a support threshold. A formula ϕ* separates *$\langle \mathcal{L}, \mathcal{L}' \rangle$ with threshold s (i.e., s-separates) if $S(\phi, \mathcal{L}) \geq s$ and $S(\phi, \mathcal{L}') = 0$.*

Separation is one of the key problems underlying discriminative process discovery. Indeed, discriminative process discovery differs from standard process discovery since it aims to *describe* a process variant with respect to the other variants of the same process. If $\mathcal{L}$ contains the traces registered for a process variant and $\mathcal{L}'$ contains the traces registered for all the other variants of the same process, a constraint formula that *separates* $\mathcal{L}$ and $\mathcal{L}'$ can *explain the difference* of one variant with respect to the others.

Problem 3 (k-DISD - k-Discriminative discovery)**.** *Let $\langle \mathcal{L}_1, \ldots, \mathcal{L}_k \rangle$ be a partition of an event log $\mathcal{L}$, Φ be a finite set of ground constraint formulae, $l \in \mathbb{N}$ be an integer and $s \in [0,1]$ a support threshold. k-DISD$(\langle \mathcal{L}_1, \ldots, \mathcal{L}_k \rangle, \Phi, l, s)$ is the problem of deciding whether k subsets $\Phi_1, \ldots, \Phi_k$ of Φ exist such that for each $i \in \{1, ...k\}$ it holds that $|\Phi_i| \leq l$ and $\bigwedge_{\phi \in \Phi_i} \phi$ s-separates $\langle \mathcal{L}_i, \mathcal{L} \setminus \mathcal{L}_i \rangle$*

Strictly related to the k-discriminative discovery is the *k-trace clustering* problem, that is the problem of finding a k sets of constraint formulae able to identify the sublogs corresponding to k process variants.

Problem 4 (k-TC - k-Trace clustering)**.** *Let $\mathcal{L}$ be an event log, Φ a finite set of ground constraint formulae, and $k, l \in \mathbb{N}$. k-TC$(\mathcal{L}, \Phi, k, l)$ is the problem of deciding weather r subsets $\Phi_1, \ldots, \Phi_r$ of Φ exist, with $r \leq k$, such that $1 \leq |\Phi_i| \leq l$ for $i \in \{1, ..., r\}$, and the sets $\mathcal{L}_i = \{\pi \in \mathcal{L} : \pi \models \bigwedge_{\phi \in \Phi_i} \phi\}$ for $i \in \{1, ..., r\}$ are a partition of $\mathcal{L}$, where each $\mathcal{L}_i$ contains at least one trace.*

In the following, we indicate by $\mathcal{L}[\Phi_i]$ the subset of traces that model Φ_i, that is $\mathcal{L}[\Phi_i] = \{\pi \in \mathcal{L} : \pi \models \bigwedge_{\phi \in \Phi_i} \phi\}$.

5 Complexity Results

In this section, we study the computational complexity of the process mining tasks introduced in Section 4.

Conformance Checking. In the general setting, checking if a constraint holds over a trace is Turing-complete since for its evaluation could be necessary to run an arbitrary procedure encoded in whatever programming language today available. However, the most used languages for declarative process specification (e.g., DECLARE or LTL_f) are based on constraints that can be evaluated in polynomial time with respect to trace length. If we restrict our attention to such class of constraints then the conformance checking problem becomes tractable.

Theorem 1. *Given a trace π and a ground constraint formula ϕ, CK(π, ϕ) can be solved in $\mathcal{O}(|\phi| \cdot t \cdot \gamma)$, where $|\phi|$ is the total number of symbols (i.e., Boolean connectives, variables, activities, templates) that appear in ϕ, t is the number of c-terms in the formula and γ is the complexity of evaluating an atomic constraint over π.*

Proof. A conformance checking algorithm works by traversing bottom-up the parse tree of ϕ (whose number of nodes is linear in $|\phi|$) and associating a Boolean value to each node. In particular, given a trace π, a node n is associated the value $\top$ if π satisfies the constraint subformula rooted at n, and the value $\bot$ otherwise. The cost of computing the value of each leaf is $\mathcal{O}(\gamma)$, and the cost of computing the value of all the leaves of the tree is $\mathcal{O}(t \cdot \gamma)$ (if ϕ contains t c-terms) – in this respect if the c-terms of the formula are different in nature, γ is the maximum cost of evaluation overall the c-terms. The value of intermediate nodes of the parse tree is computed according to the semantics of Boolean connectives and on the values of the children. □

Query Checking. In this section we show that the Query Checking problem is NP-complete. Membership directly derives from the fact that conformance checking can be checked in polynomial time (by restricting our attention to tractable constraints only). In fact, given (i.e. guessed) a ι, checking that $\iota(\phi)$ is above the support threshold corresponds precisely to conformance checking of a ground formula. Hardness result is provided via reduction from the ONE-IN-THREE POSITIVE 3SAT problem (Garey and Johnson 1979; Schaefer 1978) that, given a formula in conjunctive normal form with three positive literals per clause, is the problem of checking whether there exists a satisfying assignment so that exactly one literal in each clause is true.

