TPLP **25** (1): 92–131, 2025.
doi:10.1017/S1471068424000486

Direct Encoding of Declare Constraints in ASP[‡]

FRANCESCO CHIARIELLO
IRIT, ANITI, University of Toulouse, France,
(*e-mail:* francesco.chiariello@irit.fr)

VALERIA FIONDA, ANTONIO IELO and FRANCESCO RICCA
Università della Calabria, Rende, Italia,
(*e-mails:* valeria.fionda@unical.it, antonio.ielo@unical.it, francesco.ricca@unical.it)

submitted 11 March 2024; revised 16 December 2024; accepted 18 December 2024

Abstract

Answer set programming (ASP), a well-known declarative logic programming paradigm, has recently found practical application in Process Mining. In particular, ASP has been used to model tasks involving declarative specifications of business processes. In this area, Declare stands out as the most widely adopted declarative process modeling language, offering a means to model processes through sets of constraints valid traces must satisfy, that can be expressed in linear temporal logic over finite traces (LTL$_f$). Existing ASP-based solutions encode Declare constraints by modeling the corresponding LTL$_f$ formula or its equivalent automaton which can be obtained using established techniques. In this paper, we introduce a novel encoding for Declare constraints that *directly* models their semantics as ASP rules, eliminating the need for intermediate representations. We assess the effectiveness of this novel approach on two Process Mining tasks by comparing it with alternative ASP encodings and a Python library for Declare.

Keywords: answer set programming, process mining

1 Introduction

In the context of Process Mining, a *process* typically refers to a sequence of events or activities that are performed in order to accomplish a particular goal within a business or organizational setting (Van der Aalst and Weijters, 2004). Process Mining (van der Aalst, 2022) is an interdisciplinary field offering techniques and tools to discover, monitor, and improve real processes by extracting knowledge from event logs readily available in today's information systems. One of the main tasks of Process Mining is *conformance checking*, assessing the correctness of a specific execution of a process, known as *trace*, against a *process model*. Such a process model is a formal mathematical representation

[‡] Extended version of a distinguished paper of PADL'24 (Chiariello et al., 2024) invited for rapid publication on TPLP.

enabling various analytical and reasoning tasks related to the underlying process. Process models adopt either an imperative or a declarative language form, with the former explicitly describing all possible process executions, and the latter using logic-based constraints to define what is not permitted within process executions. Imperative process models are suitable for well-structured routine processes but often fall short in scenarios involving complex coordination patterns. Declarative models, conversely, offer a flexible alternative in such scenarios.

Linear temporal logic over finite traces ($\mathrm{LTL_f}$) (De Giacomo and Vardi, 2013) has emerged has a natural choice for declarative process modeling. However, within real-world Process Mining scenarios, $\mathrm{LTL_f}$ formulae that are unrestricted in structure are rarely used. Rather, a specific collection of predefined patterns, which have their origins in the domain of system verification (Dwyer et al., 1999), is commonly used. Limiting declarative modeling languages to a set of predefined patterns has two advantages: first, it simplifies modeling tasks (Greenman et al., 2023, 2024), and second, it allows for the development of specialized solutions that may outperform generic $\mathrm{LTL_f}$ reasoners in terms of efficiency. Specifically, Declare (van der Aalst et al., 2009) is the declarative process modeling language most commonly used in Process Mining applications and consists of a set of patterns, referred to as *templates*. The semantics of Declare templates is provided in terms of $\mathrm{LTL_f}$, thus enabling logical reasoning and deduction processes within the Declare framework (Di Ciccio and Montali, 2022).

Recent research proposals suggest that answer set programming (ASP) (Gelfond and Lifschitz, 1991) can be successfully used for various tasks within declarative Process Mining (Ielo et al., 2022; Chiariello et al., 2022). Nevertheless, to the best of our knowledge, there has been no prior attempt to encode the Declare $\mathrm{LTL_f}$ patterns library using ASP in a *direct* way. Here, "direct" means an encoding that captures the semantics of Declare constraints without the need for any intermediary translation. This paper fills this gap by introducing a direct encoding approach for the main Declare patterns. The proposed technique is benchmarked against existing ASP-based encodings across various logs commonly used in Process Mining (Lopes and Ferreira, 2019). The goal of the experimental analysis is twofold: first, to assess the performance of ASP-based methods in tasks related to conformance and query checking; and second, to evaluate the effectiveness of our proposed direct encoding strategy. To facilitate reproducibility, code and data used in our experiments are openly accessible at `https://github.com/ainnoot/padl-2024`.

Related work. ASP (Gelfond and Lifschitz, 1991; Niemelä, 1999; Brewka et al., 2011) has shown promise in planning to inject domain-dependent knowledge rules, resembling predefined patterns, encoded via an action theory language into ASP (Son et al., 2006). Researchers additionally explored injecting temporal knowledge, expressed as LTL formulae, in answer set planning (Son et al., 2006). An extension of ASP with temporal operators was proposed by (Cabalar et al., 2019); and other applications of logic programming to solve Process Mining tasks have been developed in earlier works (Lamma et al., 2007; Chesani et al., 2009). The works in which ASP has been used to tackle various computational tasks in Declare-based Process Mining (Ielo et al., 2022) are closer to this paper. In (Chiariello et al., 2022) authors propose a solution based on the well-known LTL_f-to-automata translation (De Giacomo and Vardi, 2013). In (Kuhlmann et al., 2023), the authors suggested a method for encoding the semantics of temporal operators into a logic program, enabling the encoding of arbitrary LTL_f formulae, by a reification of their syntax tree. Some other recent studies (Chesani et al., 2022, 2023) tackle a slightly different objective by using ASP to directly encode Declare constraints but with the focus of distinguishing between normal and deviant process traces.

