ASP-based Declarative Process Mining

Francesco Chiariello[1], **Fabrizio Maria Maggi**[2], **Fabio Patrizi**[1]

[1] DIAG - Sapienza University of Rome, Italy
[2] KRDB - Free University of Bozen-Bolzano, Italy
chiariello@diag.uniroma1.it, maggi@inf.unibz.it, patrizi@diag.uniroma1.it

Abstract

We put forward Answer Set Programming (ASP) as a solution approach for three classical problems in Declarative Process Mining: Log Generation, Query Checking, and Conformance Checking. These problems correspond to different ways of analyzing business processes under execution, starting from sequences of recorded events, a.k.a. *event logs*. We tackle them in their *data-aware* variant, i.e., by considering events that carry a payload (set of attribute-value pairs), in addition to the performed activity, specifying processes declaratively with a first-order extension of linear-time temporal logic over finite traces (LTL_f). The data-aware setting is significantly more challenging than the *control-flow* one: Query Checking is still open, while the existing approaches for the other two problems do not scale well. The contributions of the work include an ASP encoding schema for the three problems, their solution, including the first one for data-aware Query Checking, and experiments showing feasibility of the approach.

Introduction

Process Mining (PM) for Business Process Management (BPM) is a research area aimed at discovering common patters in Business Processes (BP) (van der Aalst 2016). The analysis starts from *event logs*, i.e., sets of traces that record the events associated with process instance executions, and typically assumes a *process model*, which may be taken as input, manipulated as an intermediate structure, or produced in output. Events describe process activities at different levels of details. In the simplest perspective, here referred to as *control-flow-only*, events model atomic *activity* performed by a process instance at some time point; in the most complex scenario, typically referred to as *multi-perspective*, events also carry a payload including a timestamp and activity data.

Processes can be specified *prescriptively*, i.e., as models, such as Petri Nets, that generate traces, or *declaratively*, i.e., through logical formulas representing the constraints that traces must satisfy in order to comply with the process. This is the approach we adopt here. Specifically, we take a (untimed) *data-aware* perspective where events include the activity and set of attribute-value pairs, the *payload*.

In Declarative PM, the de-facto standard for expressing process properties is DECLARE (van der Aalst, Pesic, and Schonenberg 2009), a temporal logic consisting in a set of template formulas of the Linear-time Temporal Logic over finite traces (LTL_f) (De Giacomo and Vardi 2013); here, we use a strictly more expressive extension, which we call *local* LTL_f, i.e., L-LTL_f. This logic features a simple automata-based machinery that facilitates its manipulation, while retaining the ability to express virtually all the properties of interest in declarative PM. Specifically, L-LTL_f subsumes DECLARE and even its multi-perspective variant MP-DECLARE (Burattin, Maggi, and Sperduti 2016) without timestamps and *correlation conditions* (i.e., conditions that relate the attributes of some event to those of other events), but does not subsume full MP-DECLARE. Observe that since MP-DECLARE does not subsume L-LTL_f either, the two logics are incomparable.

Our goal is to devise techniques for three classical problems in Declarative PM: *Event Log Generation* (Skydanienko et al. 2018), i.e., generating an event log consistent with a declarative model; *Query Checking* (Räim et al. 2014), i.e., discovering hidden temporal properties in an event log; and *Conformance Checking* (Burattin, Maggi, and Sperduti 2016), i.e., checking whether the traces of an event log conform to a process model. The main challenge is dealing with data: in the data-aware framework, Query Checking is still open, while existing tools for the other two problems do not scale well.

We put forward Answer Set Programming (ASP (Niemelä 1999)) as an effective solution approach. ASP natively provides data-manipulation functionalities which allow for formalizing data-aware constraints and has experienced over the last years a dramatic improvement in solution performance, thus results in a natural and promising candidate approach for addressing the problems of our interest. We show how such problems can be conveniently modeled as ASP programs, thus solved with any solver. Using the state-of-the-art solver Clingo[1] (Gebser et al. 2011), we experimentally compare our approach against existing ones for Log Generation and Conformance Checking, and show effectiveness of the (first) approach for Query Checking in a data-aware setting. Besides providing an actual solution

[1] potassco.org

technique, ASP facilitates reuse of specifications: the ASP encodings we propose here, indeed, differ in very few, although important, details.

Previous related work include (Wieczorek, Jastrzab, and Unold 2020), where ASP is used to infer a finite-state automaton that accepts (resp. rejects) traces from a positive (negative) input set. This can be seen as a form of Declarative Process Discovery, i.e., the problem of obtaining a (declarative) process specification, which is complementary to the problems we address here. Our approach is similar, in that we use automata to model temporal properties. However, we propose a different automata encoding and show the effectiveness of the approach on three different problems. Another related paper is (Heljanko and Niemelä 2003), which shows how ASP can be used to check a Petri Net against an LTL specification, up to a bounded time horizon. Differently from our work, it: (i) deals with LTL over infinite, as opposed to finite, runs; (ii) adopts a prescriptive, as opposed to declarative, approach; and (iii) does not deal with data in events.

From a broader perspective, we finally observe that while we deal with a set of specific problems, the work paves the way for ASP to become a general effective approach to Declarative PM.

The Framework

An *activity (signature)* is an expression of the form $A(a_1, \ldots, a_{n_A})$, where A is the *activity name* and each a_i is an *attribute name*. We call n_A the *arity* of A. The attribute names of an activity are all distinct, but different activities may contain attributes with matching names. We assume a finite set Act of activities, all with distinct names; thus, activities can be identified by their name, instead of by the whole tuple. Every attribute (name) a of an activity A is associated with a *type* $D_A(a)$, i.e., the set of values that can be assigned to a when the activity is executed. For simplicity, we assume that all domains are equipped with the standard relations $<, \leq, =, \geq, >$. All results can be immediately adapted if some relations are absent in some domain.