Theorem 2. *Given an event log $\mathcal{L}$, a non-ground constraint formula ϕ and a support threshold $s \in [0,1]$, QC$(\mathcal{L}, \phi, s)$ is NP-hard.*

Proof. Let $\psi = c_1 \wedge ... \wedge c_m$ be a Boolean formula over the variables $V(\psi) = \{X_1, \ldots, X_n\}$, where each clause contains positive literals only. The symbol $X_j^i \in V(\psi)$ with $i \in \{1,2,3\}$, $j \in \{1, \ldots, m\}$ denotes the i-th literal of the j-th clause. Consider the Query Checking instance, graphically depicted in Figure 2a, defined over the set of activities $\mathcal{A} = \{a_\top, a_\bot\}$ and such that:

- There is only one 3-template $f(.,.,.)$, whose instantiation $f(a,b,c)$ is the constraint that checks whether a trace contains exactly one among the activities a, b and c;

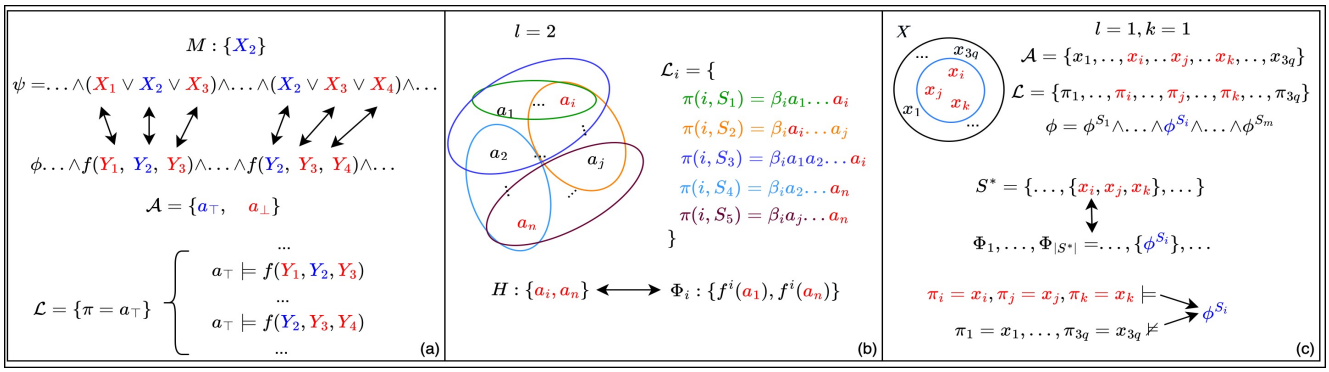


Figure 2: Reductions in the proofs of (a) Theorem 2, (b) Theorem 3 and (c) Theorem 4.

- For each variable $X_i \in V(\psi)$, there is an activity variable $Y_i \in \mathcal{V}_\mathcal{A}$ such that $\mathcal{D}(Y_i) = \mathcal{A}$;
- For each clause $c_j = (X_j^1, X_j^2, X_j^3)$ of ψ, there exists a c-term $C_j = f(Y_j^1, Y_j^2, Y_j^3)$. The symbols $Y_j^i \in \mathcal{V}_\mathcal{A}$ with $i \in \{1,2,3\}, j \in \{1,\dots,m\}$ denote the i-th activity term of the c-term C_j. Furthermore, $X_j^i = X_k \iff Y_j^i = Y_k$;
- The constraint formula is obtained as the conjunction of all c-terms, i.e., $\phi = \bigwedge_{c_j \text{ in } \psi} C_j$;
- The log $\mathcal{L}$ contains only one trace $\pi = a_\top$;
- The support threshold is $s = 1$.

We now claim that ONE-IN-THREE POSITIVE 3SAT has a solution over ψ if, and only if, there exists a solution to $\text{QC}(\mathcal{L}, \phi, 1)$.

($\rightarrow$) Let M be a satisfying assignment of ψ such that, for each clause c_j exactly one variable evaluates true. Let $\sigma_j : \{1,2,3\} \mapsto \{1,2,3\}$ be a permutation such that $M(X_j^{\sigma_j(1)}) = 1$ and $M(X_j^{\sigma_j(2)}) = M(X_j^{\sigma_j(3)}) = 0$. Consider now the interpretation ι_M that, for each clause c_j in ψ, assigns $\iota_M(Y_j^{\sigma_j(1)}) = a_\top$ and $\iota_M(Y_j^{\sigma_j(2)}) = \iota_M(Y_j^{\sigma_j(3)}) = a_\bot$. Due to the semantics of the template f, for each clause c_j in ψ, the (ground) c-term $\iota_M(f(Y_j^1, Y_j^2, Y_j^3)) = f(\iota_M(Y_j^1), \iota_M(Y_j^2), \iota_M(Y_j^3))$ is satisfied over the trace $\pi = a_\top \in \mathcal{L}$. Moreover, since the constraint formula ϕ is the conjunction of all c-terms $f(Y_j^1, Y_j^2, Y_j^3)$ for each $j \in \{1,...,m\}$, we have that $S(\iota_M(\phi), \mathcal{L}) = 1 \geq s$ holds.

($\leftarrow$) Let ι be an interpretation which solves $\text{QC}(\mathcal{L}, \phi, 1)$. Since ϕ is a conjunction of c-terms and $s = 1$, this means that the j-th c-term $C_j = f(Y_j^1, Y_j^2, Y_j^3)$ must hold over the trace π. In particular, due to the semantics of the template f, for each $f(Y_j^1, Y_j^2, Y_j^3)$ exactly one of the variables is mapped to $a_\top$ while the other two are mapped to $a_\bot$. Let $\sigma_j : \{1,2,3\} \mapsto \{1,2,3\}$ be a permutation such that $\iota(Y_j^{\sigma_j(1)}) = a_\top$, and $\iota(Y_j^{\sigma_j(2)}) = \iota(Y_j^{\sigma_j(3)}) = a_\bot$. It follows that the truth assignment $M : V(\psi) \mapsto \{0,1\}$ such that $M(X_i) = 1$ if and only if $\iota(Y_i) = a_\top$ is a satisfying assignment for ψ. In fact, if ι is a solution to $\text{QC}(\mathcal{L}, \phi, 1)$ in the j-th c-term $C_j = f(Y_j^1, Y_j^2, Y_j^3)$ we have that: *(i)* $\iota(Y_j^{\sigma_j(1)}) = 1$ and, hence, $M(X_j^{\sigma_j(1)}) = 1$; *(ii)* $\iota(Y_j^{\sigma_j(2)}) = \iota(Y_j^{\sigma_j(3)}) = a_\bot$ and, hence, $M(X_j^{\sigma_j(2)}) = M(X_j^{\sigma_j(3)}) = 0$; and, thus, for each clause $c_j = (X_j^{\sigma_j(1)}, X_j^{\sigma_j(2)}, X_j^{\sigma_j(3)})$ exactly one variable is true in M. We can conclude that M is a model for ψ since ψ is the conjunction of all clauses c_j, and each c_j is true in M. □

