A direct ASP encoding for Declare [⋆]

Francesco Chiariello[1][0000−0001−7855−7480], Valeria Fionda[2][0000−0002−7018−1324], Antonio Ielo[2][0009−0006−9644−7975], and Francesco Ricca[2][0000−0001−8218−3178]

[1]University of Naples Federico II, Naples, Italy
`francesco.chiariello@unina.it`
[2]University of Calabria, Rende, Italy
`{valeria.fionda, antonio.ielo, francesco.ricca}@unical.it`

Abstract. Answer Set Programming (ASP), a well-known declarative programming paradigm, has recently found practical application in Process Mining, particularly in tasks involving declarative specifications of business processes. Declare is the most popular declarative process modeling language. It provides a way to model processes by sets of constraints, expressed in Linear Temporal Logic over Finite Traces ($\mathrm{LTL_f}$), that valid traces must satisfy. Existing ASP-based solutions encode a Declare constraint by the corresponding $\mathrm{LTL_f}$ formula or its equivalent automaton, derived using well-established techniques. In this paper, we propose a novel encoding for Declare constraints, which models their semantics *directly* as ASP rules, without resorting to intermediate representations. We evaluate the effectiveness of the novel approach on two Process Mining tasks by comparing it to alternative ASP encodings and a Python library for Declare.

Keywords: Answer Set Programming · Process Mining · Declare

1 Introduction

A process, as defined in the Project Management Body of Knowledge [1], is (sic) *"a set of interrelated actions and activities performed to achieve a specified set of products, results, or services"*, typically performed in a periodic, recurrent or continuous fashion. Process Mining [2] is an interdisciplinary field that analyzes processes, using a blend of formal methods, data science, computer science, and business process management tools. One of the main tasks of Process Mining is *conformance checking*, which evaluates the validity of a particular process execution, referred to as a *trace*, with respect to a *process model*. This process model is a formal mathematical representation that allows for various forms of reasoning related to the underlying process. Process models can be

[⋆] This work was partially supported by the Italian Ministry of Research (MUR) under PRIN project PINPOINT - CUP H23C22000280006, PRIN project HypeKG - CUP H53D23003710006, PRIN PNRR project DISTORT - CUP H53D23008170001, PNRR projects FAIR "Future AI Research" - Spoke 9 - WP9.1 and WP9.2- CUP H23C22000860006 and by the project "Borgo 4.0", POR Campania FESR 2014-2020.

expressed using either *imperative* or *declarative* languages. Imperative process models explicitly describe all possible execution traces and are effective in representing well-structured routine processes. However, their use is impractical when the process involves a large number of activities characterized by intricate coordination patterns. In such cases, declarative process models may be more convenient. Declarative process modeling uses logic-based languages to provide a set of constraints on the possible executions, where every execution is allowed unless a constraint explicitly prohibits it. Linear temporal logic over finite traces ($\mathrm{LTL_f}$) emerged as a natural formalism for declarative process modeling [24]. However, in Process Mining applications, *"free-form"* $\mathrm{LTL_f}$ formulae are rarely used for specification purposes. Instead, a limited set of predefined patterns, derived from the realm of systems' verification literature [14], is exploited. Restricting modeling languages to a set of predefined patterns accomplishes two important objectives: *(i)* it simplifies the tasks of modelers [25], and *(ii)* it paves the way for ad-hoc implementations that may outperform generic $\mathrm{LTL_f}$ techniques in terms of efficiency. In particular, the most widely used declarative process modeling language in Process Mining applications is Declare [4], which consists of a set of patterns ("templates") whose semantics can be formalized in $\mathrm{LTL_f}$. Although, historically, the initial semantics of Declare was not provided in terms of $\mathrm{LTL_f}$, its $\mathrm{LTL_f}$ formalization became the cornerstone of many reasoning tasks within Declare [12]. Answer Set Programming (ASP; [6, 21, 32]) has been used in planning applications to inject domain-dependent knowledge rules, similar to a predefined set of patterns, obtained by encoding an action theory language into ASP [35]. Recently, ASP has been proposed to tackle various computational tasks in Declare-based Process Mining [26, 10]. The work of [11] proposes a solution based on the well-known $\mathrm{LTL_f}$-to-automata translation [22]. This transformation maps a $\mathrm{LTL_f}$ formula φ into a symbolic automaton $\mathcal{M}_\varphi$ such that, given a trace π, $\pi \models \varphi$ if and only if $\mathcal{M}_\varphi$ recognizes π. In a different study [28], authors suggested a method for encoding the semantics of temporal operators into a logic program, enabling the encoding of arbitrary $\mathrm{LTL_f}$ formulae, by a reification of their syntax tree. Nevertheless, as far as we know, there has been no prior attempt to *directly* encode the Declare $\mathrm{LTL_f}$ patterns library using ASP – "directly" refers to an encoding that represents the semantics of Declare constraints without relying on any intermediary translation. In this paper, we fill this gap by proposing a direct encoding of the most common Declare constraints and comparing it to existing ASP-based encodings on several logs commonly used in Process Mining literature [31]. The experimental evaluation aims to achieve two primary objectives: *(i)* compare the ASP-based methods on the conformance and query checking tasks; and, *(ii)* evaluate the performance of our direct encoding approach. Code and data to reproduce our experiments are publicly available at `https://github.com/ainnoot/padl-2024`.

2 Preliminaries

In this section fundamental concepts related to Process Mining, linear temporal logic over finite traces, the Declare process modeling language, and Answer Set Programming are discussed.

2.1 Process Mining

Process Mining [2] is a research area at the intersection of Process Science and Data Science. It leverages data-driven techniques to extract valuable insights from operational processes by analyzing event data (i.e., event logs) collected during their execution. A process can be seen as a sequence of activities that collectively allow to achieve a specific goal. A trace represents a concrete execution of a process recording the exact sequence of events and decisions taken in a specific instance. Process Mining plays a significant role in Business Process Management [36], by providing data-driven approaches for the analysis of events logs directly extracted from enterprise information systems. Typical Process Mining tasks include: *Conformance checking* that aims at verifying if a trace is conformant to a specified model and, for logic-based techniques, *Query Checking* that evaluates *queries* (i.e., formulae incorporating variables) against the event log. Several formalisms can be used in process modelling, with Petri nets [3] and BPMN [37] being among the most widely used, both following an imperative paradigm. Imperative process models explicitly describe all the valid process executions and can be impractical when the process under consideration is excessively intricate. In such cases, declarative process modelling [1] is a more appropriate choice. Declarative process models specify the desired properties (in terms of constraints) that each valid process execution must satisfy, rather than prescribing a step-by-step procedural flow. Using declarative modeling approaches allows to easily specify the desired behaviors: everything that does not violate the rules is allowed. Declarative specifications are typically expressed in Declare [4], Linear Temporal Logic over Finite Traces ($\mathrm{LTL_f}$) [16], or Linear Temporal Logic over Process Traces ($\mathrm{LTL_p}$) [17].