An *event* is the execution of an activity (at some time) and is formally captured by an expression of the form $e = A(v_1, \ldots, v_{n_A})$, where $A \in Act$ is an activity name and $v_i \in D_A(a_i)$. The properties of interest in this work concern *(log) traces*, formally defined as finite sequences of events $\tau = e_1 \cdots e_n$, with $e_i = A_i(v_1, \ldots, v_{n_{A_i}})$. Traces model process executions, i.e., the sequences of activities performed by a process instance. A finite collection of executions into a set L of traces is called an *event log*.

L-LTL$_f$

We adopt a declarative approach to process modeling, meaning that processes are specified through a set of constraints over their executions, i.e., over the traces they produce. The formal language we use to express properties of traces is a variant of the Linear-time Logic over finite traces, LTL$_f$ (De Giacomo and Vardi 2013), adapted to deal with the data attributes of activities. We call such variant *Linear-time Logic with local conditions over finite traces*, or *local* LTL$_f$ for short, and denote it as L-LTL$_f$.

Given a finite set of activities Act, the formulas φ of L-LTL$_f$ over Act are inductively defined as follows:

$$\varphi = \mathbf{true} \mid A \mid a \odot a' \mid a \odot v \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{X}\varphi \mid \varphi \mathbf{U} \varphi,$$

where: a and a' are attribute names from some activity in Act, $v \in D_A(a)$, for some $A \in Act$, $\odot$ is an operator from $\{<, \leq, =, \geq, >\}$, and A is an activity name from Act. Formulas of the form $\mathbf{true}$, A, $a \odot a'$, and $a \odot v$ are called *atomic*; formulas not containing the operators $\mathbf{X}$ and $\mathbf{U}$ are called *event formulas*.

The logic is interpreted over positions of finite traces. Formula $\mathbf{true}$ holds at every position; A checks whether activity A occurs in the trace at the given position; $a \odot a'$ (resp. $a \odot v$) compares the value assigned to attribute a with that of attribute a' (resp. with that of value v), at the given position; boolean operators combine formulas as usual; the *next* operator $\mathbf{X}\varphi$ checks whether φ holds in the suffix starting at the next position; finally, the *until* operator $\varphi \mathbf{U} \varphi'$ checks whether φ' is satisfied at some position k, and whether φ holds in all positions that precede k, up to the given position.

Formally, we define by induction when a trace $\tau = e_1 \cdots e_n$ *satisfies* an L-LTL$_f$ formula φ at position $1 \leq i \leq n$, written $\tau, i \models \varphi$, as follows:

- $\tau, i \models \mathbf{true}$;
- $\tau, i \models A$ iff $e_i = A(v_1, \ldots, v_{n_A})$;
- $\tau, i \models a \odot a'$ iff for $e_i = A_i(v_1, \ldots, v_{n_{A_i}})$ and $A_i(a_1, \ldots, a_{n_{A_i}})$ the signature of A_i, it is the case that, for some j and k, $a = a_j$, $a' = a_k$, and $v_j \odot v_k$; [2]
- $\tau, i \models a \odot v$ iff for $e_i = A_i(v_1, \ldots, v_{n_{A_i}})$ and $A_i(a_1, \ldots, a_{n_{A_i}})$ the signature of A_i, it is the case that, for some j, $a = a_j$ and $v_j \odot v$;
- $\tau, i \models \varphi_1 \wedge \varphi_2$ iff $\tau, i \models \varphi_1$ and $\tau, i \models \varphi_2$;
- $\tau, i \models \neg\varphi$ iff $\tau, i \not\models \varphi$;
- $\tau, i \models \mathbf{X}\varphi$ iff $i < n$ and $\tau, i+1 \models \varphi$;
- $\tau, i \models \varphi_1 \mathbf{U} \varphi_2$ iff there exists $j \leq n$ s.t. $\tau, j \models \varphi_2$ and for every $1 \leq k \leq j-1$, it is the case that $\tau, k \models \varphi_1$.

Notice that while, in general, the satisfaction of an L-LTL$_f$ formula φ at some position i of τ depends on the whole trace, and precisely on the suffix of τ starting at position i, event formulas depend only on the event at i.

As in standard LTL$_f$, $\mathbf{X}$ denotes the *strong next* operator (which requires the existence of a next event where the inner formula is evaluated), while $\mathbf{U}$ denotes the *strong until* operator (which requires the right-hand formula to eventually hold, forcing the left-hand formula to hold in all intermediate events). The following are standard abbreviations: $\varphi_1 \vee \varphi_2 \doteq \neg(\neg\varphi_1 \wedge \neg\varphi_2)$; $\varphi_1 \rightarrow \varphi_2 \doteq \neg\varphi_1 \vee \varphi_2$; $\mathbf{F}\varphi = \mathbf{true}\mathbf{U}\varphi$ (*eventually* φ); and $\mathbf{G}\varphi = \neg\mathbf{F}\neg\varphi$ (*globally*, or *always*, φ).

Through L-LTL$_f$ one can express properties of process traces that involve not only the process control-flow but also the manipulated data.

[2] Notice that this requires *compatibility* between the domains $D_{A_i}(a_j)$ and $D_{A_i}(a_k)$ wrt relation $\odot$.