k-Discriminative Discovery and k-Trace Clustering. In this section we show that both k-Discriminative Discovery and k-Trace Clustering problems are NP-complete. Membership in NP for k-Discriminative Discovery follows from Theorem 1, since given a set of constraints formulae and k logs, the support of each constraint formula w.r.t. the logs can be computed in polynomial time (under the assumption that constraints can be evaluated in polynomial time). Then, in the following we will prove NP-hardness via a reduction from the Hitting Set problem (Garey and Johnson 1979), that, given a collection $\{S_1, \dots, S_k\}$ of subsets of the universe $U_n = \{a_1, ..., a_n\}$ and an integer l, is the problem of checking if there exists a subset $H \subseteq U_n$ such that $|H| \leq l$ and H intersects (hits) every set in $\{S_1, \dots, S_k\}$.

Theorem 3. *Given k event logs $\langle \mathcal{L}_1, ...\mathcal{L}_k \rangle$, a set of ground constraint formulae Φ, an integer l and a support threshold $s \in [0,1]$, k-DISD$(\langle \mathcal{L}_1, \dots, \mathcal{L}_k \rangle, \Phi, l, s)$ is NP-hard.*

Proof. Let $C = \{S_1, ..., S_k\}$ be a collection of subsets of the universe $U(n) = \{a_1, ...a_n\}$ and $l \in \mathbb{N}$. Consider the k-Discriminative Discovery instance k-DISD$(\langle \mathcal{L}_1, \dots, \mathcal{L}_k \rangle, \Phi, l, s)$ graphically depicted in Figure 2b such that:

- The activity set on which constraint formulae are defined is $\mathcal{A} = U(n) \cup \{\beta_1, ..., \beta_k\}$.
- The logs $\mathcal{L}_1, ..., \mathcal{L}_k$ are such that $\mathcal{L}_i = \{\pi(i, S_j) : 1 \leq j \leq k\}$, where $\pi(i, S_j)$ denotes the trace obtained by concatenating the activity β_i with the activities in S_j taken in whatever order (e.g, $\pi(i, S_j) = \beta_i a_1 a_2 ... a_q$ if $S_j = \{a_1, a_2, ..., a_q\}$);
- There is a 1-template $g(\cdot)$, whose instantiation $g(p)$ checks whether a trace does not begin with the activity p; that is $\pi \models g(p)$ if, and only if, $\pi[0] \neq p$;

- There is a 1-templates $h(\cdot)$, whose instantiation $h(p)$ checks whether a trace does not contain the activity p; that is $\pi \models h(p)$ if, and only if, $p \notin \pi$.
- The set of constraints formulae is $\Phi = \bigcup_{p \in U} \{f^i(p) : 1 \leq i \leq k\}$, where $f^i(p) = (g(\beta_i) \implies h(p))$. In particular, the semantics of each f^i_p is as follows:
  - $\pi(i, S_j) \models f^i(p)$ holds for each $\pi(i, S_j)$. Indeed, since $\pi(i, S_j)[0] = \beta_i$ we have that $\pi(i, S_j) \not\models g(\beta_i)$;
  - For each $r \neq i$ we have that $\pi(r, S_j) \models f^i(p)$ holds if, and only if, $p \notin \pi(r, S_j)$ meaning that $p \notin S_j$ – note that if $r \neq i$ we have that $\pi(r, S_j) \models g(\beta_i)$ holds and, thus, $\pi(r, S_j) \models h(p)$ must also hold.
  - The support threshold is $s = 1$.

We now claim that the Hitting Set has a solution over (C, U_n, k) if, and only if, there exists a solution to k-DISD($\langle \mathcal{L}_1, \ldots, \mathcal{L}_k \rangle, \Phi, l, s$).

($\rightarrow$) Let H be a solution to the hitting set instance. Consider the sets of constraint formulae $\{\Phi_1, ..., \Phi_k\}$ such that $\Phi_i = \{f^i(p) : p \in H\}$ for $i \in \{1, ..., k\}$. Let $\mathcal{L}_q$ be an arbitrary log in $\{\mathcal{L}_1, \ldots, \mathcal{L}_k\}$. Consider the set of constraint formulae Φ_q. By definition $\pi(q, S_j) \models f^q(p)$ for each $f^q(p) \in \Phi_q$ and each $\pi(q, S_j) \in \mathcal{L}_q$ and, thus, $S(\bigwedge_{\phi \in \Phi_q} \phi, \mathcal{L}_q) = 1$. Now, consider an arbitrary log $\mathcal{L}_r$ with $r \neq q$ and let $\pi(r, S_t)$ be an arbitrary trace in $\mathcal{L}_r$. Since H is a solution to the hitting set instance, it means that there exists at least one element $p \in S_t \cap H$ and by definition $\pi(r, S_t) \not\models f^r(p) \in \Phi_q$ and, thus, $S(\bigwedge_{\phi \in \Phi_q} \phi, \mathcal{L}_r) = 0$. The latter reasoning holds for each log $\mathcal{L}_r \in \{\Phi_1, ..., \Phi_k\} \setminus \mathcal{L}_q$ and, thus, $S(\bigwedge_{\phi \in \Phi_q} \phi, \bigcup_{\mathcal{L}_r \in \{\Phi_1, ..., \Phi_k\} \setminus \mathcal{L}_q} \mathcal{L}_r) = 0$ also holds. We can conclude that the sets $\Phi_1, ..., \Phi_k$ of constraint formulae are a solution to the k-DISD($\langle \mathcal{L}_1, \ldots, \mathcal{L}_k \rangle, \Phi, l, s$) instance by noticing that for each Φ_i it holds that $|\Phi_i| \leq l$.