Paper structure. The paper is organized as follows. Section 2 provides preliminaries on Process Mining, LTL_f and ASP; Section 3 adapts automata-based (Chiariello et al., 2022) and syntax tree-based (Kuhlmann et al., 2023) ASP encodings to Declare; Section 4 introduces our novel direct ASP encoding for Declare; Section 5 reports the results of the experimental evaluation. Section 6 concludes the paper.

2 Preliminaries

In this section fundamental concepts related to Process Mining, LTL_f, the Declare process modeling language, and ASP are discussed.

2.1 Process mining

Process Mining (van der Aalst, 2022) 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 (Weske, 2019), 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 modeling, with Petri nets (van der Aalst, 1998) and BPMN (Chinosi and Trombetta, 2012) 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 modeling (Di Ciccio and Montali, 2022) 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 expressed in Declare (van der Aalst et al., 2009), LTL_f (Finkbeiner and Sipma, 2004), or LTL over process traces (LTL_p) (Fionda and Greco, 2018).

2.2 Linear temporal logic over finite traces

This section recaps minimal notions of LTL$_f$ (De Giacomo and Vardi, 2013). We introduce finite traces, the logic's syntax and semantics, then informally describe its temporal operators, and some Process Mining application-specific notation.

Finite traces. Let $\mathscr{A}$ be a set of propositional symbols. A *finite trace* is a sequence $\pi = \pi_0 \cdots \pi_{n-1}$ of n propositional interpretations $\pi_i \subseteq \mathscr{A}$, $i = 0, \ldots, n-1$, for some $n \in \mathbb{N}$. The interpretation π_i is also called a *state*, and $|\pi| = n$ denotes the trace *length*.

Syntax. 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 LTL (Pnueli, 1977), but it is interpreted over *finite* traces rather than *infinite* ones. An LTL$_f$ formula φ over $\mathscr{A}$ is defined according to the following grammar:

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

where $a \in \mathscr{A}$. We assume common propositional ($\vee$, $\rightarrow$, $\longleftrightarrow$, etc.) and temporal logic shorthands. In particular, for temporal operators, we define the *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 $\varphi\mathsf{W}\varphi' \equiv \mathsf{G}\varphi \vee \varphi\mathsf{U}\varphi'$ and *weak next* operator $\mathsf{X}_w\varphi \equiv \neg\mathsf{X}\neg\varphi \equiv \mathsf{X}\varphi \ \vee \neg\mathsf{X}\top$.

Semantics. Let φ be an 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:

- $\pi, i \models \top$;
- $\pi, i \models a$ iff $a \in \pi_i$;
- $\pi, i \models \neg\varphi$ iff $\pi, i \models \varphi$ does not hold;
- $\pi, i \models \varphi_1 \wedge \varphi_2$ 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 $i \leq j \leq |\pi| - 1$ 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 $\pi \models \varphi$. Although LTL$_f$ and LTL share the same syntax, interpreting a formula over finite traces results in very different properties. As an example, consider the *fairness* constraint of the form $\mathsf{GF}\varphi$. In LTL, this kind of formulae means that *it is always true that in the future φ holds*. However, in LTL$_f$, this is equivalent to stating that *φ holds in the last state of the trace*. Thus, while the formula admits only infinite traces as counterexamples when interpreted in LTL, it admits finite counterexamples when interpreted as LTL$_f$. It holds more generally that, as stated in (De Giacomo and Vardi, 2013), *direct nesting of temporal operators yields un-interesting formulae in LTL$_f$*.

Automaton associated to LTL$_f$ formulae. Each LTL$_f$ formula φ over $\mathscr{A}$ can be associated to a minimal finite-state automaton $\mathscr{M}(\varphi)$ over the alphabet $2^{\mathscr{A}}$ such that for any trace π it holds that $\pi \models \varphi$ iff π, is accepted by $\mathscr{M}(\varphi)$ (De Giacomo and Vardi, 2013; De Giacomo and Favorito, 2021). A common assumption in LTL$_f$ applications to Process Mining, referred to as *Declare assumption* (De Giacomo et al., 2014) or *simplicity assumption*

Table 1. *Some* declare *templates as LTL$_p$ formulae. 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 are highlighted in red. The* succession *(resp.* alternateSuccession, chainSuccession*) template is defined as the conjunction of (*alternate, chain*)* response *and* precedence *templates*