2.2 Linear Temporal Logic over Finite Traces

This section recaps minimal notions of Linear Temporal Logic over Finite Traces ($\mathrm{LTL_f}$) [24]. We start by introducing its syntax and semantics, and then we informally describe its temporal operators, and some Process Mining application-specific notation.

Syntax Let $\mathcal{A}$ be a finite set of propositional symbols. A *finite trace* is a sequence $\pi = \pi_0 \cdots \pi_{n-1}$, with $n \in \mathbb{N}$. For each i, $\pi_i \subseteq \mathcal{A}$ is its i-th *state*, and $|\pi| = n$ denotes the trace *length*. $\mathrm{LTL_f}$ is an extension of propositional logic that can be used to reason about temporal properties of traces. It shares the same syntax as Linear Temporal Logic (LTL) [33], but it is interpreted over *finite* traces

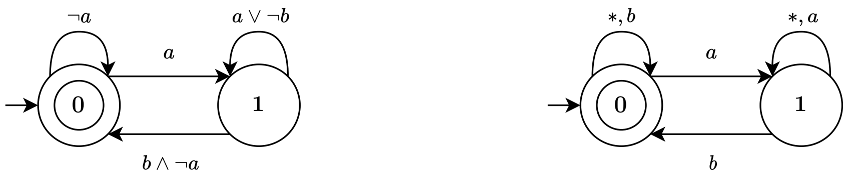

Fig. 1. Left: Minimal automaton for the $\mathrm{LTL_f}$ formula $\varphi = \mathsf{G}\ (a \rightarrow \mathsf{X}\ \mathsf{F}\ b)$. Models of propositional formulae (labeling the transitions) compactly represent sets of symbols; Right: Minimal automaton for φ interpreted as a $\mathrm{LTL_p}$ formula, where $*$ denotes any $x \in \mathcal{A} \setminus \{a, b\}$. A comma on edges denotes multiple transitions.

rather than *infinite* ones. An $\mathrm{LTL_f}$ formula φ over $\mathcal{A}$ is defined according to the following grammar:

$$\varphi ::= \top \mid a \mid \neg\varphi \mid \varphi \land \varphi \mid \mathsf{X}\ \varphi \mid \varphi_1\ \mathsf{U}\ \varphi_2,$$

where $a \in \mathcal{A}$. We assume common propositional ($\lor$, $\rightarrow$, $\longleftrightarrow$, etc.) and temporal logic shorthands. In particular, for temporal operators, we define *eventually* operator $\mathsf{F}\ \varphi \equiv \top\ \mathsf{U}\ \varphi$, the *always* operator $\mathsf{G}\ \varphi \equiv \neg\mathsf{F}\ \neg\phi$, the *weak until* operator $\varphi\ \mathsf{W}\ \varphi' \equiv \mathsf{G}\ \varphi \lor \varphi\ \mathsf{U}\ \varphi'$ and *weak next* operator $\mathsf{X_w}\ \varphi \equiv \neg\mathsf{X}\ \neg\varphi \equiv \mathsf{X}\ \varphi\ \lor \neg\mathsf{X}\ \top$.
Semantics Let φ be an $\mathrm{LTL_f}$ formula, π a finite trace, $0 \leq i < |\pi|$ an integer. The *satisfaction relation*, denoted by $\pi, i \models \varphi$, is defined recursively as follows:

- $\pi, i \models \top$;
- $\pi, i \models p$ iff $p \in \pi_i$;
- $\pi, i \models \neg\varphi$ iff $\pi, i \models \varphi$ does not hold;
- $\pi, i \models \varphi_1 \land \varphi_2$ iff $\pi, i \models \varphi_1$ and $\pi, i \models \varphi_2$;
- $\pi, i \models \mathsf{X}\ \varphi$ iff $i < |\pi| - 1$ and $\pi, i+1 \models \varphi$;
- $\pi, i \models \varphi_1\ \mathsf{U}\ \varphi_2$ iff $\exists j$ with $i \leq j \leq |\pi|$ s.t. $\pi, j \models \varphi_2$ and $\forall k$ with $i \leq k < j$, $\pi, k \models \varphi_1$.

We say that π is a *model* for φ if $\pi, 0 \models \varphi$, denoted as $\pi \models \varphi$. Each $\mathrm{LTL_f}$ formula φ can be associated to a minimal automaton $\mathcal{M}(\varphi)$ over $2^{\mathcal{A}}$ such that for whatever trace π it holds that $\pi \models \varphi$ if and only if π, interpreted as a string over $2^{\mathcal{A}}$, is accepted by $\mathcal{M}(\varphi)$ [22, 24]. A common assumption in $\mathrm{LTL_f}$ applications to Process Mining, referred to as *Declare assumption* [23] or *simplicity assumption* [11], is that exactly one activity occurs in each state. $\mathrm{LTL_f}$ with this additional restriction is known as $\mathrm{LTL_p}$ [17], and traces therein, known as *process traces*, can be seen as strings over the alphabet $\mathcal{A}$. This has the following practical implication: given a $\mathrm{LTL_p}$ formula φ, the minimal automaton $\mathcal{M}(\varphi)$ of φ can be simplified into a deterministic automaton [11], as shown in Figure 1.

2.3 Declare modeling language

Declare [4] is a declarative process modeling language that consists of a set of *templates* that express temporal properties of process execution traces. The seman-

Template	**LTL$_\mathrm{p}$**	**Description**
Choice(a,b)	$\mathsf{F}\ (a \vee b)$	a or b must be executed.
ExclusiveChoice(a,b)	$\mathsf{Choice(a,b)} \wedge \neg(\mathsf{F}\ a \wedge \mathsf{F}\ b)$	Either a or b must be executed, but not both.
RespEx(a,b)	$\mathsf{F}\ a \rightarrow \mathsf{F}\ b$	If a is executed, then b must be executed as well.
CoExistence(a,b)	$\mathsf{RespEx(a,b)} \wedge \mathsf{RespEx(b,a)}$	Either a and b are both executed, or none of them is executed.
Response(a,b)	$\mathsf{G}\ (a \rightarrow \mathsf{F}\ b)$	Every time a is executed, b must be executed afterwards.
Precedence(a,b)	$\neg b\ \ \mathsf{W}\ \ a$	b can be executed only if a has been executed before.
Alt.Response(a,b)	$\mathsf{G}\ (a \rightarrow \mathsf{X}\ (\neg a\ \mathsf{U}\ b))$	Every a must be followed by b, without any other a in between.
Alt.Precedence(a,b)	$\mathsf{Precedence(a,b)}$ $\wedge \mathsf{G}\ (b \rightarrow \mathsf{X_w}\ \mathsf{Precedence(a,b)})$	Every b must be preceded by a, without any other b inbetween.
ChainResponse(a,b)	$\mathsf{G}\ (a \rightarrow \mathsf{X}\ b)$	If a is executed then b must be executed next.
ChainPrecedence(a,b)	$\mathsf{G}\ (\mathsf{X}\ b \rightarrow a) \wedge \neg b$	Task b can be executed only immediately after a.