($\leftarrow$) Let $\{\Phi_1, ..., \Phi_k\}$ be a solution to k-DISD($\langle \mathcal{L}_1, \ldots, \mathcal{L}_k \rangle, \Phi, l, s$). Consider an arbitrary set of constraint formulae Φ_i, and the set $H = \{h : f^i(h) \in \Phi_i\}$. Since $\{\Phi_1, ..., \Phi_k\}$ is a solution to the k-discriminative discovery problem, it means that $S(\bigwedge_{\phi \in \Phi_i} \phi, \mathcal{L}_i) = 1$ and $S(\bigwedge_{\phi \in \Phi_i} \phi, \mathcal{L}_j) = 0$ for $1 \leq j \leq k$ and $j \neq i$. In particular, consider an arbitrary log $\mathcal{L}_j$ with $j \neq i$. For all traces $\pi(j, S_q) \in \mathcal{L}_j$ there exists at least one constraint formula $f^i(p) \in \Phi_i$ such that $\pi(j, S_q) \not\models f^i(p)$ because $p \in \pi(j, S_q)$. This implies that $p \in H$ and, thus, H hits S_q. Recall that $\mathcal{L}_j$ contains exactly one trace $\pi(j, S_q)$ for each $S_q \in C$. Thus, we can conclude that H is a solution to the hitting set instance (C, U_n, k) by noticing that $|H| = |\Phi_i| \leq l$. □

In the following we show k-Trace Clustering to be NP-complete. If $|\Phi| = n$, then there are $\sum_{i=0}^{k} \binom{n}{i}$ candidate solutions. Membership in NP follows from Theorem 1, since checking a candidate solution $\Phi_1, \ldots, \Phi_r$ with $r \leq k$ can be performed in polynomial time (under the assumption that constraints can be evaluated in polynomial time) by performing: *(i)* r conformance checking to compute $\mathcal{L}[\Phi_j]$ – where $\mathcal{L}[\Phi_j]$ denotes the set of traces in $\mathcal{L}$ that satisfy Φ_j; *(ii)* a polynomial-time check to determine whether $\mathcal{L}[\Phi_1], \ldots, \mathcal{L}[\Phi_r]$ is a partition of $\mathcal{L}$. We next prove hardness by a reduction from the Exact Cover by 3-Sets (Garey and Johnson 1979), that, given a finite set X of elements with $|X| = 3q$ for some $q \in \mathbb{N}$ and a finite collection of sets S, with $S_i \subseteq X$ and $|S_i| = 3$ for each $S_i \in S$, is the problem of checking if there exists a subset of S which partitions X.

Theorem 4. *Given a log $\mathcal{L}$, a set of ground constraint formulae Φ and integers $k, l \in \mathbb{N}$, k-TC($\mathcal{L}, \Phi, k, l$) is NP-hard.*

Proof. Consider the Exact Cover by 3-Sets problem instance (X, S), on the finite set $X = \{x_1, \ldots, x_{3q}\}$ and the collection $S = \{S_1, \ldots, S_m\}$ with $S_i \subseteq X$ and $|S_i| = 3$ for $i \in \{1, ..., m\}$. Starting from (X, S) we build the following k-Trace clustering instance (depicted in Figure 2c):

- The set of activities on which traces and constraint formulae are defined is $\mathcal{A} = X$.
- The log is $\mathcal{L} = \{\pi_x = x : x \in X\}$, that is, $\mathcal{L}$ contains a trace $\pi_x = x$ for each $x \in X$.
- There is one 3-templates $f(.,.,.)$ whose instantiation $f(x, y, z)$ is the constraint whose valuation is defined s.t. $\pi_p \models f(x, y, z)$ if, and only if, $p = x$ or $p = y$ or $p = z$.
- The set of constraints is $\Phi = \bigcup_{i=1}^{m} \{f(x_1, x_2, x_3) : S_i = \{x_1, x_2, x_3\}\}$ – note that wlog we can assume whatever order among the elements of S_i since the valuation of $f(x_1, x_2, x_3)$ only checks for the presence of an element without considering their relative order. In the following we will indicate by ϕ^{S_i} the constraint $f(x_1, x_2, x_3)$ such that $\{x_1, x_2, x_3\} = S_i \in S$.
- The bound on the size of subsets of formulae is $l = 1$.
- The integer $k = |X|/3$.