Template	LTL$_p$
$Choice(a, b)$	$\mathsf{F}(a \vee b)$
$ExclusiveChoice(a, b)$	$Choice(a, b) \wedge \neg(\mathsf{F}a \wedge \mathsf{F}b)$
$RespondedExistence(a, b)$	$\mathsf{F}a \rightarrow \mathsf{F}b$
$Coexistence(a, b)$	$RespondedExistence(a, b) \wedge$ RespondexExistence(b,a)
$Response(a, b)$	$\mathsf{G}(a \rightarrow \mathsf{F}b)$
$Precedence(a, b)$	$\neg b\ \mathsf{W}\ a$
$Alt.Response(a, b)$	$\mathsf{G}(a \rightarrow \mathsf{X}(\neg a \mathsf{U} b))$
$Alt.Precedence(a, b)$	$Precedence(a, b) \wedge \mathsf{G}(b \rightarrow \mathsf{X}_w Precedence(a, b))$
$ChainResponse(a, b)$	$\mathsf{G}(a \rightarrow \mathsf{X}b)$
$ChainPrecedence(a, b)$	$\mathsf{G}(\mathsf{X}b \rightarrow a) \wedge \neg b$

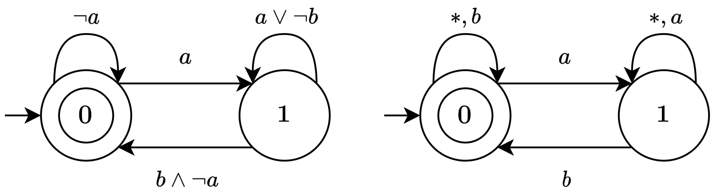


Fig. 1. Left: minimal automaton for the LTL$_f$ formula $\varphi = \mathsf{G}(a \rightarrow \mathsf{XF}b)$. Models (labeling the transitions) represent sets of symbols; right: minimal automaton for φ interpreted as a LTL$_p$ formula, where $*$ denotes any $x \in \mathscr{A} \setminus \{a, b\}$. A comma on edges denotes multiple transitions.

(Chiariello et al., 2023), is that exactly one activity occurs in each state. LTL$_f$ with this additional restriction is known as LTL$_p$ (Fionda and Greco, 2018). The assumption has the following practical implication: given a LTL$_p$ formula φ, the minimal automaton $\mathscr{M}(\varphi)$ of φ can be simplified into a deterministic automaton over $\mathscr{A}$ (Chiariello et al., 2023), as shown in Figure 1.

2.3 Declare *modeling language*

Declare (van der Aalst et al., 2009) is a declarative process modeling language that consists of a set of *templates* that express temporal properties of process execution traces. The semantics of each Declare template is defined in terms of an underlying LTL$_p$ formula. Table 1 provides the LTL$_p$ definition of some Declare templates, as reported in (De Giacomo et al., 2014). 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

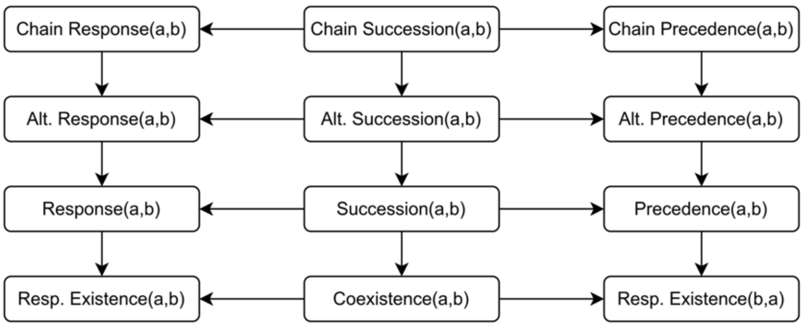


Fig. 2. *Subsumption hierarchy* between declare relation and existence templates. The arrow $t_1 \rightarrow t_2$ denotes that $\pi \models t_1 \implies \pi \models t_2$, e.g. t_1 is more specific (*subsumes*) than t_2.

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, modeling 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 (Cecconi et al., 2022).

Main declare constraints. Declare constraints are arranged into "chains", that strengthen/weaken the basic relation templates $Response(a, b)$ and $Precedence(a, b)$. The $Response(a, b)$ constraint specifies that whenever a occurs, b must occur after, while the $Precedence(a, b)$ constraint states that b can occur only if a was executed before. The constraints $AlternateResponse(a, b)$ and $AlternatePrecedence(a, b)$ strengthen respectively the $Response(a, b)$ and $Precedence(a, b)$ constraints, imposing that involved activities must "alternate", for example once the activation occurs, the target must occur before the activation re-occurs. The constraint $ChainResponse(a, b)$ imposes that the target must immediately follow the activation (i.e.,, $\pi_t = a \rightarrow \pi_{t+1} = b$); the constraint $ChainPrecedence(a, b)$ that the target must immediately precede the activation (i.e., $\pi_t = b \rightarrow \pi_{t-1} = a$). We can also weaken the temporal relation properties: the $Coexistence$ relation states that two activities must occur together; the $ExclusiveChoice$ states that exactly one of them must occur; $RespondedExistence$ that one implies the occurrence of the other. These templates do not specify temporal behavior, but only co-occurrences of events. Finally, the $Succession$ template family combines, by means of propositional conjunction, $Response$ and $Precedence$ constraints. This hierarchy of constraints, graphically represented in Figure 2, suggests a progression from more general properties, at the bottom of the hierarchy, to more specific ones, at the top. As an example, the $ChainResponse(a, b)$ is the most specific constraint in the *Response* chain; it implies $AlternateResponse(a, b)$, which in turn implies $Response(a, b)$, which implies $RespondedExistence(a, b)$.