Table 1. Some Declare templates as LTL$_\mathrm{p}$ formulae along with their informal description, as reported in [23]. We slightly edit the definitions for ChainPrecedence(a,b) and AlternatePrecedence(a,b), to align their semantics to the informal description commonly assumed in Process Mining applications. Changes w.r.t the original source [23] are highlighted in red. The Succession (resp. AlternateSuccession, ChainSuccession) template is defined as the conjunction of (Alternate, Chain) Response and Precedence templates.

tics of each Declare template is defined in terms of an underlying LTL$_\mathrm{p}$ formula. Table 1 provides the LTL$_\mathrm{p}$ definition of some Declare templates, as reported in [23]. Declare templates can be classified into four distinct categories, each addressing different aspects of process behavior: *existence* templates, specifying the necessity or prohibition of executing a particular activity, potentially with constraints on the number of occurrences; *choice* templates, centered around the concept of execution choices as they model scenarios where there is an option regarding which activities may be executed; *relation* templates, establishing a dependency between activities as they dictate that the execution of one activity necessitates the execution of another, often under specific conditions or requirements; *negation* templates, modelling mutual exclusivity or prohibitive conditions in activity execution. In Table 1, Choice(a,b) and ExclusiveChoice(a,b) are examples of *choice* templates; while the others fall under the *relation* category. A Declare model is a set of *constraints*, where a constraint is a particular instantiation of a template, over specific activities, called respectively *activation* and *target* for binary constraints. Informally, the activation of a Declare constraint is the activity whose occurrence imposes a constraint over the occurrence of the target on the rest of the trace. A more formal account of activation-target semantics of Declare constraints can be found in [8]. The following example showcases the informal semantics of the Response template, which will serve as a running example in the rest of the paper.

Example 1 (Semantics of the Response template). The informal semantics for Response(a,b) is that *whenever a occurs in the trace, b will appear in the future.* Formally, the template is defined as $\mathsf{G}\ (a \rightarrow \mathsf{F}\ b)$. Thus, if a occurs at time t in a trace π, in order for the constraint to be satisfied, b must appear in the trace suffix $\pi_{t+1}, \ldots, \pi_n$. In the context of a customer service process, let's consider the response template instantiated with a = customer_complains and b = address_complain, corresponding to the template instantiation, i.e., the constraint, Response(customer_complains,address_complain). Such constraint imposes that when a customer complaint is received (activation activity), a follow-up action, such as addressing the complaint (target activity), must be executed. The trace π = customer_complains, logging_complain, address_complain, feedback_collection satisfies the above constraint while the trace π' = customer_complains, logging_complain, address_complain, customer_complains, feedback_collection does not, indeed the second occurrence of customer_complains is not followed by any address_complain event.

This paper focuses on the Declare *conformance checking* and Declare *(template) query checking* tasks, as defined below:

Conformance checking. Let $\mathcal{L}$ be an event log (a multiset of traces) and $\mathcal{M}$ a Declare model. The conformance checking task $(\mathcal{L}, \mathcal{M})$ consists in computing the subset of traces $\mathcal{L}' \subseteq \mathcal{L}$ such that for each $\pi \in \mathcal{L}'$, $\pi \models c$ for all $c \in \mathcal{M}$.

Query checking. Let $\mathcal{L}$ be an event log, and c a constraint. The support of c on $\mathcal{L}$, denoted by $\sigma(c, \mathcal{L})$, is defined as the fraction of traces $\pi \in \mathcal{L}$ such that $\pi \models c$. High support for a constraint is usually interpreted as a measure of relevance for the given constraint on the log $\mathcal{L}$. Given a Declare template t and a *support threshold* $s \in (0, 1]$, the query checking task $(t, \mathcal{L}, s)$ consists in computing variable-activity bindings such that the constraint c we obtain by instantiating t with such bindings has a support greater than s on $\mathcal{L}$.

Interested readers can refer to [12, 14] as a starting point for Declare.

2.4 Answer Set Programming

Answer set programming (ASP) [6, 21] is a declarative programming paradigm based on the stable models semantics, which has been used to solve many complex AI problems [15]. We now provide a brief introduction describing the basic language of ASP. We refer the interested reader to [6, 21, 19] for a more comprehensive description of ASP. The syntax of ASP follows Prolog's conventions: variable terms are strings starting with an uppercase letters; constant terms are either strings starting by lowercase letter or are enclosed in quotation marks, or are integers. An *atom* of arity n is an expression of the form $p(t_1, \ldots, t_n)$ where p is a predicate and $t_1, \ldots, t_n$ are terms. A (positive) *literal* is an atom a or its negation (negative literal) *not a* where *not* denotes negation as failure. A *rule* is an expression of the form $h \leftarrow b_1, \ldots, b_n$ where $b_1, \ldots, b_n$ is a conjunction of literals, called the *body*, $n \geq 0$, and h is an atom called the *head*. All variables in a rule must occur in some positive literal of the body. A *fact* is a rule with an empty body (i.e., $n = 0$). A *program* is a finite set of rules. Atoms, rules and programs that do not contain variables are said to be ground. The Herbrand Universe U_P is the collection of constants in the program P. The Herbrand Base B_P is the set of ground atoms that can be generated by combining predicates from P with the constants in U_P. The ground instantiation of P, denoted by $ground(P)$, is the union of ground instantiations of rules in P that are obtained by replacing variables with constants in U_P. An *interpretation* I is a subset of B_P. A positive (resp. negative) literal ℓ is true w.r.t. I, if $\ell \in I$ (resp. $\overline{\ell} \notin I$); it is false w.r.t. I if $\ell \notin I$ (resp. $\overline{\ell} \in I$). An interpretation I is a *model* of P if for each $r \in ground(P)$, the head of r is true whenever the body of r is true. Given a program P and an interpretation I, the (Gelfond-Lifschitz) reduct [21] P^I is the program obtained from $ground(P)$ by (i) removing all those rules having in the body a false negative literal w.r.t. I, and (ii) removing negative literals from the body of remaining rules. Given a program P, the model I of P is a *stable model* or *answer set* if there is no $I' \subset I$ such that I' is a model of P^I.

In the paper, also use more advanced ASP constructs such as *choice rules* and *function symbols*. We refer the reader to [7] for a description of more advanced ASP constructs. In the rest of the paper, ASP code examples will use CLINGO [20] input language.