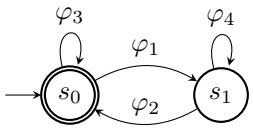

Figure 1: Automaton for the *Response* constraint.

Example 1 *The* L-LTL$_f$ *formula* $\varphi = \mathbf{G}(a \rightarrow \mathbf{F}b)$*, a so-called* Response *constraint, says that whenever activity* a *occurs, it must be eventually followed by activity* b*. A possible* data-aware *variant of* φ *is the formula* $\varphi' = \mathbf{G}((a \wedge n < 5) \rightarrow \mathbf{F}b)$*, which says that whenever activity* a *occurs with attribute* n *less than* 5*, it must be followed by activity* b*.*

Formulas of LTL$_f$, thus L-LTL$_f$, have the useful property of being fully characterized by finite-state, possibly nondeterministic, automata. Specifically, for every L-LTL$_f$ formula φ there exists a finite-state automaton (FSA) $\mathcal{A}_\varphi$ that accepts all and only the traces that satisfy φ (De Giacomo and Vardi 2013). Such automata are standard FSA with transitions labelled by event formulas. For a fixed set of activities Act, let Γ_{Act} be the set of event formulas over Act. An FSA over a set of activities Act is a tuple $\mathcal{A} = \langle Q, q_0, \rho, F \rangle$, where:

- Q is a finite set of *states*;
- $q_0 \in Q$ is the automaton *initial state*;
- $\delta \subseteq Q \times \Gamma_{Act} \times Q$ is the automaton *transition relation*;
- $F \subseteq Q$ is the set of automaton *final states*.

Without loss of generality, we assume that formulas labeling transitions are conjunctions of literals. It is immediate to show that every FSA can be rewritten in this way.

A *run* of an FSA $\mathcal{A}$ on a trace $\tau = e_1 \cdots e_n$ (over Act) is a sequence of states $\rho = q_0 \cdots q_n$ s.t. for all $0 \leq i \leq n-1$ there exists a transition $\langle q_i, \gamma, q_{i+1} \rangle \in \delta$ s.t. $\tau, i+1 \models \gamma$. A trace τ over Act is *accepted* by $\mathcal{A}$ iff it *induces* a *run* of $\mathcal{A}$ that ends in a final state. Notice that satisfaction of γ, this being an event formula, can be established by looking at each event e_{i+1} at a time, while disregarding the rest of the trace; thus, in order to construct the induced run ρ, one can proceed in an online fashion, as the next event arrives, by simply triggering, at every step, a transition outgoing from q_i whose label is satisfied by the event.

Example 2 *Consider again the formulas* φ *and* φ' *shown above, and the (paramtric) automaton depicted in Fig 1. It is easy to see that for* $\varphi_1 = a$*,* $\varphi_2 = b$*,* $\varphi_3 = \neg a$*,* $\varphi_4 = \neg b$*, the resulting automaton accepts all and only the traces that satisfy* φ*, as well as that for* $\varphi_1 = a \wedge n < 5$*,* $\varphi_2 = b$*,* $\varphi_3 = \neg a$*,* $\varphi_4 = \neg b$*, the obtained automaton accepts all and only the traces that satisfy* φ'*.*

The details about the construction of $\mathcal{A}_\varphi$ from φ are not in the scope of this work, and we refer the interested reader to (De Giacomo and Vardi 2013) for more information; we rely on the results therein. We observe that while the automaton construction is time-exponential in the worst-case, wrt the size of the input formula φ, tools exist, such as Lydia[3] (De Giacomo and Favorito 2021), which exhibit efficient performance in practice; this, combined with the fact that the specifications of practical interest are typically small, makes the approaches based on automata construction usually feasible in practice. We can now formalize the problems addressed in this paper.

Event Log Generation. Given a set Φ of L-LTL$_f$ formulas over a set of activities Act and a positive integer t, return a trace τ over Act of length t s.t., for every formula $\varphi \in \Phi$, it is the case that $\tau \models \varphi$. In words, the problem amounts to producing a trace of length t over Act that satisfies all the input constraints in Φ. A more general version of the problem requires to generate a log L of n traces of fixed length t satisfying the constraints. For simplicity, we consider the first formulation.

Query Checking. Query Checking takes as input formulas from the extension of L-LTL$_f$ *with activity variables*, defined as follows:

$$\varphi = \mathbf{true} \mid ?V \mid A \mid a \odot a' \mid a \odot v \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{X}\varphi \mid \varphi \mathbf{U} \varphi,$$

where symbols starting with "?" are *activity variables* and the other symbols are as in L-LTL$_f$ (given Act).

Given an L-LTL$_f$ formula with activity variables, by assigning an activity (from Act) to every variable, we obtain a "regular" L-LTL$_f$ formula. Formally, for an L-LTL$_f$ formula φ (over Act), containing a (possibly empty) set of activity variables $Vars_\varphi$, an *assignment* to $Vars_\varphi$ is a total function $\nu : Vars_\varphi \rightarrow Act$. Given φ and an assignment ν to its activity variables, $\varphi[\nu]$ denotes the (regular) L-LTL$_f$ formula obtained by replacing, in φ, every variable symbol $?V$ with an activity name from Act. Observe that if $Vars_\varphi = \emptyset$ there exists only one assignment ν and $\varphi = \varphi[\nu]$. Given a trace τ, since $\varphi[\nu]$ is a regular L-LTL$_f$ formula, we can check whether $\tau \models \varphi[\nu]$.