The following example showcases the informal semantics of the Response template, which will serve as a running example in the rest of the paper in the ASP encodings section.

Example 2.1 (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, 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, that is the constraint, Response(customer_complains, address_complain). Such constraint imposes that when a customer complaint is received (activation activity), a follow-up action to address the complaint (target activity), must be executed. On the other hand, the trace $\pi =$ (customer_complains, logging_complain, address_complain, feedback_ collection) satisfies the above constraint. In contrast, the trace $\pi' =$ customer_complains, logging_complain, address_complain, customer_complains, feedback_collection does not since the second occurrence of customer_complains is not followed by any address_complain event.

Another example shows how the *Response* constraint can be further specialized.

Example 2.2 (Semantics of the AlternateResponse and ChainResponse templates).
Consider the execution traces $\pi_1 = aaabc$, $\pi_2 = abacb$, $\pi_3 = abab$. The constraint $Response(a, b)$ is valid over all these traces, while π_1 violates $AlternateResponse(a, b)$ because a repeats before b occurs after the first occurrence of a; The constraint $ChainResponse(a, b)$ is valid only on trace π_3, since both π_1 and π_2 have an a that is not immediately followed by a b.

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

Let $\mathscr{L}$ be an event log (a multiset of traces) and $\mathscr{M}$ a Declare model. The *conformance checking* task $(\mathscr{L}, \mathscr{M})$ consists in computing the subset of traces $\mathscr{L}' \subseteq \mathscr{L}$ such that for each $\pi \in \mathscr{L}'$, $\pi \models c$ for all $c \in \mathscr{M}$. Basically, conformance checking determines whether a give process execution is compliant with a process model.

Let $\mathscr{L}$ be an event log, and c a constraint. The support of c on $\mathscr{L}$, denoted by $\sigma(c, \mathscr{L})$, is defined as the fraction of traces $\pi \in \mathscr{L}$ 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 $\mathscr{L}$. Given a Declare template t and a *support threshold* $s \in (0, 1]$, the *query checking* task $(t, \mathscr{L}, s)$ 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 $\mathscr{L}$. Query checking enables to discover which instantiation of a given template exhibit high (above the threshold) support on an event log, and is a valuable tool to explore and analyze event logs (Räim et al., 2014).

Example 2.3 (Query checking and conformance checking).
Consider the template $Response(a, ?y)$ (where $?y$ is a placeholder for an activity) over the log $L = \{abab, abac, abadabd\}$, and with support $\sigma = 0.5$. All the possible instantiations of the "partial template", which we obtain by substituting the placeholder $?y$ with an activity that appears in the event log, are the constraints $Response(a, b)$, $Response(a, c)$, and $Response(a, d)$, which yield respectively support $1, \frac{1}{3}, \frac{2}{3}$ over L. The substitution $?y = a$, yields an unsatisfiable $\mathrm{LTL_f}$ formula, hence zero support. Thus, the query checking task $(Response(a, ?y), L, \sigma = 0.5)$ admits $\{Response(a, b), Response(a, d)\}$ as answers. In fact, if we perform the conformance checking task of the constraint $Response(a, c)$ the only compliant trace would be $abac$, with a support below 0.5.

Interested readers can refer to (Di Ciccio and Montali, 2022) as a starting point for Declare-related literature.

2.4 Answer set programming

ASP (Gelfond and Lifschitz, 1991; Brewka et al., 2011) is a declarative programming paradigm based on the stable models semantics, which has been used to solve many complex AI problems (Erdem et al., 2016). We now provide a brief introduction describing the basic language of ASP. We refer the interested reader to (Gelfond and Lifschitz, 1991; Brewka et al., 2011; Gebser et al., 2012) 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 $h :- b_1, \ldots, b_n$ 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 (Gelfond and Lifschitz, 1991) 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*. A choice rule is an expression of the form

$$L\ \{h_1; \ldots; h_k\}\ U \mathrel{:-} b_1, \ldots, b_n.$$

where h_i are atoms and b_i are literals. Its semantics is that whenever the body of the rule is satisfied in an interpretation I, then $L \leq |I \cap \{h_1, \ldots, h_k\}| \leq U$. Choice rules can be rewritten into a set of normal rules (Gebser et al., 2012).

A function symbol is a "composite term" of the form $f(t_1, \ldots, t_k)$ where t_i are terms and f is a predicate name. For what we are concerned in this paper, function symbols will be essentially used to model the input, acting as syntactic sugar that simplifies the modeling of records. This presentation choice makes the encodings more readable, since they allow for a compact notation of records. It is easy to see that function symbols can be easily replaced by more lengthy standard ASP specifications with additional terms in the input predicates.

We refer the reader to (Calimeri et al., 2020) for a description of more advanced ASP constructs. In the rest of the paper, ASP code examples will use Clingo (Gebser et al., 2019) syntax.