3 Translation-based ASP encodings for Declare

This section introduces ASP encodings for conformance checking of Declare models and query checking of Declare constraints with respect to an input event log, based on the translation to automata and syntax trees. Both encodings share the same input fact schema to specify which Declare constraints belong to the model, or which constraint we are performing query checking against. These encodings are *indirect*, since they rely on a translation, but also *general* in the sense that they can be applied to the evaluation of arbitrary LTL_p formulae. This is achieved, in the case of the syntax tree encoding, by reifying the syntax tree of a formula and by explicitly modeling the semantics of each LTL_p temporal operator through a logic program, and in the case of the automaton encoding, by exploiting the well-known LTL_p-to-automaton translation [11, 24]. Thus, one can use these two encodings to represent Declare constraints by their LTL_p definitions. The automaton-based encoding is adapted from [10], the syntax tree-based encoding is adapted from [28] - integrating changes to allow for the above-mentioned shared fact schema and evaluation over multiple traces. A similar encoding has also been used in [27] to learn LTL_f formulae from sets of example traces, using the ASP-based inductive logic programming system ILASP [29]. We start by defining how event logs and Declare constraints are encoded into facts, then introduce conformance checking and query checking encodings with the two approaches.

Encoding process traces. For our purposes, an event log $\mathcal{L}$ is a multiset of process traces, thus a multiset of strings over an alphabet of propositional symbols $\mathcal{A}$ (representing activities). We assume that each trace $\pi \in \mathcal{L}$ is uniquely indexed by an integer, and we denote that the trace π has index i by $id(\pi) = i$. This is a common assumption in Process Mining, where i is referred to as the *trace identifier*. Traces are modeled through the predicate `trace/3`, where the atom $trace(i, t, a)$ encodes that $\pi_t = a, id(\pi) = i$ — that is, the t-th activity in the i-th trace π is a. Given a process trace π, we denote by $E(\pi)$ the set of facts that encodes it. Thus, an event log $\mathcal{L}$ is encoded as $E(\mathcal{L}) = \bigcup_{\pi \in \mathcal{L}} E(\pi)$.

Example 2 (Encoding a process trace). Consider an event log composed of the two process traces $\pi^0 = abc$ and $\pi^1 = xyz$, respectively with identifiers 0 and 1, over the propositional alphabet $\mathcal{A} = \{a, b, c, x, y, z\}$. This is encoded by the following set of facts:

```
trace(0,0,a). trace(0,1,b). trace(0,2,c).
trace(1,0,x). trace(1,1,y). trace(1,2,z).
```

Each Declare template, informally, can be understood as a "LTL$_\text{p}$ formula with variables". Substituting these variables with activities yields a Declare constraint. How templates are instantiated into constraints, and how constraints are evaluated over traces, depends on the ASP encoding we use. However, all encodings share a common fact schema where constraints are expressed as templates with bound variable substitutions.

Encoding Declare constraints. A Declare constraint is modeled by predicates `constraint/2` and `bind/3`. The former model which Declare template a given constraint is instantiated from and the latter which activity-variable bindings instantiate the constraint. An atom $constraint(cid, template)$ encodes that the constraint uniquely identified by cid is an instance of the template $template$. The atom $bind(cid, arg, value)$ encodes that the constraint uniquely identified by cid is obtained by binding the argument arg to the activity $value$. Given a Declare model $\mathcal{M} = \{c_1, \ldots, c_n\}$, where the subscript i uniquely indexes the constraint c_i, we denote by $E(\mathcal{M})$ the set of facts that encodes $\mathcal{M}$, that is $E(\mathcal{M}) = \bigcup_{c \in \mathcal{M}} E(c)$. Recall that in Declare $\pi \models \mathcal{M}$ if and only if $\pi \models c$ for all $c \in \mathcal{M}$, thus there is no notion of "order" among the constraints within $\mathcal{M}$ and it does not matter how indexes are assigned to constraints as long as they are unique.

Example 3 (Encoding a Declare model). Consider the model $\mathcal{M}$ composed of the two constraints Response(a_1, a_2) and Precedence(a_2, a_3). $\mathcal{M}$ is encoded by the following facts:

```
constraint(0,"Response").
bind(0,arg_0,a_1).
bind(0,arg_1,a_2).
```

```
constraint(1,"Precedence").
bind(1,arg_0,a_2).
bind(1,arg_1,a_3).
```

3.1 Encoding Conformance Checking

All the Declare conformance checking encodings we propose consist of a stratified normal logic program P_{CF}. Given a log $\mathcal{L}$ and a Declare model $\mathcal{M}$, it holds that for every trace $\pi_i \in \mathcal{L}$, $\pi_i \models c_j \in \mathcal{M}$ if and only if the unique model of $P_{CF} \cup E(\mathcal{M}) \cup E(\mathcal{L})$ contains the atom $sat(i, j)$. Complete encodings for all the templates in Table 1 are available online.

Automaton encoding. The automaton encoding, reported in Figure 2, models Declare templates through their corresponding automaton obtained by translating the template's LTL_p definition [22]. The automaton's complete transition function is reified into a set of facts that defines the template in ASP. The predicates `initial/2, accepting/2` model the initial and accepting states of the automaton, while `template/4` stores the transition function of the template-specific automaton. In particular, `arg_0` refers to the template *activation*, and `arg_1` refers to the template *target*. A constraint c instantiated from a template binds its `arg_0`, `arg_1` to specific activities. The constant `"*"` is used as a placeholder for any activity in $\mathcal{A} \backslash \{x, y\}$ – where x and y are the bindings of `arg_0` and `arg_1`. Activities not explicitly mentioned as within the atomic propositions in an LTL_p formula φ have the same influence to $\pi \models \varphi$. Consequently, all unbound activities can be denoted by the symbol `"*"` in the automaton transition table. As an example, consider the constraint $c =$ Response(a,b), shown in Figure 4. Evaluating the trace `abwqw` is equivalent to evaluating the trace `abtts`, which would be equivalent to evaluating the trace `ab***`, since a and b are the only propositional formulae that appear in the definition of Response(a,b).

Syntax tree-based encoding. The syntax tree encoding, shown in Figure 3, reifies the syntax tree of a LTL_p formula into a set of facts, where each node represents a sub-formula. The semantics of temporal operators and propositional operators is defined in terms of ASP rules. Analogously to the automaton encoding, templates are defined in terms of reified syntax trees, which are used to evaluate each constraint according to the template they are instantiated from. The following normal rules define the semantics of each temporal and propositional operator. We report the rules for operators $\{$ U , X $, \neg, \wedge\}$ which are the basic operators of LTL_p. The full encoding, that also includes definitions of derived operators, is available online. In particular, the `true/4` predicate tracks which sub-formula of a constraints' definition is true at any given time. As an example, the atom $true(c, f, t, i)$ encodes that *at time t the constraint sub-formula f of constraint c is satisfied on the i-th trace.* The predicates `conjunction/3`, `negate/2`, `next/2`, `until/3` model the topology of the syntax tree of the corresponding formula. The first term refers to a node identifier, while the other terms (one for unary operators , two for the binary operators are the node identifiers of its child nodes. The `atom/2` predicate models that a given node (first term) is an atom, bound to a particular argument (second term) by the `bind/3` predicate which is used in encoding of Declare constraints. Figure 5 shows an example.

```
% Automaton initial state
cur_state(C,TID,S,0) :-
  trace(TID,_,_),
  initial(Template,S),
  constraint(C,Template).

% Last point of each trace
last(TID,T) :-
  trace(TID,T,_),
  not trace(TID,T+1,_).

% A trace is accepted
sat(TID,C) :-
  cur_state(C,TID,S,T+1),
  last(TID,T),
  template(Template,C),
  constraint(C, Template),
  accepting(Template,S).
```

```
% Reads activation/target
cur_state(C,TID,S2,T+1) :-
  cur_state(C,TID,S1,T),
  constraint(C,Template),
  template(Template,S1,Arg,S2),
  trace(TID,T,A),
  bind(C,Arg,A).



% Reads "*"
cur_state(C,TID,S2,T+1) :-
  cur_state(C,TID,S1,T),
  constraint(C,Template),
  template(Template,S1,"*",S2),
  trace(TID,T,A),
  not bind(C,_,A).
```