An instance of Query Checking consists in a log L and an L-LTL$_f$ formula φ with activity variables $Vars_\varphi$; a solution is a set Λ of assignments to $Vars_\varphi$ s.t. for every assignment $\nu \in \Lambda$ and every trace $\tau \in L$, it holds that $\tau \models \varphi[\nu]$.

In words, query checking requires to find a set Λ of assignments ν, each transforming the input formula φ into an L-LTL$_f$ formula $\varphi[\nu]$ satisfied by all the traces of the input log L. Observe that φ variables can only span over activities.

Conformance Checking. Given a trace τ and a set Φ of L-LTL$_f$ formulas, both over the same set of activities Act, check whether, for all formulas $\varphi \in \Phi$, $\tau \models \varphi$. The problem can also be defined in a more general form, where τ is replaced by a log L of traces over Act and the task requires to check whether for all the traces τ of L and all $\varphi \in \Phi$, it holds that $\tau \models \varphi$.

Answer Set Programming (ASP)

An ASP program consists in a set of *rules* which define *predicates* and impose relationships among them. The task of an ASP *solver* is that of finding a finite *model* of the program, i.e., an interpretation of the predicates that satisfies

[3] github.com/whitemech/lydia/releases/tag/v0.1.1

the program rules. ASP rules are written in a fragment of (function-free) First-order Logic (FOL) extended with a special *negation-as-failure* (NAF) operator (in addition to classical negation) which allows for distinguishing facts that are false from facts that are unknown. The presence of this operator, combined with the classical FOL negation, has a huge impact on the programs one can write and the way models are found. Here, we do not discuss these details, referring the interested reader to (Gelfond and Lifschitz 1988; Niemelä 1999). For our purposes, it will be sufficient restricting to the class of rules with the NAF operator as the only available negation operator (that is, disallowing classical negation).

Syntax

The basic constructs of ASP programs are: 1. *constants*, identified by strings starting with a lower-case letter; 2. *variables*, identified by strings starting with an upper-case letter; 3. *terms*, i.e., constants or variables; 4. *atoms*, i.e., expressions of the form $p(t_1, \ldots, t_n)$, where p is a *predicate*, identified by a string, and each t_i is a *term*. A predicate p is said to have arity n if it occurs in an expression of the form $p(t_1, \ldots, t_n)$. An atom containing only constant terms is said to be *ground*.

ASP rules are obtained by combining the basic elements through boolean operators and the NAF operator. In this work, we use rules of the following form:

$$a \leftarrow l_1, \ldots, l_n, \texttt{not}\ l_{n+1}, \ldots, \texttt{not}\ l_m$$

where a and each l_i are atoms $p(t_1, \ldots, t_n)$, `not` denotes the NAF operator, and every variable occurring in the rule also occurs in some atom $l_1, \ldots, l_n$. The left-hand side is called the rule's *head* and is optional. When the head is absent, the rule is called an *integrity constraint*. The right-hand side is called the *body* and can be left empty, in which case the $\leftarrow$ symbol is omitted and the rule is called a *fact*.

Semantics

Intuitively, a *model* of an ASP program $\mathcal{P}$ is a set of ground atoms that satisfies all program rules. In general, many models exist. Among these, only those that are minimal wrt set inclusion and that contain a ground atom only "if needed", i.e., if it occurs as the head of a ground rule, are taken as solutions, called the *answer sets* of $\mathcal{P}$ in the ASP terminology. The task of an ASP solver is to compute such sets.

Given an ASP program $\mathcal{P}$ and a rule $r \in \mathcal{P}$, the set of r *ground instantiations* is the set $\mathcal{G}(r)$ of rules obtained by replacing all the variables in r with all the constants mentioned in $\mathcal{P}$ (the so-called *Herbrand universe* of $\mathcal{P}$), in all possible ways, so that all rules in $\mathcal{G}(r)$ contain only ground atoms. Then, the *ground* instantiation $\mathcal{G}(\mathcal{P})$ of a program $\mathcal{P}$ is the union of all the ground instantiations of its rules, i.e., $\mathcal{G}(\mathcal{P}) = \bigcup_{r \in \mathcal{P}} \mathcal{G}(\mathcal{P})$.

An *interpretation* I of a program $\mathcal{P}$ is a set of ground atoms $p(c_1, \ldots, c_n)$, where p is a predicate of arity n occurring in $\mathcal{P}$ and $c_1, \ldots, c_n$ are constants from the Herbrand universe of $\mathcal{P}$. Given a *positive* (i.e., without occurrences of `not`) program $\mathcal{P}$, an interpretation I is a *model* of $\mathcal{P}$ if, for every ground rule, $a \leftarrow l_1, \ldots, l_n$ in $\mathcal{G}(\mathcal{P})$, whenever $l_i \in I$ for $i = 1, \ldots, l_n$, it holds that $a \in I$. An *answer set* of $\mathcal{P}$ is a model I that is minimal wrt set inclusion.

The semantics of general programs is obtained as a reduction to positive programs. Namely, the *reduct* of a ground program $\mathcal{G}(\mathcal{P})$ wrt an interpretation I is the positive ground program $\mathcal{G}(\mathcal{P})^I$ obtained by:

- deleting all the rules

  $$a \leftarrow l_1, \ldots, l_n, \texttt{not}\ l_{n+1}, \ldots, \texttt{not}\ l_m$$

  of $\mathcal{G}(\mathcal{P})$ s.t. $l_i \in I$ for some $i \in \{n+1, \ldots, m\}$;
- replacing all the remaining rules

  $$a \leftarrow l_1, \ldots, l_n, \texttt{not}\ l_{n+1}, \ldots, \texttt{not}\ l_m$$

  with $a \leftarrow l_1, \ldots, l_n$.