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 (De Giacomo and Vardi, 2013; Chiariello et al., 2023). Thus, one can use these two encodings to represent Declare constraints by their LTL_p definitions. The automaton-based encoding is adapted from (Chiariello et al., 2022), the syntax tree-based encoding is adapted from (Kuhlmann et al., 2023; Kuhlmann and Corea, 2024) — integrating changes to allow for the above-mentioned shared fact schema and evaluation over multiple traces. A similar encoding has also been used in (Ielo et al., 2023) to learn LTL_f formulae from sets of example traces (using the ASP-based inductive logic programming system ILASP (Law, 2023)) and to implement an ASP-based LTL_f bounded satisfiability solver (Fionda et al., 2024).

Earlier work in answer set planning (Son et al., 2006) also exploited LTL constraints using a similar syntax tree-reification approach. 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.

3.1 Encoding process traces and declare *models*

Encoding process traces. For our purposes, an event log $\mathscr{L}$ is a multiset of process traces, thus a multiset of strings over an alphabet of propositional symbols $\mathscr{A}$ (representing activities). We assume that each trace $\pi \in \mathscr{L}$ 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 $\mathscr{L}$ is encoded as $E(\mathscr{L}) = \bigcup_{\pi \in \mathscr{L}} E(\pi)$.

Example 3.1 (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 $\mathscr{A} = \{a, b, c, x, y, z\}$. 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).
```

In our encoding, activities are represented as constants that appear at least once in a trace in the event log, for example a such that $trace(\_, \_, a)$ is a fact in the input event log.

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 $\mathscr{M} = \{c_1, \dots, c_n\}$, where the subscript i uniquely indexes the constraint c_i, we denote by $E(\mathscr{M})$ the set of facts that encodes $\mathscr{M}$, that is $E(\mathscr{M}) = \bigcup_{c \in \mathscr{M}} E(c)$. Recall that in Declare $\pi \models \mathscr{M}$ if and only if $\pi \models c$ for all $c \in \mathscr{M}$, thus there is no notion of "order" among the constraints within $\mathscr{M}$ and it does not matter how indexes are assigned to constraints as long as they are unique.

Example 3.2 (Encoding a Declare model).
Consider the model $\mathscr{M}$ composed of the two constraints Response(a_1, a_2) and Precedence(a_2, a_3). $\mathscr{M}$ 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).
```

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

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

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

```
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).


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. 3. 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 conformance checking

All the Declare conformance checking encodings we propose consist of a stratified normal logic program (Lloyd, 1984; Apt et al., 1988) P_{CF} such that given a log $\mathscr{L}$ and a Declare model $\mathscr{M}$ we have that for all $\pi_i \in \mathscr{L}$, $\pi_i \models c_j \in \mathscr{M}$ if and only if the unique model of $P_{CF} \cup E(\mathscr{M}) \cup E(\mathscr{L})$ contains the atom $sat(i, j)$. The binary template Response will be our example to showcase the different encodings. Complete encodings for all the templates in Table 1 are available online.

Automaton encoding. The automaton encoding, reported in Figure 3, models Declare templates through their corresponding automaton obtained by translating the template's $\mathrm{LTL_p}$ definition (De Giacomo and Favorito, 2021). 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 $\mathscr{A} \setminus \{x, y\}$ – where x and y are the bindings of `arg_0` and `arg_1`. Activities not explicitly mentioned as within the atomic propositions in an $\mathrm{LTL_p}$ 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 5, reifies the syntax tree of a $\mathrm{LTL_p}$ formula into a set of facts, where each node represents a subformula. The semantics of temporal operators and propositional operators is defined in terms of ASP rules. Analogously to the automaton encoding, templates are defined in

```
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).
```

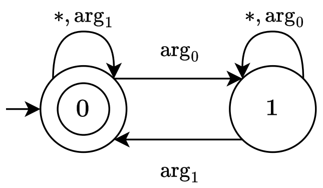


Fig. 4. Left: facts that encode the response template; right: a minimal finite state machine whose recognized language is equal to the set of models of response, under $\mathrm{LTL_p}$ semantics.

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 $\{\mathsf{U}, \mathsf{X}, \neg, \wedge\}$ which are the basic operators of $\mathrm{LTL_p}$. 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 terms `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 6 shows an example.

3.3 Encoding query checking

The query checking problem takes as input a Declare template $\mathscr{T}$, an event log $\mathscr{L}$ and consists in deciding which constraints c can be instantiated from $\mathscr{T}$ such that $\sigma(c, \mathscr{L}) \geq k$, where $\sigma(c, \mathscr{L})$ is the support and denotes the fraction of traces in $\mathscr{L}$ that are models of c. The problem has been formally introduced in (Chan, 2000) for temporal logic formulae, and in (Räim et al., 2014) it has been framed into a Process Mining setting, in the context of $\mathrm{LTL_f}$. An ASP-based solution to the problem has been provided in (Chiariello et al., 2022), 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 (Donadello et al., 2022) 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 (Fionda et al., 2023) to the $\mathrm{LTL_f}$ setting. In order to encode the query checking problem, we slightly change our input model representation, as reported in the following example.

Example 3.3 (Query checking)
Consider the query checking problem instance over the template *Response*, with both its activation and target ranging over $\mathscr{A}$. The `var_bind/3` predicate, analogously to `bind_3`,

```
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),
  {true(C,G,T,TID): trace(TID,T,_),
       T>=Ti, T<Tj} = Tj-Ti,
  true(C,H,Tj,TID).
```

Fig 5. ASP program to evaluate each sub-formula of the LTL$_p$ 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` terms.