Fig. 2. ASP program to execute a finite state machine corresponding to a constraint, encoded as `template/4` facts, on input strings encoded by `trace/3` facts.

3.2 Encoding Query Checking

The query checking problem takes as input a Declare template $\mathcal{T}$, an event log $\mathcal{L}$ and consists in deciding which constraints c can be instantiated from $\mathcal{T}$ such that $\sigma(c, \mathcal{L}) \geq k$, where $\sigma(c, \mathcal{L})$ is the support and denotes the fraction of traces in $\mathcal{L}$ that are models of c. The problem has been formally introduced in [9] for temporal logic formulae, and in [34] it has been framed into a Process Mining setting, in the context of LTL_f. An ASP-based solution to the problem has been provided in [10], through the same automaton encoding we have been referring to throughout the paper, and instead an exhaustive search-based, Declare-specific implementation is provided in the Declare4Py [13] library. From the ASP perspective, a conformance checking encoding can be easily adapted to perform query checking, by searching over possible variable-activities bindings that yield a constraint above the chosen support threshold. In particular, we adapt the query checking encoding presented in [18] to the LTL_f setting. In order to encode the query checking problem, we slightly change our input model representation, as reported in the following example.

Example 4. Consider the query checking problem instance over the template *Response*, with both its activation and target ranging over $\mathcal{A}$. The `var_bind/3` predicate, analogously to `bind_3`, models that in a given template a parameter is bound to a variable. Notice that the ASP formulation can be easily generalized to query check sets of Declare constraints, while available tools address only a single constraint at a time. For the query checking problem, we are interested in

```
last(TID,T) :-
  trace(TID,T,_),
  not trace(TID,T+1,_).

true(C,F,T,TID) :-
  constraint(C,Template),
  template(Template,
    atom(F,Arg)
  ),
  bind(C,Arg,A),
  trace(TID,T,A).

true(C,F,T,TID) :-
  constraint(C,Template),
  template(Template,
    conjunction(F,G,H)
  ),
  trace(TID,T,_),
  true(C,G,T,TID),
  true(C,H,T,TID).

true(C,F,T,TID) :-
  constraint(C,Template),
  template(Template,negate(F,G)),
  not true(C,G,T,TID),
  trace(TID,T,_).
```

```
sat(C,TID) :-
  true(C,0,0,TID).

true(C,F,Ti,TID) :-
  constraint(C,Template),
  template(Template,next(F,G)),
  trace(TID,Ti,_),
  Tj=Ti+1,
  Ti<M,
  last(TID,M),
  true(C,G,Tj,TID).

true(C,F,Ti,TID) :-
  constraint(C,Template),
  template(Template,
      until(F,G,H)),
  trace(TID,Ti,_),
  trace(TID,Tj,_),
  Tj>=Ti,
  Tj<=M,
  last(TID,M),
  X {true(C,G,T,TID):
      trace(TID,T,_), T >=Ti,
      T<Tj} X,
  X = Tj-Ti,
  true(C,H,Tj,TID).
```

Fig. 3. ASP program to evaluate each sub-formula of the LTL$_\text{p}$ definition of a given template, encoded as `template/2` facts, on input strings encoded by a syntax tree representation through the `conjunction/3`, `negate/2`, `until/3`, `next/2` and `atom/2`.

```
template("Response",0,"*",0).
template("Response",0,arg_1,0).
template("Response",0,arg_0,1).
template("Response",1,arg_1,0).
template("Response",1,"*",1).
template("Response",1,arg_0,1).
accepting("Response",0).
initial("Response",0).
```

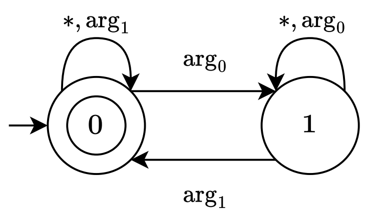

Fig. 4. Left: Facts that encode the Response template; Right: A minimal finite state machine whose recognized language is equal to the set of models of Response, under LTL$_\text{p}$ semantics.

```
template("Response",always(0,1)).
template("Response",implies(1,2,3)).
template("Response",atom(2,arg_0)).
template("Response",eventually(3,4)).
template("Response",atom(4,arg_1)).
```

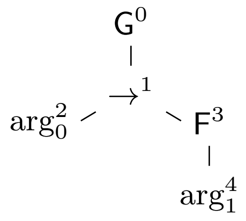

Fig. 5. Left: Facts that encode the Response template; Right: Syntax tree of the Response template $\mathrm{LTL_p}$ definition.

tuples of activities that, when substituted to the constraints' variables, yield a constraint whose support is above the threshold over the input log. The `domain/2` predicate can be used to give each variable its own subset of possible values, but in this case, for both variables, the domain of admissible substitutions spans over $\mathcal{A}$. The choice rule generates candidate substitutions that are pruned by the constraints if they are above the maximum number of violations. Given an input support threshold $s \in (0, 1]$, the constant `max_violations` is set to the nearest integer above $(1 - s) \cdot |\mathcal{L}|$.

```
constraint(c,"Response").
var_bind(c,arg_0,var(a)).
var_bind(c,arg_1,var(b)).
domain(var(a),A) :- trace(_,_,A).
domain(var(b),A) :- trace(_,_,A).
{ bind(C,Arg,Value): domain(Var,Value) } = 1 :- var_bind(C,Arg,Var).
:- #count{X: not sat(C, X), trace(X, _, _)} > max_violations.
```

4 Direct ASP encoding for Declare

The previous encodings are general techniques that enable reasoning over arbitrary $\mathrm{LTL_p}$ formulae. The encoding discussed in this section instead is an ad-hoc, direct translation of the semantics of Declare constraints into ASP rules. The general approach we followed in defining the templates, is to model *constraint failures* through a `fail/2` predicate. Due to the activation-target semantics of Declare templates, sometimes it is required to assert that an activation condition is matched in the suffix of the trace by a correlation condition. In the encoding, this is modeled by the `witness/3` predicate. This mirrors the *activation* and *target* concepts in the definition of Declare constraints. However, the encodings are not based on a systematic, algorithmic rewriting. We show an example using the Response and Precedence templates, typical patterns in the verification literature [14].