Intuitively, the first transformation removes a rule, as already satisfied by I; the second transformation removes the so-called *negative body* of the rule, because it is satisfied. As it can be easily seen, the resulting program $\mathcal{G}(\mathcal{P})^I$ does not mention the `not` operator. The interpretation I is an *answer set* of $\mathcal{P}$ if it is an answer set of $\mathcal{G}(\mathcal{P})^I$.

In this work, we do not discuss the algorithms to compute the answer sets of a program, but focus on how the problems of our interest can be encoded in ASP and then solved by an ASP solver, in such a way that the returned Answer Sets represent the solution to our problems. This is the focus of the next section. For the experiments, we use the state-of-the-art solver Clingo.

ASP for Declarative Process Mining

We encode Log Generation, Conformance Checking, and Query Checking into ASP programs. For every L-LTL$_f$ formula φ we deal with, we assume available the corresponding automaton $\mathcal{A}_\varphi$. The three programs share some common parts, such as the automata and the traces, which are modeled through suitable predicates and ASP rules. Each encoding re-uses some of these parts, possibly customized, together with additional fragments used to model problem-specific features.

Activities are captured by the unary predicate $act(A)$, where A is the activity name. In the presence of data, *activity signatures* are modeled by the binary predicate $has\_attr(A, N)$, where A is the activity name and N is the attribute name. Attributes may be typed by stating the set of values they can take, through predicate $val(N, V)$, where N is the attribute name and V one of its possible values. A *trace* is modeled by the binary predicate $trace(A, T)$, where A is the activity and T the time point where it occurs. Time points come from predicate $time(T)$, which contains the values $\{0, \ldots, \ell\}$, for ℓ the trace length. The trace is defined on time points from 0 to $\ell - 1$. In the presence of data, activity attributes are paired with values through predicate $has\_val(T, N, V)$, where T is the time point, N the attribute name, and V the assigned value. Notice that the association is based on the time point (exactly one activity is performed at one time point). Simple integrity constraints are used to ensure that the mentioned attributes belong in

fact to the activity and that the sassigned value comes from the corresponding type.

Automata are encoded with predicates $init(S)$, $acc(S)$, $trans(S, F, S')$, and $hold(F, T)$. The first and the second one model the initial state and the accepting states of the automaton, the third one models the existence of a transition from S to S' under the event formula represented by integer F, and the last one models satisfaction of (event) formula F at time point T. In the presence of multiple L-LTL$_f$ formulas, each automaton is identified by a unique integer value and an additional parameter is added to the above predicates to refer to the various automata.

Example 3 *The ASP encoding of the automaton for the* LTL$_f$ *formula* $\mathbf{G}(a \rightarrow \mathbf{F}b)$*, shown in Fig. 1, for* $\varphi_1 = a$*,* $\varphi_2 = b$*,* $\varphi_3 = \neg a$*,* $\varphi_4 = \neg b$*, is as follows:*

$$
\begin{array}{l}
init(1, s0).\\
acc(1, s0).\\
trans(1, s0, 1, s1).\\
hold(1, 1, T) \leftarrow trace(a, T).\\
trans(1, s1, 2, s0).\\
hold(1, 2, T) \leftarrow trace(b, T).\\
trans(1, s0, 3, s0).\\
hold(1, 3, T) \leftarrow \texttt{not}\ hold(1, 1, T), time(T).\\
trans(1, s1, 4, s1).\\
hold(1, 4, T) \leftarrow trace(A, T), activity(A), A \neq b.
\end{array}
$$

where a *and* b *are activities and each formula* φ_i *(*$i = 1, \ldots, 4$*) is identified by index* i *in the encoding.*

In a data-aware setting, conditions on data can be simply added to the rules for $hold$*s. For example the following rule:*

$$
\begin{array}{l}
hold(1, 1, T) \leftarrow\\
\quad trace(a, T), has\_value(number, V, T), V < 5.
\end{array}
$$

expresses the fact that the event formula φ_1 *holds at time* T *if activity* a *occurs at time* T *in the trace, with a value less than* 5 *assigned to its attribute* $number$*.*

To capture satisfaction of an L-LTL$_f$ formula φ by a trace τ, we model the execution of the automaton $\mathcal{A}_\varphi$ on τ. To this end, we introduce predicate $state(I, S, T)$, which expresses the fact that automaton (with index) I is in state S at time T. Since the automaton is nondeterministic in general, it can be in many states at time point T (except for the initial one). The rules defining $state$ are the following:

$$
\begin{array}{l}
state(I, S, 0) \leftarrow init(I, S).\\
state(I, S', T) \leftarrow state(I, S, T-1),\\
\qquad\qquad trans(I, S, F, S'), hold(I, F, T-1).
\end{array}
$$

The first one says that at time point 0 every automaton I is in its respective initial state. The second one says that the current state of automaton I at time point T is S' whenever the automaton is in state S at previous time point $T-1$, the automaton contains a transition from S to S' under some event formula with index F and the formula holds at time $T-1$ in the trace.

Finally, the fact that a trace is accepted by all automata, i.e., that the trace satisfies the corresponding formulas, is stated by requiring that, for each automaton, at least one of the final states be accepting ($tlength(T)$ denotes the length T of the trace):

$$\leftarrow \{state(I, S, T) : \texttt{not}\ acc(I, S), tlength(T)\} = 0.$$

Next, we use these fragments to describe the ASP encodings for the problems of interest. For lack of space, we discuss only the main rules.