We now claim that the Exact Cover by 3-Sets has a solution over (X, S) if, and only if, there exists a solution to k-TC($\mathcal{L}, \Phi, l, k$).
($\rightarrow$) Let $S^* \subseteq S$ be a solution to the Exact Cover problem over (X, S), that is, S^* is a partition of X. Let us consider an arbitrary order of the subsets in S^* and indicate by S^*_i the i-th set in S^*. Consider the set of constraints $\Phi_1, ..., \Phi_{|S^*|}$ such that each $\Phi_i = \{\phi^{S^*_i}\}$ for each $i \in \{1, ..., |S^*|\}$ (note that $|\Phi_i| = 1$ for all $i \in \{1, ..., |S^*|\}$ and $|S^*|$ is $|X|/3$). Let π_x be an arbitrary trace in $\mathcal{L}$. Since S^* solves Exact Cover by 3-Sets, then there exists $S^*_i \in S^*$ such that $x \in S^*_i$. Thus, we also have that $\pi_x \models \phi^{S^*_i}$ and $\pi_x \in \mathcal{L}[\Phi_i]$ hold. We can conclude that $\Phi_1, ..., \Phi_{|S^*|}$ cover the entire log $\mathcal{L}$. Then, let Φ_i, Φ_j be two sets of constraint formulae in $\{\Phi_1, ..., \Phi_{|S^*|}\}$. Suppose that there exists a trace $\pi_x \in \Phi_i \cap \Phi_j$, this implies that $\pi_x \models \phi^{S^*_i} \wedge \phi^{S^*_j}$ - due to the semantics of $\phi^{S^*_i}$ and $\phi^{S^*_j}$. This means that $x \in S_i \cap S_j$. However, since S^* is a partition of X we know that $S_i \cap S_j = \emptyset$ for each i, j and we can conclude that $\mathcal{L}[\Phi_i] \cap \mathcal{L}[\Phi_j] = \emptyset$ for each i, j. This prove that $\mathcal{L}[\Phi_1], ..., \mathcal{L}[\Phi_{|S^*|}]$ is a partition of $\mathcal{L}$ and, thus, the set $\Phi_1, ..., \Phi_{|S^*|}$ is a solution to k-TC($\mathcal{L}, \Phi, l, k$).
($\leftarrow$) Let $\Phi_1, \ldots, \Phi_r$, with $r \leq k$ and $|\Phi_i| = 1$ for $i \in \{1, ..., r\}$ be a solution to the k-Trace Clustering problem. It means that $\mathcal{L}[\Phi_1], \ldots, \mathcal{L}[\Phi_r]$ induce a partition on $\mathcal{L}$. Consider now the set $S^* = \bigcup_{i=1}^{r} \{S_j : \Phi_i = \{\phi^{S_j}\}\}$. For each element $x \in X$, and the corresponding trace $\pi_x \in \mathcal{L}$, we have that there exists exactly one set of constraint formulae

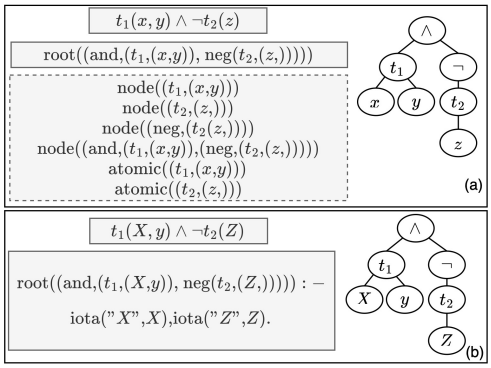


Figure 3: Encoding of (a) ground formula; (b) non-ground formula.

$\overline{\Phi} = \{\phi^{S_j}\} \in \{\Phi_1, \ldots, \Phi_r\}$ such that $\pi_x \in \mathcal{L}[\overline{\Phi}]$. Due to the semantics of ϕ^{S_j} it means that $\pi_x \models \phi^{S_j}$ and, thus, $x \in S_j$. By construction, S_j is included in S^*. Thus, we can conclude that the set $S^* = \bigcup_{i=1}^{r} \{S_j : \Phi_i = \{\phi_{S_j}\}\}$ is a solution to Exact Cover by 3-Sets over (X, S). □

6 ASP Encodings

Answer Set Programming (ASP) (Brewka, Eiter, and Truszczynski 2011; Gelfond and Lifschitz 1991; Niemelä 1999) is a logic programming paradigm, that allows to model in a declarative way problems up to the Σ_2^P complexity class (Dantsin et al. 2001). In this paper, we assume the reader is familiar with the syntax of and semantics of ASP (a nice introductions to ASP is given by (Gebser et al. 2012; van Harmelen, Lifschitz, and Porter 2008; Baral 2010)). The solutions presented in the following are encoded according to the in input language of GRINGO (Gebser et al. 2015; Gebser, Schaub, and Thiele 2007), and resort to externally-evaluated atoms available in CLINGO (Gebser et al. 2016). We refer the reader to (Gebser et al. 2012) for more details.

Encoding of constraint formulae in ASP. We model constraint formulae as anonymous nested function symbols which encode the formula's parse tree, where `neg`, `and`, `or` constants represent logical operators. The predicate `root/1` (as usual in ASP we denote by p/k a predicate p of arity k) encodes the input formula, and the following ASP program computes its subformulae (predicate `node/1`), and identifies atomic constraints (predicate `atomic/1`).

```
node(C):- root(C).
node(LHS; RHS):- node((and,LHS,RHS)).
node(LHS; RHS):- node((or,LHS,RHS)).
node(C):- node((neg,C)).
atomic((T,A)):- node((T,A)), T != neg.
```

Example 6. *Figure 3a reports the encoding of the formula $t_1(x, y) \wedge \neg t_2(z)$ and the output of the above program.* ◁

Conformance checking. As outlined in Theorem 1, we can perform conformance checking of a constraint formula by traversing its tree structure in a bottom-up fashion and aggregating previously computed evaluations according to each subformula semantics. This is obtained as follows:

```
holds((and,LHS,RHS),TID):- holds(LHS,TID),
    holds(RHS,TID), node((and,LHS,RHS)).
holds((or,LHS,RHS),TID):- node((or,LHS,RHS)),
    holds(LHS,TID).
holds((or,LHS,RHS),TID):- node((or,LHS,RHS)),
    holds(RHS,TID).
holds((neg,C),TID):- node((neg,C)),not
holds(C,TID),
    trace(TID,_).
holds(C,@atomic_check(C)):- atomic(C).
```

Where `trace/2` models a control-flow process variant, where the first term is a unique identifier and the second term is the number of occurrences in the event log, and the `holds/2` predicate models that a formula accepts a given control-flow variant uniquely identified by `trace/2`. Atomic constraints are evaluated (the last rule of the program above) by using external backends, such as `Declare4Py` (Donadello et al. 2022) or Python's `regex` regular expression matching library, to be injected into the main program. This is done exploiting CLINGO's `@`-terms, where `atomic_check` identifies a Python procedure, which evaluates the atomic constraint `C` over all the event log. In alternative, atomic constraints can be evaluated in a pre-processing step and added as facts to the program.