Example 5 (Modeling the Response template directly in ASP). Recall from Table 1 that the template Response(a, b) is defined as the $\mathrm{LTL_p}$ formula $\mathsf{G}\ (a \rightarrow$

F b), whose informal meaning is that *whenever a happens, b must happen somewhere in the future.* Thus, every time we observe an a at time t, in order for $Response(a, b)$ to be true, we have to observe b at a time instant $t' \geq t$. The first rule below encodes this situation. If we observe at least one a that is not matched by any b in the future, the constraint fails, which is encoded in the second rule.

```
witness(C,T,TID) :-
  constraint(C, "Response"),
  bind(C,arg_0,X),
  bind(C,arg_1,Y),
  trace(TID,T,X),
  trace(TID,T',Y), T'>=T.
```

```
fail(C,TID) :-
  constraint(C,"Response"),
  bind(C,arg_0,X),
  bind(C,arg_1,Y),
  trace(TID,T,X),
  not witness(C,T,TID).
```

Example 6 (Modeling the Precedence template directly in ASP). Recall from Table 1 that the constraint Precedence(x, y) is defined as the $\mathrm{LTL_p}$ formula $\neg x$ W y = G $(\neg x)$ $\vee$ $\neg x$ U y, whose informal meaning is that *if y happens, x must have happened before.* Notice that in order to witness the failure of this constraint, it is enough to reason about the trace prefix up to the first occurrence of y.

```
fail(C,TID) :-
  constraint(C,"Precedence"),
  bind(C,arg_0,X),
  bind(C,arg_1,Y),
  trace(TID,T',Y),
  T = #min{Q: trace(TID,Q,X)},
  trace(TID,T,X), T'<=T.
```

```
fail(C,TID) :-
  constraint(C,"Precedence"),
  bind(C,arg_0,X),
  bind(C,arg_1,Y),
  trace(TID,_,Y),
  not trace(TID,_,X).
```

Let P_{CF} denote the logic program that encodes $\mathrm{LTL_p}$ semantics or Declare semantics. The logic program $P \cup E(\mathcal{M}) \cup E(\mathcal{L})$ has a unique model, and contains the atom $sat(c, i)$ if and only if $\pi_i \models c$. We validate our direct encoding definitions of Declare semantics by a bounded model checking approach, searching for a counterexample trace π that is accepted (rejected) by the direct encoding but rejected (accepted) by the automaton of the corresponding constraint, over the propositional alphabet $\{a, b, *\}$ which represents respectively the first two parameters of the Declare constraint under test, and a placeholder "everything else" character, as discussed in the automata encoding subsection.

5 Experiments

In this section, we report the results of our experiments comparing different methods to perform conformance checking and query checking of Declare models, using the ASP-based representations outlined in the previous sections and Declare4Py, a Python library for Declare tasks. Methods will be referred as $\mathrm{ASP}_{\mathcal{D}}$,

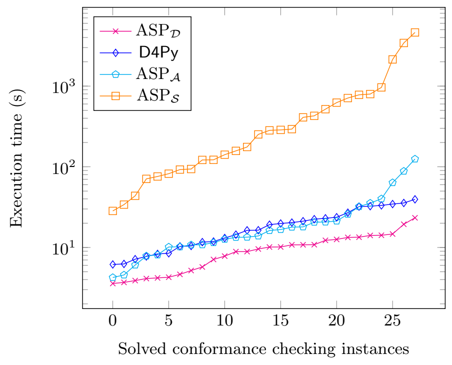

Fig. 6. Conformance checking cactus.

Log	ASP$_\mathcal{D}$	D4Py	ASP$_\mathcal{A}$	ASP$_\mathcal{S}$
ID	**23.3**	39.5	124.9	4621.7
RP	**10.8**	16.3	25.8	409.8
PT	**5.2**	8.5	12.6	121.6
SC	**4.3**	11.6	13.4	141.2
PL	**10.8**	35.6	20.7	624.1
DD	**14.2**	22.4	40.1	963.2
BC	**14.1**	23.7	20.5	796.6

Table 3. Run time in seconds to perform conformance checking on $\mathcal{C}^{\mathrm{IV}}$, the model that contains the most constraints, on each log.

ASP$_\mathcal{A}$, ASP$_\mathcal{S}$ and D4Py - denoting respectively our direct encoding, the automata and syntax tree-based translation methods and Declare4Py. We start by describing datasets (logs and Declare models), and execution environment to conclude by discussing experimental results.

Data. We validate our approach on real-life event logs from past BPI Challenges [31]. These event logs are well-known and actively used in Process Mining literature. For each event log $\mathcal{L}_i$, we use Declare4Py to mine the set of Declare constraints $\mathcal{C}_i$ whose support on $\mathcal{L}_i$ is above 50%. Then, we define four models, $\mathcal{C}_i^{\mathrm{I}}, \mathcal{C}_i^{\mathrm{II}}, \mathcal{C}_i^{\mathrm{III}}, \mathcal{C}_i^{\mathrm{IV}}$, containing respectively the first 25%, 50%, 75% and 100% of the constraints in a random shuffling of $\mathcal{C}_i$, such that $\mathcal{C}_i^{\mathrm{I}} \subset \mathcal{C}_i^{\mathrm{II}} \subset \mathcal{C}_i^{\mathrm{III}} \subset \mathcal{C}_i^{\mathrm{IV}}$. Table 2 summarizes some statistics about the logs and the Declare models we mined over the logs. All resource measurements take into account the fact that ASP encodings require an additional translation step from the XML-based format of event

Log name	$\lvert\mathcal{A}\rvert$	Average $\lvert\pi\rvert$	$\lvert\mathcal{L}\rvert$	$\lvert\mathcal{C}^{\mathrm{IV}}\rvert$
Sepsis Cases (SC)	16	14.5	1050	76
Permit Log (PL)	51	12.3	7065	26
BPI Challenge 2012 (BC)	23	12.6	13087	10
Prepaid Travel Cost (PC)	29	8.7	2099	52
Request For Payment (RP)	19	5.4	6886	52
International Declarations (ID)	34	11.2	6449	152
Domestic Declarations (DD)	17	5.4	10500	52

Table 2. Statistics for the logs used in the experiments: $\lvert\mathcal{A}\rvert$ denotes the number of activities for the log; Average $\lvert\pi\rvert$ is the average trace length for the log; $\lvert\mathcal{L}\rvert$ is the number of traces in the log; $\lvert\mathcal{C}^{\mathrm{IV}}\rvert$ is the total number of Declare constraints above 50% support in the log, by Declare4Py miner.