Event Log Generation The encoding schema of Event Log Generation is as follows:

1. Activities, attributes, attribute types, and trace length are provided as input and formalized as discussed above.
2. For each input L-LTL$_f$ constraint φ, the corresponding automaton $\mathcal{A}_\varphi$ is generated and modeled as discussed, using a unique integer value to identify it.
3. Suitable integrity constraints are defined to ensure that: each time point in the trace has exactly one activity; every attribute is assigned exactly one value; and the attributes assigned at a given time point actually belong to the activity occurring at that time point.
4. Finally, predicate $state$ is defined as above and it is required that every automaton ends up in at least one final state at the last time point.
5. Predicates $trace$ and has_val contain the solution, i.e., they model a sequence of activities whose attributes have an assigned value, which satisfies all the input constraints.

Query Checking The ASP specification of query checking is analogous to that of Log Generation except for the following. Firstly, the problem takes as input a set of fully specified traces. This is dealt with in a simple way, by adding a parameter to predicate $trace$ representing the (unique) identifier of the trace and, consequently, by adding such parameter to all the predicates that depend on $trace$ (e.g., has_val, $hold$, $state$). Secondly, the input L-LTL$_f$ formulas contain activity variables. To deal with them, additional predicates $var(V)$ and $assgnmt(V, W)$ are introduced to account for, respectively, variables V and assignments of value W to variable V. Besides this, the automata $\mathcal{A}_{\varphi_i}$ associated with the formulas are obtained by treating activity variables as if they were activity symbols (without affecting the construction, which does not consider the semantics of such objects), thus obtaining automata whose transitions are labelled by event formulas, possibly containing activity variables instead of activity symbols. Such formulas become regular event formulas once values are assigned to variables and can thus be evaluated on the (events of the) input trace. Formally, this requires a slightly different definition of predicate $hold$, which must now take $assgnmt$ into account. To see how this is done, consider the formula $\varphi = \mathbf{G}((?A1 \wedge number < 5) \rightarrow \mathbf{F}?A2)$. The corresponding automaton is the same as that of Fig. 1, where $\varphi_1 = ?A1 \wedge number < 5$, $\varphi_2 = ?A2$, $\varphi_3 = \neg\varphi_1$, and $\varphi_4 = \neg\varphi_2$. For formula φ_1, we have the following definition of predicate $hold$:

$$
\begin{array}{l}
hold(1, 1, J, T) \leftarrow trace(J, A, T), assgnmt(varA1, A),\\
\qquad\qquad has\_val(J, integer, V, T), V < 5.
\end{array}
$$

The parameter J stands for the trace identifier, as discussed above. The above rule generalizes the corresponding one in Log Generation in the presence of activity variable $?A1$. As it can be seen, in order to evaluate formula φ_1 (second parameter in $hold$) of automaton 1 (first parameter), such variable (modeled as $varA1$) must be instantiated first, through predicate $assgnmt$. Observe that once all variables are assigned a value, the whole formula φ becomes variable-free, and the corresponding automaton is a regular automaton. The returned extensions of $assgnmt$ and has_val represent, together, the problem solution.

Conformance Checking Conformance Checking can be seen as a special case of Query Checking with a single input trace and where all input formulas are variable-free. In this case, the problem amounts to simply checking whether the whole specification is consistent, which is the case if and only if the input trace, together with the assignments to the respective activity attributes, satisfy the input formulas.

We close the section by observing how these problems provide a clear example of how the declarative approach allows for specification reuse. All the specifications, indeed, share the main rules (for trace, automaton, etc.) and are easily obtained as slight variants of each other, possibly varying the (guessed) predicates representing the solution.

Experiments

In this section, we provide a comparison with state-of-the-art tools for log generation and conformance checking based on multi-perspective declarative models. In addition, we provide an estimate about the scalability of our query checking tool, which is, instead, totally novel. The state-of-the art tool used for log generation is the one presented in (Skydanienko et al. 2018) based on Alloy and that is tailored for MP-Declare. We will see that our ASP implementation of the log generation task is much more scalable than the Alloy-based one and, at the same time, supports a more expressive data-aware rule language. The tool used for conformance checking is the one presented in (Burattin, Maggi, and Sperduti 2016). Here, the execution times are comparable but the state-of-the-art tool used is tailored for Declare and optimized to check the conformance with respect to Declare rules only. The experiments have been conduced on a standard laptop Dell XPS 15 with an intel i7 processor and 16GB of RAM. All the execution times discussed in this section have been averaged over 3 runs. Source code, declarative models and event logs used in the experiments are available at https://github.com/fracchiariello/process-mining-ASP.

Log Generation

For testing the log generation tools, we have used 8 synthetic models and 8 models derived from real life logs. The experiments with the synthetic models allowed us to test the scalability of the tools in a controlled environment and using models with specific characteristics. The experiments with the real models have been used to test the tools in real environments. For the experiments with the synthetic models, we built 8 reference models containing 3, 5, 7, and 10 constraints with and without data conditions. Each model was obtained from the previous one by adding new constraints and preserving those already present. Times are in ms.