If answer set of the program above contains the atom `holds(root(...),TID)` then the control-flow variant identified by `TID` is conformant to the formula in input.

Query checking. In query checking, the input formula contains variables, and the domain of variables is modeled by predicate `domain/2`. Variables of the constraint formula are identified in ASP using string terms of the form `"X"` for each variable X, i.e., atoms `domain("X", x)` for $x \in \mathcal{D}(X)$ – at the encoding level, there is no difference between activity variables and domain variables.

The following choice rule guesses one assignment (modeled by `iota/2`) for each variable in the constraint formula:

```
1 {iota(X,Y): domain(X,Y)} 1 :- domain(X,_).
```

Next, a rule of the following form is added to select the instantiation of the input formula corresponding to the assignment in `iota/2`:

```
root(F) :- iota("X1",X1),...,iota("Xn",Xn).
```

where F is the parse tree of the input formula, as in the conformance checking encoding, and Xi, with $i \in \{1, \ldots, n\}$, is the i-th variable of the n that occurs in the constraint formula in input.

Example 7. *Figure 3b reports the rule encoding the formula $t_1(X, y) \wedge \neg t_2(Z)$, containing variables `X` and `Z`.* ◁

Finally, the following constraint discards assignments with support below the threshold:

```
support(C,S) :- atomic(C),
   S=#sum{W,TID: trace(TID,W), holds(C,TID)}.
support(C,S) :- node(C),
   S=#sum{W,TID: trace(TID,W), holds(C,TID)}.
:- support(C,S), root(C), S<supp_thr.
```

Here, `supp_thr` is a CLINGO constant modeling the desired support threshold.

An ASP program performing query checking can be obtained by adding to the above-specified rules the program reported in the previous paragraph that performs conformance checking of a formula specified by the `root/1` fact.

Trace Clustering. The input is modelled by predicates `formula/1`, `accepts/2`, and `rejects/2`. The first encodes the input formulae, whereas the latter two store whether a given formula accepts or rejects a control-flow variant. The number of partitions k and the maximum number of formulae per set of constraints p that will induce a partition are provided as CLINGO constants. The following ASP program encodes trace clustering:

```
partition(1..k).
1 { assign(C,P): formula(C) } p
      :- partition(P).
subset_rejects(P,TID) :- assign(C,P),
      rejects(C,TID).
subset_accepts(P,TID) :- partition(P),
      trace(TID), not subset_rejects(P,TID).
:- trace(TID),
      #count{P: subset_accepts(P,TID)} != 1.
:- partition(P),
      #count{TID: subset_accepts(P,TID)} = 0.
```

The first rule guesses whether at least one and at most p formula can be assigned to a partition. The next two rules compute predicates `subset_accepts(TID,P)`, and `subset_rejects(TID,P)` modeling that the conjunction of constraints in partition `P` accepts or (resp. rejects) the control-flow variant with identifier `TID`. The constraints select non-trivial partitions. Since each trace is accepted by precisely one partition, and each trace is either accepted or rejected, formulae subsets induce disjoint sublogs.

Discriminative discovery. The task is similar to Trace Clustering, but here the log is already partitioned. The input predicate `log/2` is used to model which control-flow variants belong to which log partition. As before, `partition/1` predicate models available partitions, and `formula/1` the formulae in input. Input constants `p` and `sup_thr` are defined as before. `threshold/2` is an input predicate that models the number of traces that should be satisfied to be above the support threshold (this is to avoid using floats for the threshold).

The following program models discriminative discovery:

```
partition(L) :- label(_,L).
above_support(C,L) :- partition(L), formula(C),
  #sum{F,TID: accepts(C,TID), trace(TID,F),
     label(TID,L)} > S,
  threshold(L,S).
rejects_something(C,L) :- partition(L),
    formula(C), rejects(C,TID), log(TID,L).
1 { assign(C,L): above_supp(C,L),
    rejects_something(C,L'), L != L' }
    p :- partition(L).
log_reject(L,TID) :- assign(C,L), rejects(C,TID).
:- log(L'), log(TID,L), not log_reject(L',TID),
   L != L'.
```

The first rule computes predicate `above_support/2` that models constraints above the support threshold on a given log partition. The subsequent rule defines predicate `rejects_something/2` that models constraint rejecting at least one control-flow variant on a given log partition. The choice rule guesses an assignment of a subset of formulae to a log partition. To prune the search space, an assignment is guessed if a formula (i) has enough support on the given log partition, and (ii) rejects at least one control-flow variant in a different partition. The `log_reject/2` models that the (conjunction of the) constraint formulae assigned to a specific log partition reject a specific control-flow variant. Finally, the constraint discards candidate answer sets where a control-flow variant is rejected by multiple log partitions.

7 Experimental Evaluation

This section discusses the results of a first experimental analysis conceived to evaluate the feasibility of our ASP-based implementation.

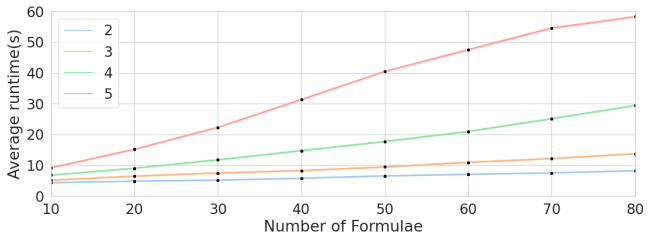
Benchmark instances. We run our experiments on a well-known event log in process mining literature, the *Sepsis Cases Event Log* (Mannhardt and Blinde 2017). In the experimental study, we evaluated our ASP-based prototype's performances on randomly generated sets of formulae. In particular, we considered the following two types of instances. The first one includes sets of n constraint formulae of depth d that contain 2^d distinct atomic constraints, denoted by $CF(n, d)$. The second type, denoted by $QC(b, D)$, includes formulae that: *(i)* are the conjunction of b constraint terms; *(ii)* contain b variables with a domain of cardinality D; and *(iii)* each formula has a unique solution to the query checking problem over the log with support $s = 1$.