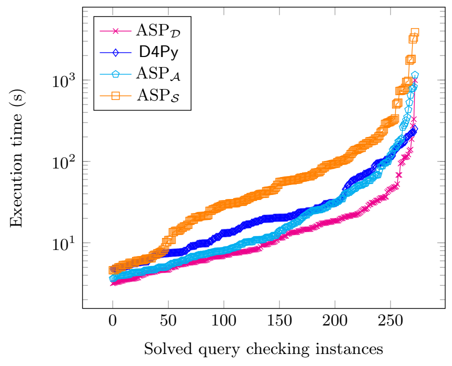

Fig. 7. Query checking cactus plot.

Log	ASP$_{\mathcal{D}}$	D4Py	ASP$_{\mathcal{A}}$	ASP$_{\mathcal{S}}$
ID	**817.2**	1624.5	1654.0	3522.4
RP	884.2	565.8	**318.2**	1179.4
PT	**223.6**	451.1	236.1	427.9
SC	**163.8**	267.0	173.1	665.1
PL	**1614.0**	4227.7	3926.8	5397.5
DD	**407.7**	698.2	479.2	2436.2
BC	**2304.8**	2467.7	6636.0	27445.3

Table 4. Cumulative run time in seconds to perform all query checking tasks on a given log.

logs to a set of facts. The translation time is included in the measurement times and is comparable with the time taken by Declare4Py.

Execution environment. The experiments in this section 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, Declare4Py 1.0 and pyrunlim [5] to measure resources usage. Experiments were run sequentially. All data and scripts to reproduce our experiments are available at this repository.

Conformance Checking. We consider the conformance checking tasks $(\mathcal{L}_i, \mathcal{M})$, with $\mathcal{M} \in \{\mathcal{C}_i^{\mathrm{I}}, \mathcal{C}_i^{\mathrm{II}}, \mathcal{C}_i^{\mathrm{III}}, \mathcal{C}_i^{\mathrm{IV}}\}$, over the considered logs and its Declare models. Figure 4 reports the solving times for each method in a cactus plot. Recall that a point (x, y) in a cactus plot represents the fact that a given method solves the x-th instance, ordered by increasing execution times, in y seconds. Table 3 reports the same data aggregated by the event log dimension, best run-time in bold. Overall, our direct encoding approach is faster than the other ASP-based encodings as well as Declare4Py on considered tasks. ASP$_{\mathcal{A}}$ and Declare4Py perform similarly, whereas ASP$_{\mathcal{S}}$ is less efficient.

Query Checking. We consider the query checking instances $(t, \mathcal{L}_i, s)$ where t is a Declare template, from the ones defined in Table 1, $s \in \{0.50, 0.75, 1.00\}$ is a support threshold, and $\mathcal{L}_i$ is a log. Figure 5 summarizes the results in a cactus plot, and Table 4 aggregates the same data on the log dimension, best runtime in bold. ASP$_{\mathcal{D}}$ is again the best method overall, outperforming other ASP-based methods with the exception of ASP$_{\mathcal{A}}$ on the RP log tasks. Again, ASP$_{\mathcal{A}}$ and D4Py perform similarly and ASP$_{\mathcal{S}}$ is the worst.

Discussion. We conjecture that the significant increase in maximum memory usage, as indicated in Table 5, is the primary factor contributing to the performance gap observed for ASP$_{\mathcal{S}}$ in both tasks. In fact, we observe that in conformance checking ASP$_{\mathcal{D}}$ is more efficient w.r.t. memory consumption when

Log	**Conformance Checking**				**Query Checking**			
	$\mathrm{ASP}_{\mathcal{D}}$	D4Py	$\mathrm{ASP}_{\mathcal{A}}$	$\mathrm{ASP}_{\mathcal{S}}$	$\mathrm{ASP}_{\mathcal{D}}$	D4Py	$\mathrm{ASP}_{\mathcal{A}}$	$\mathrm{ASP}_{\mathcal{S}}$
BC	**323.6**	566.3	546.0	8757.9	3157.0	**580.7**	1450.8	18866.5
DD	536.6	**386.0**	927.2	14473.8	728.1	**336.4**	513.0	2706.0
ID	837.1	**578.4**	2338.9	55435.2	1498.5	**583.5**	763.6	5029.8
PL	**312.8**	2062.2	579.5	11388.3	2434.2	2071.0	**1023.3**	7006.2
PC	**222.2**	281.9	341.2	5211.2	435.3	283.4	**282.8**	1209.3
RP	372.3	**347.6**	610.8	9706.0	574.8	**312.3**	396.9	1843.3
SC	**195.0**	245.6	336.4	5786.5	480.4	**244.6**	247.2	2146.7

Table 5. Max memory usage (MB) over all the conformance checking and query checking tasks, aggregated by log, for all the considered methods. Lowest value in boldface.

compared to D4Py, and, when combined, these two methods collectively show better memory efficiency when compared to other ASP-based methods. As for the query checking tasks, D4Py is the most efficient method memory-wise. This is expected, since the imperative nature of its implementation allows to *"iterate and discard"* candidate assignments, rather than requiring their explicit grounding, as in the ASP-based techniques. In conclusion, it is worth noticing that for both tasks, $\mathrm{ASP}_{\mathcal{S}}$ is the least efficient memory-wise implementation, and it also tends to exhibit lower efficiency in terms of running times across nearly all logs.

6 Conclusion

Declare is a declarative process modeling language, which describes processes by sets of temporal constraints. Declare specifications can be expressed as $\mathrm{LTL}_{\mathrm{p}}$ formulae, and traditionally have been evaluated by executing the equivalent automata [12]. Translation-based approaches (on automata, or syntax trees) are at the foundation of existing ASP-based solutions [10, 28]. This paper proposes a novel direct encoding of Declare in ASP that is not based on translations. Moreover, for the first time, we put on common ground (regarding input fact schema) and compare available ASP solutions for conformance checking and query checking. Our experimental evaluation over well-known event logs provides the first aggregate picture of the performance of the methods considered. The results show that our direct encoding outperforms other methods in terms of execution time, and thus that ASP provides a compact, declarative and efficient way to implement Declare constraints in the considered tasks. As far as future work is concerned, we plan to extend our approach to the *data perspective* [30], i.e., attaching data payloads to each activity in a trace, as well as considering other Process Mining tasks such as log generation tasks, along the lines of [10].

References

[1] *A Guide To The Project Management Body Of Knowledge (PMBOK Guides)*. Project Management Institute, 2004. ISBN: 193069945X.

[2] Wil M. P. van der Aalst. "Process Mining: A 360 Degree Overview". In: *PM Handbook*. Vol. 448. LNIP. Springer, 2022, pp. 3–34.

[3] Wil M. P. van der Aalst. "The Application of Petri Nets to Workflow Management". In: *J. Circuits Syst. Comput.* 8.1 (1998), pp. 21–66.