Table 1: Log Generation (times in ms)

# constr. → / Trace len ↓	3	5	7	10
10	35975	35786	36464	37688
15	50649	51534	54402	54749
20	69608	70342	73122	73222
25	85127	85598	87065	89210
30	101518	101882	106062	107520
10	18733	18947	19539	20007
15	25700	25723	27344	26897
20	32047	33837	33107	33615
25	39114	38666	40556	41055
30	46207	46706	47613	49410
10	595	614	622	654
15	876	894	904	956
20	1132	1155	1178	1250
25	1364	1413	1444	1543
30	1642	1701	1746	1874
10	249	270	289	340
15	349	390	408	457
20	436	496	538	601
25	519	568	611	712
30	622	666	726	837

Table 2: Log Generation, real life (times in ms)

Model (80) → / Trace len ↓	BPI2012	DD	ID	PL	PTC	RP	RT	Sepsis
10	656	100*	726*	3901	1183	119*	319	460
15	817	887	2865	4538	1820	1069	353	564
20	846	832	3160	4102	2194	813	860	640
25	1061	930	4129	6169	2889	1063	483	780
30	1433	1026	5226	9231	2370	1220	630	923
10	31935	2364*	30762*	59468	65783	2703*	24909	38241
15	43337	58572	152188	85942	97098	66641	34408	57178
20	57596	80665	237777	122511	146420	95005	44608	85808
25	72383	118975	359665	174596	221434	134851	54808	120110
30	86910	181027	563794	236697	330753	187972	63379	174838

Table 1 shows, in the first and the second block, the execution times for the Alloy log generation with and without data conditions in the declarative models; in the third and the fourth blocks, the results obtained with the ASP-based log generator with and without data. The table shows the execution times needed to generate logs of 10000 traces (of lenght ranging from 10 to 30). Consistent results are obtained also on additional experiments for logs of size between 100 and 5000, not reported here for space reasons.

The results obtained with models containing data conditions show that the ASP-based tool scales very well, requiring less than 2 sec in the worst case. This occurs when a model with 10 constraints is used to generate 10000 traces of length 30. As expected, the execution time increases linearly when the length of the traces in the generated logs increases. The number of constraints in the declarative model also affects the tool performance but with a lower impact.

Without data conditions the results are similar but, as expected, the execution time is lower and increases less quickly when the complexity of the model and of the generated log increases. In the worst case (of a model containing 10 constraints used to generate 10000 traces of length 30), the execution time is lower than 1 second.

The results obtained with the tool based on Alloy follow the same trends but the execution times with the Alloy-based tool are almost 60 times higher than the ones obtained with the tool based on ASP.

The real life logs used in the experiments are taken from the real life log collection available at https://data.4tu.nl/. We

Table 3: Conformance Checking (times in ms)

Tool →	ASP		Declare Analyzer	
Trace Len ↓	data	no data	data	no data
10	665	635	598	110
15	1100	1035	805	145
20	1456	1354	1092	155
25	2071	1896	1273	177
30	2407	2219	1337	215

Table 4: Conformance Checking (times in ms)

	BPI2012	DD	ID	PL	PTC	RP	RT	Sepsis
60	33426	13084	49969	78625	8412	9354	49501	7116
70	33242	13245	46388	55475	5596	9359	35537	3796
80	24482	10176	29969	33775	4699	6836	35483	1778
90	8445	4568	17576	26590	2787	5608	35483	731
60	2882	2771	9800	8521	1549	2122	15262	1194
70	2852	3249	7416	5358	959	2102	10351	705
80	2291	2103	3993	2677	755	1532	11285	318
90	1691	1525	1946	1595	404	1091	10628	250

used the declarative process model discovery tool presented in (Maggi et al. 2018) to extract a process model using the default settings. The models in the real cases are much more complex and include a number of constraints between 10 and 49 for a minimum support of 80. The execution times needed for the log generation task with the ASP-based log generator and with the tool based on Alloy are shown in Table 2 (first and second block, respectively). The asterisk indicates that for that specific model was not possible to generate 10000 unique traces. The complexity of the real life models makes even more evident the significant advantage of using the ASP-based tool with respect to Alloy. In particular, in the worst case, the ASP-based tool requires around 9 seconds (to generate 10000 traces of length 30 for log PL) while the Alloy-based generator takes almost 4 minutes.

Conformance checking

Also for Conformance Checking, we used synthetic and real life datasets. The former include the same declarative models as Log Generation, plus synthetic logs of 1000 traces of lengths from 10 to 30. Table 3 shows the execution times for the ASP-based conformance checking with and without data conditions and the results obtained with the Declare Analyzer tool *for the syntehtic datasets* (times in ms). The results show that in all cases the execution times increase when the model becomes larger and the traces in the log become longer. The execution times obtained with Clingo and the Declare Analyzer are comparable for data-aware constraints. On the other hand, when the constraints in the models do not contain data conditions, the Declare Analyzer is around 5 times faster with respect to our approach. This might be due to the use of the `#max` aggregate to infer a trace's length, which yields a performance degradation. A possible solution could be computing the trace length in advance and then provide it in the ASP encoding as a fact.

In the real life experiments, we tested the conformance checking tools using models obtained with the discovery tool by varying the minimum support between 60 and 90. The minimum support indicates the min percentage of traces in which a constraint should be fulfilled to be added to the discovered model. Clearly, a higher minimum support implies that the discovered models contain less constraints. As expected (see Table 4), the execution times decrease when the minimum support used to discover the reference models increases. Also in this case, the Declare Analyzer (second block in Table 4) is faster. However, the ASP-based tool also scales well (first block in Table 4) requiring in the worst case around 1min for Conformance Checking.