The atomic constraints and terms that appear in $CF(n, d)$ and $QC(b, D)$ were mined by using the `Declare4Py` tool with a support threshold of 0.3 over the log.

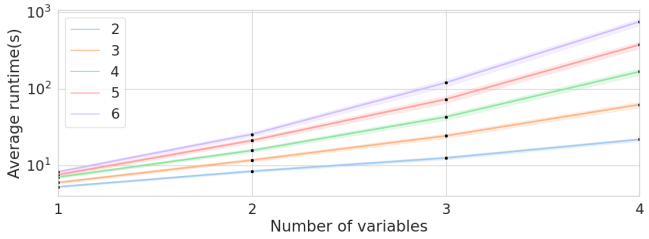
For conformance checking, trace clustering, and discriminative discovery, we sampled 30 sets of formulae from $CF(n, d)$ for each $n \in \{10, 20, \ldots, 80\}$ and $d \in \{2, 3, 4, 5\}$, for a total number of 960 sets of formulae. Furthermore, for discriminative discovery and trace clustering, we set $l = 15$ as the maximum cardinality for formulae's subsets and randomly sampled the log's labels. For query checking, we sampled 30 formulae from $QC(b, D)$ for each $b \in \{1, 2, 3, 4\}$, and $D \in \{2, 3, 4, 5, 6\}$, yielding a total of 600 formulae, and set the minimum support threshold to 1.

Execution environment. All experiments were executed on an Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz, 512GB RAM machine, using CLINGO version 5.4.0, Python 3.10, and `Declare4Py` 1.0. All experiments were run in parallel using GNU Parallel (Tange 2011). We measured average execution times over samples of the same size and reported them in our plots, where bands represent 95% confidence intervals. As described in Sec. 6, atomic constraints are evaluated by running Python procedures exploiting CLINGO's `@`-terms. Conformance checking of atomic formulae is done during pre-processing to build the input (see Sec. 6) for query checking, discriminative discovery, and trace clustering. Scripts and data for reproducibility are available on GitHub (https://github.com/kr2023-6949/experiments).

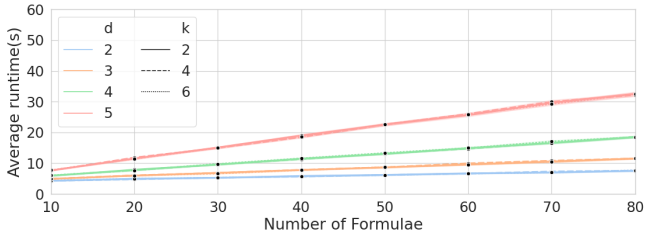
Conformance checking. Figure 4a reports the average running times (in seconds) for conformance checking. The run-



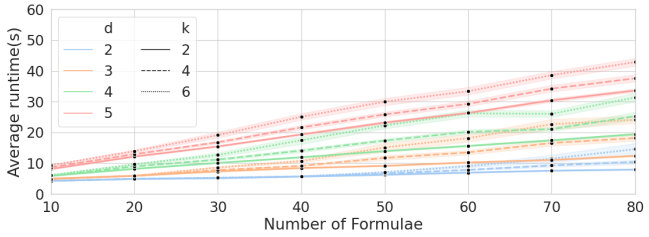
(a) Conformance checking of formulas of depth d.



(b) Query checking. D is the variable domains' cardinality.



(c) Discriminative discovery on k partitions of formulas of depth d.



(d) Trace clustering on k partitions of formulas of depth d.

Figure 4: Experimental results: Average running times in seconds.

time grows smoothly while increasing the number of constraint formulae in input and their depth. As it can be noted, in the hardest setting (i.e., 80 constraint formulae of depth 5, thus each one is made of 32 atomic constraints) running time approaches 60 seconds, that we consider acceptable.

Query checking. Figure 4b reports the average running times for query checking (log. scale y-axis). Instances with domain size $D = 2$ are easily solved (less than 60 seconds on average). As expected, finding a solution becomes more time-consuming as D increases. This is mainly due to the exponential number of formula tree instantiations (see the second rule of the query checking encoding in Sec. 6). Consequently, the ground ASP program size grows exponentially with D and the solver slows down.

Discriminative discovery. Figure 4c reports average running times for discriminative discovery with $k \in \{2, 4, 6\}$ partitions. The average running time grows smoothly with the number of constraint formulae, and hardness is mostly influenced by the depth of the formula. Performance seems unaffected by k, this is probably due to the fact that both constraint formulae and trace labels were randomly sampled, and the discriminative discovery tasks are likely over-constrained independently from k. All in all, the hardest setting (i.e., on $CF(80, 5)$ with k=6) runs in about 30 secs.

Trace clustering. Figure 4d reports average running times for trace clustering using $k \in \{2, 4, 6\}$ partitions. In this case, in contrast to discriminative discovery, the average running time is affected by the formulae's depth and the number of partitions. The hardest setting (i.e., 80 formulae with depth $d = 5$ and $k = 6$ partitions) is still feasible and requires less than 60 seconds to be solved.