[4] Wil M. P. van der Aalst, Maja Pesic, and Helen Schonenberg. "Declarative workflows: Balancing between flexibility and support". In: *Comput. Sci. Res. Dev.* 23.2 (2009), pp. 99–113.

[5] Mario Alviano. *The pyrunlim tool.* 2014. URL: `https://github.com/alviano/python/tree/master/pyrunlim`.

[6] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. "Answer set programming at a glance". In: *Commun. ACM* 54.12 (2011), pp. 92–103.

[7] Francesco Calimeri et al. "ASP-Core-2 Input Language Format". In: *Theory Pract. Log. Program.* 20.2 (2020), pp. 294–309.

[8] Alessio Cecconi et al. "Measuring the interestingness of temporal logic behavioral specifications in process mining". In: *Inf. Syst.* 107 (2022), p. 101920. URL: `https://doi.org/10.1016/j.is.2021.101920`.

[9] William Chan. "Temporal-logic Queries". In: *Computer Aided Verification.* Ed. by E. Allen Emerson and Aravinda Prasad Sistla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 450–463. ISBN: 978-3-540-45047-4.

[10] Francesco Chiariello, Fabrizio Maria Maggi, and Fabio Patrizi. "ASP-Based Declarative Process Mining". In: *AAAI.* AAAI Press, 2022, pp. 5539–5547.

[11] Francesco Chiariello, Fabrizio Maria Maggi, and Fabio Patrizi. "From LTL on Process Traces to Finite-state Automata". In: *BPM.* Vol. 3469. CEUR WP. CEUR-WS.org, 2023, pp. 127–131.

[12] Claudio Di Ciccio and Marco Montali. "Declarative Process Specifications: Reasoning, Discovery, Monitoring". In: *PM Handbook.* Vol. 448. LNIP. Springer, 2022, pp. 108–152.

[13] Ivan Donadello et al. "Declare4Py: A Python Library for Declarative Process Mining". In: *BPM (PhD/Demos).* Vol. 3216. CEUR WP. CEUR-WS.org, 2022, pp. 117–121.

[14] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. "Patterns in Property Specifications for Finite-State Verification". In: *ICSE.* ACM, 1999, pp. 411–420.

[15] Esra Erdem, Michael Gelfond, and Nicola Leone. "Applications of Answer Set Programming". In: *AI Mag.* 37.3 (2016), pp. 53–68.

[16] Bernd Finkbeiner and Henny Sipma. "Checking Finite Traces Using Alternating Automata". In: *Form.Meth.Syst.Des.* 24.2 (2004), pp. 101–127.

[17] Valeria Fionda and Gianluigi Greco. "LTL on Finite and Process Traces: Complexity Results and a Practical Reasoner". In: *J. Artif. Intell. Res.* 63 (2018), pp. 557–623.

[18] Valeria Fionda, Antonio Ielo, and Francesco Ricca. "Logic-based Composition of Business Process Models". In: *Proceedings of the 20th Interna-*

tional Conference on Principles of Knowledge Representation and Reasoning, KR 2023, Rhodes, Greece, September 2-8, 2023. Ed. by Pierre Marquis, Tran Cao Son, and Gabriele Kern-Isberner. 2023, pp. 272–281. DOI: 10.24963/KR.2023/27. URL: https://doi.org/10.24963/kr.2023/27.

[19] Martin Gebser et al. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[20] Martin Gebser et al. "Multi-shot ASP solving with clingo". In: *Theory Pract. Log. Program.* 19.1 (2019), pp. 27–82.

[21] Michael Gelfond and Vladimir Lifschitz. "Classical Negation in Logic Programs and Disjunctive Databases". In: *New Gener. Comput.* 9.3/4 (1991), pp. 365–386.

[22] Giuseppe De Giacomo and Marco Favorito. "Compositional Approach to Translate LTLf/LDLf into Deterministic Finite Automata". In: *ICAPS*. AAAI Press, 2021, pp. 122–130.

[23] Giuseppe De Giacomo, Riccardo De Masellis, and Marco Montali. "Reasoning on LTL on Finite Traces: Insensitivity to Infiniteness". In: *AAAI*. AAAI Press, 2014, pp. 1027–1033.

[24] Giuseppe De Giacomo and Moshe Y. Vardi. "Linear Temporal Logic and Linear Dynamic Logic on Finite Traces". In: *IJCAI*. IJCAI/AAAI, 2013, pp. 854–860.

[25] Ben Greenman et al. "Little Tricky Logic: Misconceptions in the Understanding of LTL". In: *Art Sci. Eng. Program.* 7.2 (2023).

[26] Antonio Ielo, Francesco Ricca, and Luigi Pontieri. "Declarative Mining of Business Processes via ASP". In: *PMAI@IJCAI*. Vol. 3310. CEUR WP. CEUR-WS.org, 2022, pp. 105–108.

[27] Antonio Ielo et al. "Towards ILP-based LTLf passive learning". In: *ILP*. 2023, (To Appear).

[28] Isabelle Kuhlmann, Carl Corea, and John Grant. "An ASP-Based Framework for Solving Problems Related to Declarative Process Specifications". In: *NMR*. Vol. 3464. CEUR WP. CEUR-WS.org, 2023, pp. 129–132.

[29] Mark Law. "Conflict-Driven Inductive Logic Programming". In: *Theory Pract. Log. Program.* 23.2 (2023), pp. 387–414. DOI: 10.1017/S1471068422000011. URL: https://doi.org/10.1017/S1471068422000011.

[30] Massimiliano de Leoni and Wil M. P. van der Aalst. "Data-aware process mining: discovering decisions in processes using alignments". In: *SAC*. ACM, 2013, pp. 1454–1461.

[31] Iezalde F. Lopes and Diogo R. Ferreira. "A Survey of Process Mining Competitions: The BPI Challenges 2011-2018". In: *Business Process Management Workshops*. Vol. 362. Lecture Notes in Business Information Processing. Springer, 2019, pp. 263–274.

[32] Ilkka Niemelä. "Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm". In: *Ann. Math. Artif. Intell.* 25.3-4 (1999), pp. 241–273.

[33] Amir Pnueli. "The Temporal Logic of Programs". In: *FOCS*. IEEE Computer Society, 1977, pp. 46–57.

[34] Margus Räim et al. "Log-Based Understanding of Business Processes through Temporal Logic Query Checking". In: *OTM Conferences*. Vol. 8841. Lecture Notes in Computer Science. Springer, 2014, pp. 75–92.

[35] Tran Cao Son et al. "Domain-dependent knowledge in answer set planning". In: *ACM Trans. Comput. Log.* 7.4 (2006), pp. 613–657. DOI: 10.1145/1183278.1183279. URL: https://doi.org/10.1145/1183278.1183279.

[36] Mathias Weske. *Business Process Management - Concepts, Languages, Architectures, Third Edition*. Springer, 2019.

[37] Stephen A White. "Introduction to BPMN". In: *IBM* 2.0 (2004).