Table 5: ASP Query Checking (times in ms)

Constraints →	Existence	Responded	Response	Chain	Absence	Not Resp.	Not Resp.	Not Chain
Trace len ↓		Existence		Response		Existence		Response
10	521	736	534	503	566	783	602	385
15	704	1113	801	788	784	1180	879	606
20	1321	1675	1143	1128	1373	1821	1304	865
25	1397	3218	1528	1561	1562	2823	1807	1104
30	1674	2878	1824	1906	1905	2784	2028	1301
10	399	658	541	632	441	799	806	772
15	616	1183	824	1057	595	1319	1121	1182
20	903	1778	1339	1550	874	1887	2127	2062
25	1188	2381	1724	2036	1101	3246	3200	2486
30	1461	3278	2066	2632	1333	3391	2766	2846

Query checking

Since for Query Checking no competitor exists in the PM literature, we run a set of controlled experiments to check how execution times vary under different conditions. We used the same synthetic logs used for Conformance Checking. We tested 8 queries corresponding to 8 standard Declare templates with and without data conditions. The results are shown in Table 5 (with and without data in the first and second block respectively). The execution times are comparable for different types of queries and the presence of data do not affect the performance. In addition, as expected, the execution times increase when the traces in the log become longer.

Conclusions

We have devised an ASP-based approach to solve three classical problems from Declarative PM, namely Log Generation, Query Checking and Conformance Checking, in a data-aware setting. Our results include correct ASP-encoding schemata and an experimental evaluation against other approaches. The experimental results show that our approach drastically improves the performance on Log Generation with respect to the state-of-the-art tool from PM. Time performance are slightly worse wrt to the existing ad-hoc conformance checking, optimized for the Declare language. The approach provides the first solution to Query Checking in a data-aware setting, a problem still open so far. We believe that, by showing how the selected problems can be encoded and solved with ASP, we are not only offering a solution technique but, more in general, we are putting forward ASP an effective modeling paradigm for Declarative PM in a data-aware setting. For future work, we plan to extend the approach to deal with actual, non-integer, timestamps in events and to go beyond local LTL_f by investigating the introduction of *across-state* quantification to relate the values assigned to attributes at a given time point to those assigned at a different time point.

Acknowledgments

Work partly supported by the ERC Advanced Grant WhiteMech (No. 834228), the EU ICT-48 2020 project TAILOR (No. 952215), and the Sapienza Project DRAPE: DataawaRe Automatic Process Execution.

References

Burattin, A.; Maggi, F. M.; and Sperduti, A. 2016. Conformance checking based on multi-perspective declarative process models. *Expert Syst. Appl.*, 65: 194–211.

De Giacomo, G.; and Favorito, M. 2021. Compositional Approach to Translate LTLf/LDLf into Deterministic Finite Automata. In Biundo, S.; Do, M.; Goldman, R.; Katz, M.; Yang, Q.; and Zhuo, H. H., eds., *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS 2021, Guangzhou, China (virtual), August 2-13, 2021*, 122–130. AAAI Press.

De Giacomo, G.; and Vardi, M. Y. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In Rossi, F., ed., *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 854–860. IJCAI/AAAI.

Gebser, M.; Grote, T.; Kaminski, R.; and Schaub, T. 2011. Reactive Answer Set Programming. In Delgrande, J. P.; and Faber, W., eds., *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, volume 6645 of *Lecture Notes in Computer Science*, 54–66. Springer.

Gelfond, M.; and Lifschitz, V. 1988. The Stable Model Semantics for Logic Programming. In Kowalski, R.; Bowen; and Kenneth, eds., *Proceedings of International Logic Programming Conference and Symposium*, 1070–1080. MIT Press.

Heljanko, K.; and Niemelä, I. 2003. Bounded LTL model checking with stable models. *Theory Pract. Log. Program.*, 3(4-5): 519–550.

Maggi, F. M.; Ciccio, C. D.; Francescomarino, C. D.; and Kala, T. 2018. Parallel algorithms for the automated discovery of declarative process models. *Inf. Syst.*, 74(Part): 136–152.

Niemelä, I. 1999. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Ann. Math. Artif. Intell.*, 25(3-4): 241–273.

Räim, M.; Ciccio, C. D.; Maggi, F. M.; Mecella, M.; and Mendling, J. 2014. Log-Based Understanding of Business Processes through Temporal Logic Query Checking. In *On the Move to Meaningful Internet Systems: OTM 2014 Conferences - Confederated International Conferences: CoopIS, and ODBASE 2014, Amantea, Italy, October 27-31, 2014, Proceedings*, 75–92.

Skydanienko, V.; Francescomarino, C. D.; Ghidini, C.; and Maggi, F. M. 2018. A Tool for Generating Event Logs from Multi-Perspective Declare Models. In van der Aalst, W. M. P.; Casati, F.; Kumar, A.; Mendling, J.; Nepal, S.; Pentland, B. T.; and Weber, B., eds., *Proceedings of the Dissertation Award, Demonstration, and Industrial Track at BPM 2018 co-located with 16th International Conference on Business Process Management (BPM 2018), Sydney, Australia, September 9-14, 2018*, volume 2196 of *CEUR Workshop Proceedings*, 111–115. CEUR-WS.org, Raffaele Conforti and Massimiliano de Leoni and Marlon Dumas.

van der Aalst, W. M. P. 2016. *Process Mining - Data Science in Action, Second Edition*. Springer. ISBN 978-3-662-49850-7.

van der Aalst, W. M. P.; Pesic, M.; and Schonenberg, H. 2009. Declarative workflows: Balancing between flexibility and support. *Comput. Sci. Res. Dev.*, 23(2): 99–113.

Wieczorek, W.; Jastrzab, T.; and Unold, O. 2020. Answer Set Programming for Regular Inference. *Applied Sciences*, 10(21): 7700.