Tools for basic tasks. There are no tools able to deal with the general problems introduced in this paper, however, if we restrict our attention to DECLARE (that is the de facto standard in declarative process mining) we can do some final considerations for Conformance Checking and Query Checking. As for Conformance Checking, note that a DECLARE model is a conjunction of Declare constraints and can be trivially represented by a constraint formula. Then, the only extra computation performed by our tool w.r.t. existing tools (e.g. `Declare4Py` or `RuM`) is the evaluation of a very small stratified program and no significant overhead has been observed. As for Query Checking, the only tools that solve a similar problem are `Declare4Py` and (Chiariello, Maggi, and Patrizi 2022) that perform query checking of a single DECLARE constraint by considering each of its possible instantiations and, thus, it is a much simpler problem than the one introduced in this paper. Measuring performance reduces again to conformance checking.

8 Conclusion and Future Work

Many languages have been proposed to encode process models, each featuring some distinctive feature making it more suitable in specific circumstances. Up to now, there were no means to combine them in a declarative way. This paper defines a logic-based framework for the declarative composition of heterogeneous process models. The computational complexity of some relevant process mining reasoning tasks (recast in the proposed framework) has been studied. An implementation of the framework in ASP has been provided, and a proof-of-concept evaluation on a well-known log showed the approach's feasibility.

As future work, an interesting avenue of further research is extending the proposed framework to deal with other process mining perspectives (e.g., data perspective).

Acknowledgments

This work is partially supported by the Italian Ministry of Research under PRIN project "exPlaInable kNowledge-aware PrOcess INTelligence" (PINPOINT), CUP H23C22000280006.

References

Baral, C. 2010. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Brewka, G.; Delgrande, J. P.; Romero, J.; and Schaub, T. 2015. asprin: Customizing answer set preferences without a headache. In *AAAI*, 1467–1474. AAAI Press.

Brewka, G.; Eiter, T.; and Truszczynski, M. 2011. Answer set programming at a glance. *Commun. ACM* 54(12):92–103.

Chesani, F.; Francescomarino, C. D.; Ghidini, C.; Grundler, G.; Loreti, D.; Maggi, F. M.; Mello, P.; Montali, M.; and Tessaris, S. 2022a. Optimising business process discovery using answer set programming. In *Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 498–504.

Chesani, F.; Francescomarino, C. D.; Ghidini, C.; Loreti, D.; Maggi, F. M.; Mello, P.; Montali, M.; Palmieri, E.; and Tessaris, S. 2022b. Discovering business processes models expressed as DNF or CNF formulae of declare constraints. In *Italian Conference on Computational Logic (CILC)*, volume 3204, 201–216.

Chiariello, F.; Maggi, F. M.; and Patrizi, F. 2022. Asp-based declarative process mining. In *Conference on Artificial Intelligence (AAAI)*, 5539–5547.

Dantsin, E.; Eiter, T.; Gottlob, G.; and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Comput. Surv.* 33(3):374–425.

Donadello, I.; Riva, F.; Maggi, F. M.; and Shikhizada, A. 2022. Declare4py: A python library for declarative process mining. In *BPM (PhD/Demos)*, volume 3216 of *CEUR Workshop Proceedings*, 117–121. CEUR-WS.org.

ER, M.; Arsad, N.; Astuti, H. M.; Kusumawardani, R. P.; and Utami, R. A. 2018. Analysis of production planning in a global manufacturing company with process mining. *J. Enterp. Inf. Manag.* 31(2):317–337.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Gebser, M.; Harrison, A.; Kaminski, R.; Lifschitz, V.; and Schaub, T. 2015. Abstract gringo. *Theory Pract. Log. Program.* 15(4-5):449–463.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Ostrowski, M.; Schaub, T.; and Wanko, P. 2016. Theory solving made easy with clingo 5. In *ICLP (Technical Communications)*, volume 52 of *OASIcs*, 2:1–2:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Gebser, M.; Schaub, T.; and Thiele, S. 2007. Gringo : A new grounder for answer set programming. In *LPNMR*, volume 4483 of *Lecture Notes in Computer Science*, 266–271. Springer.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Gener. Comput.* 9(3/4):365–386.

Jans, M.; van der Werf, J. M. E. M.; Lybaert, N.; and Vanhoof, K. 2011. A business process mining application for internal transaction fraud mitigation. *Expert Syst. Appl.* 38(10):13351–13359.

Koschmider, A.; Kaczmarek, K.; Krause, M.; and van Zelst, S. J. 2022. Demystifying noise and outliers in event logs: Review and future directions. In *Business Process Management Workshops*.

Mannhardt, F., and Blinde, D. 2017. Analyzing the trajectories of patients with sepsis using process mining. In *RADAR+EMISA@CAiSE*, volume 1859 of *CEUR Workshop Proceedings*, 72–80. CEUR-WS.org.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.* 25(3-4):241–273.

Partington, A.; Wynn, M. T.; Suriadi, S.; Ouyang, C.; and Karnon, J. 2015. Process mining for clinical processes: A comparative analysis of four australian hospitals. *ACM Trans. Manag. Inf. Syst.* 5(4):19:1–19:18.

Pesic, M.; Schonenberg, H.; and van der Aalst, W. M. P. 2007. DECLARE: full support for loosely-structured processes. In *IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, 287–300.

Schaefer, T. J. 1978. The complexity of satisfiability problems. In *STOC*, 216–226. ACM.

Sedrakyan, G.; Weerdt, J. D.; and Snoeck, M. 2016. Process-mining enabled feedback: "tell me what I did wrong" vs. "tell me how to do it right". *Comput. Hum. Behav.* 57:352–376.

Tange, O. 2011. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine* 36(1):42–47.

van der Aalst, W. M. P., and Carmona, J., eds. 2022. *Process Mining Handbook*, volume 448 of *Lecture Notes in Business Information Processing*. Springer.

van der Aalst, W. M. P., and et al. 2011. Process mining manifesto. In Daniel, F.; Barkaoui, K.; and Dustdar, S., eds., *Proceedings of the Business Process Management Workshops (BPM)*, 169–194.

van Harmelen, F.; Lifschitz, V.; and Porter, B. W., eds. 2008. *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*. Elsevier.