```
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)).
```

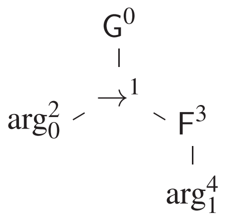


Fig. 6. Left: facts that encode the response template; right: syntax tree of the response template LTL$_p$ definition.

models that in a given template a parameter is bound to a variable. For the query checking problem, we are interested in 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 $\mathscr{A}$. 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 $(1 - s) \cdot |\mathscr{L}|$.

```
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), constraint(C,_), trace(X, _, _)} > max_violations.
```

Notice the ASP formulation can be easily generalized (by simply adding facts encoding a Declare constraint and slightly modifying the final constraint) to query check entire Declare models (e.g., multiple constraints, where a variable might occur as activation/target of distinct constraints), rather than a single constraint at a time, by adding the following facts:

```
constraint(c,"Precedence"). var_bind(c,arg_0,var(a)). var_bind(c,arg_1,var(c)).
domain(var(c),A) :- trace(_,_,A).
```

Here, the variable a is the activation of the *Response* constraint (as before), as well as the target of the *Precedence* constraint.

4 Direct ASP Encoding for Declare

The encodings described in previous section are general techniques that enable reasoning over arbitrary $\mathrm{LTL_p}$ formulae. Both encodings require, respectively, to keep track of each subformula evaluation on each time-point of a trace in the case of the syntax-tree encoding, and to keep track of DFA state during the trace traversal, in the case of the automaton encoding. However, Declare patterns do not involve *complex* temporal reasoning, with deep nesting of temporal operators. Hence Declare templates admit a succinct and direct encoding in ASP that does not keep track of such evaluation at each time-point of the trace. The encoding discussed in this section exploits this, providing an ad-hoc, direct translation of the semantics of Declare constraints into ASP rules.

The general approach we follow in defining the templates, is to model cases in which constraints fail through a `fail/2` predicate. In our encoding, a constraint c over a trace tid holds true if we are unable to produce the atom $fail(c, tid)$. 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 target 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. Note that, this approach is not based on a systematic, algorithmic rewriting, but on a template-by-template ad hoc analysis. In the rest of the section, we show the principles behind our ASP encoding for the Declare templates in Table 1.

Response template. Recall from Table 1 that Response(a, b) is defined as the $\mathrm{LTL_p}$ formula $\mathsf{G}(a \rightarrow \mathsf{F}b)$, 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, as encoded in the second rule. The following equivalences exploit the duality of temporal operators G, and F:

$$\neg\mathsf{G}(a \rightarrow \mathsf{F}b) \equiv \mathsf{F}\neg(a \rightarrow \mathsf{F}b) \equiv \mathsf{F}\neg(\neg a \vee \mathsf{F}b) \equiv \mathsf{F}(a \wedge \neg\mathsf{F}b)$$

Thus, we encode *Response* failure conditions with the following logic program:

```
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).
```

The rule above on the left yields a $witness(c,t,tid)$ atom whenever trace tid at time t satisfies $a \wedge \mathsf{F}b$. In the rule on the right, the $fail(c,tid)$ atom models that there exists some time-point t such that $\pi_t = a$, but $\neg(a \wedge \mathsf{F}b)$ thus (since $a \in \pi_t$) that $\neg\mathsf{F}b$; that is, there exists a time-point t such that $a \wedge \neg\mathsf{F}b$. Subsection D.1.1 in the appendix exemplifies this argument graphically.

Precedence template. Recall from Table 1 that the constraint Precedence(x,y) is defined as the LTL$_\text{p}$ formula $\neg y \mathsf{W} x = \mathsf{G}(\neg y) \vee \neg y \mathsf{U} x$, whose informal meaning is that *if y occurs in the trace, 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, since in the definition of the template the until operator is not under the scope of a temporal operator. The following equivalences hold:

$$\neg(\neg y \mathsf{W} x) \equiv \neg(\mathsf{G}(\neg y) \vee \neg y \mathsf{U} x) \equiv \neg\mathsf{G}(\neg y) \wedge \neg(\neg y \mathsf{U} x) \equiv \mathsf{F}y \wedge \neg(\neg y \mathsf{U} x) \equiv \mathsf{F}y \wedge y \mathsf{R} \neg x$$

We model this failure condition with the following logic rules:

```
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).
```

The rule on the left models the formula $(y \mathsf{R} \neg x)$, that is satisfied by traces where x does not occur up to the point where y first becomes true. The rule on the right models traces where x does not occur at all, but y does. We model this separately to be compliant with `clingo`'s behavior where `#min` would yield the special term `#sup` over an empty set of literals. Subsection D.2.1 in the appendix exemplifies how the rules model the constraint graphically.

Next we consider the *AlternateResponse* and *AlternatePrecedence* templates. Differently from the previous cases, mapping out failure from the LTL$_\text{p}$ formula is less intuitive, due to temporal operator nesting in the template definitions, and requires more care.

AlternateResponse template. Recall from Table 1 that $AlternateResponse(a,b)$ is defined as the LTL$_\text{p}$ formula $\mathsf{G}(a \rightarrow \mathsf{X}(\neg a \mathsf{U} b))$, whose informal meaning is that *whenever a happens, b must happen somewhere in the future and, up to that point, a must not happen.* Failure conditions follow from the this chain of equivalences:

$$\neg\mathsf{G}(a \rightarrow \mathsf{X}(\neg a \mathsf{U} b) \equiv \mathsf{F}(a \wedge \neg\mathsf{X}(\neg a \mathsf{U} b)) \equiv \mathsf{F}(a \wedge \mathsf{X}_w \neg(\neg a \mathsf{U} b)) \equiv \mathsf{F}(a \wedge \mathsf{X}_w(a \mathsf{R} \neg b))$$

Thus, the failure condition is to observe an a at time t, such that $\mathsf{X}_w(a\mathsf{R}\neg b)$ holds, that is an occurrence of a at time $t' > t$ with $b \notin \pi_k$ for all $t < k < t'$. To model the constraint, we ensure at least one b appears in-between the a occurrences. We model this by the following rules:

```
witness(C,T,TID) :-
  constraint(C, "Alternate Response"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y),
  trace(TID,T,X),
  T'' = #min{Q:
    trace(TID,Q,X), Q > T
  },
  trace(TID,T',Y),
  T'' > T',
  T' > T.
```

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

A witness here is observing an a at time t such that the next occurrence of a at time t', with at least one b in-between. To get this succinct encoding, we rely on `clingo` `#min` behavior: `#min` of an empty term set is the constant `#sup`, so the arithmetic literal `T''` `>T'` will always be true when there's no occurrence of a following the one at time T – this deals with the "last" occurence of a in the trace. If two "adjacent" occurrences of a do not have a b in-between, that does not count as a witness (as by the constraint's semantics). Subsection D.1.2 in the appendix exemplifies this argument graphically.

AlternatePrecedence template. Recall from Table 1 that the constraint $AlternatePrecedence(x,y)$ is defined as the LTL_p formula

$$(\neg y\mathsf{W}x) \wedge \mathsf{G}(b \rightarrow \mathsf{X}_w(\neg y\mathsf{W}x)) = Precedence(a,b) \wedge \mathsf{G}(b \rightarrow \mathsf{X}_w Precdence(a,b))$$

whose informal meaning is that *every time y occurs in the trace, it has been preceded by x and no y happens in-between*. To map its failure conditions, we build this chain of equivalences, where $\alpha = Precedence(a,b)$:

$$\neg(\alpha \wedge \mathsf{G}(y \rightarrow \mathsf{X}_w\alpha)) \equiv \neg\alpha \vee \mathsf{F}(y \wedge \neg\mathsf{X}_w\alpha)$$

Above we described the encoding of $Precedence(x,y)$, thus, now we focus on the second term:

$$\mathsf{F}(y \wedge \neg\mathsf{X}_w\alpha) \equiv \mathsf{F}(y \wedge \neg last \wedge \neg\mathsf{X}\alpha) \equiv \mathsf{F}(y \wedge \neg last \wedge \mathsf{X}_w\neg\alpha) \equiv \mathsf{F}(y \wedge \mathsf{X}\neg\alpha)$$

where *last* is a propositional symbol that is true only in the last state of the trace (that is, $last \equiv \mathsf{X}_w\neg\top$). Finally, by substituting $\neg\alpha$ with the failure condition of $Precedence(x,y)$:

$$\mathsf{F}(y \wedge \mathsf{X}\neg\alpha) \equiv \mathsf{F}(y \wedge \mathsf{X}(\mathsf{F}y \wedge y\mathsf{R}\neg x))$$

Thus, $AlternatePrecedence(x,y)$ admits the same failure conditions as $Precedence(a,b)$, that are reported in the rules on the left below. Furthermore, from the second term we derive the failure condition which regards the occurrence of a b that is followed by a trace suffix where $Precedence(a,b)$ does not hold; that is, the b is followed by another b with no a in between. This is modeled by the rule on the right:

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

```
fail(C,TID) :-
  constraint(C, "Alternate
       Precedence"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y),
  trace(TID, T0, Y),
  trace(TID, T2, Y),
  #count{Q:
    trace(TID,Q,X),
    Q >= T0, Q <= T2
  } = 0,
  T2 > T0.
```

Subsection D.2.2 in the appendix exemplifies this argument graphically.

ChainResponse template. Recall that from Table 1 that the constraint *ChainResponse*(x, y) is defined as the LTL$_\text{p}$ formula $\mathsf{G}(x \rightarrow \mathsf{X}y)$, whose informal meaning is that *whenever x occurs, it must be immediately followed by y.* The failure conditions follow from the following chain of equivalences:

$$\neg(\mathsf{G}(x \rightarrow \mathsf{X}y)) \equiv \mathsf{F}\neg(x \rightarrow \mathsf{X}y) \equiv \mathsf{F}\neg(\neg x \vee \mathsf{X}y) \equiv \mathsf{F}(x \wedge \neg\mathsf{X}y) \equiv \mathsf{F}(x \wedge \mathsf{X}_w\neg y)$$

That is, an x occurs that is not followed by a y – either because x occurs as the last activity of the trace or that x is followed by a different activity. We can model this by the following rule:

```
fail(C,TID) :-
  constraint(C, "Chain Response"),
  bind(C, arg_0, X), bind(C, arg_1, Y),
  trace(TID,T,X), not trace(TID,T+1,Y).
```

Subsection D.1.3 in the appendix exemplifies this argument graphically.

ChainPrecedence template. Recall that from Table 1 that the constraint *ChainResponse*(x, y) is defined as the LTL$_\text{p}$ formula $\mathsf{G}(\mathsf{X}y \rightarrow x) \wedge \neg b$, whose informal meaning is that *whenver y occurs, it must have been immediately preceded by x.* The failure conditions follow from the following chain of equivalences:

$$\neg(\mathsf{G}(\mathsf{X}y \rightarrow x) \wedge \neg y) \equiv (\mathsf{F}\neg(\mathsf{X}y \rightarrow x) \vee y \equiv \mathsf{F}(\mathsf{X}y \wedge \neg x) \vee y$$

In this case, *ChainPrecedence* fails if the predecessor of y is not x, or if y occurs in the first instant of the trace. This is modeled by the following rules:

```
fail(C, TID) :-
  constraint(C, "Chain Precedence"),
  bind(C, arg_0, X), bind(C, arg_1, Y),
  trace(TID,T+1,Y), trace(TID,T,_),
  not trace(TID,T,X).
fail(C, TID) :-
  constraint(C, "Chain Precedence"),
  bind(C, arg_1, Y), trace(TID, 0, Y).
```

Subsection D.2.3 in the appendix exemplifies this argument graphically.

Modeling the succession hierarchy. Templates in the *Succession* chain are defined as the conjunction of the respective *Precedence*, *Response* templates at the same "level of the subsumption hierarchy (see Figure 2); thus, it is possible to encode them in the `fail/2`, `witness/3` schema, since $\neg(\varphi \wedge \psi) = \neg\varphi \vee \neg\psi$. Hence, the failure conditions for *Succession*-based templates are the union of the failure conditions of its sub-formulae (all the cases were thoroughly described above in this section).

Modeling non-temporal templates. Choice (i.e., *Choice*, *ExclusiveChoice*) and Existential templates (i.e., *RespondedExistence*, *Coexistence*) are defined only in terms of conjunction and disjunction of atomic formulae (e.g., activity occurrences in the trace). Thus, they are easy to implement using normal rules whose body contains `trace/3` literals to model that $a \in \pi_i, a \notin \pi_i$. As an example, consider the constraint *RespondedExistence*(a, b) that states that *whenever a occurs in the trace, b must occur as well*. Its LTL_p definition is $\mathsf{F}(a) \rightarrow \mathsf{F}(b)$, its failure conditions follow from the following chain of equivalences:

$$\neg(\mathsf{F}(a) \rightarrow \mathsf{F}(b)) \equiv \neg(\neg\mathsf{F}(a) \vee \mathsf{F}(b) \equiv \mathsf{F}(a) \wedge \neg\mathsf{F}(b)$$

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

All the encodings for the Declare constraints in Table 1 are available in the repository.[1]

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 recent Python library for Declare-based Process mining tasks which also implements – among other tasks, such as *log generation* and *process discovery* – conformance checking and query checking functionalities. While the approaches discussed here are declarative in nature, Declare4Py implements Declare semantics by imperative procedures, based on the algorithms in (Burattin et al., 2016), that scan the input traces. Declare4Py supports also other tasks, and most importantly, supports a data-aware variant of Declare, that take into account data attributes associated to events in a trace, while here we deal with standard, control-flow only Declare. Methods will be referred to as $\text{ASP}_\mathscr{D}$, $\text{ASP}_\mathscr{A}$, $\text{ASP}_\mathscr{S}$ 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. Subsection 5.2 encompass further experimental analysis about memory consumption and behavior on longer traces for our ASP encoding.

1 https://github.com/ainnoot/padl-2024.

Table 2. *Log statistics: $|\mathscr{A}|$ is the number of activities; average $|\pi|$ is the average trace length; $|\mathscr{L}|$ is the number of traces; $|\mathscr{C}^{IV}|$ is the number of* declare *constraints above 50% support*

Log name	$\lvert\mathscr{A}\rvert$	Average $\lvert\pi\rvert$	Max $\lvert\pi\rvert$	Min $\lvert\pi\rvert$	$\lvert\mathscr{L}\rvert$	$\lvert\mathscr{C}^{IV}\rvert$
Sepsis Cases (SC)	16	14.5	185	3	1050	76
Permit Log (PL)	51	12.3	90	3	7065	26
BPI Challenge 2012 (BC)	23	12.6	96	3	13087	10
Prepaid Travel Cost (PC)	29	8.7	21	1	2099	52
Request For Payment (RP)	19	5.4	20	1	6886	52
International Declarations (ID)	34	11.2	27	3	6449	152
Domestic Declarations (DD)	17	5.4	24	1	10500	52

Data. We validate our approach on real-life event logs from past BPI Challenges (Lopes and Ferreira, 2019). These event logs are well-known and actively used in Process Mining literature. For each event log $\mathscr{L}_i$, we use D4Py to mine the set of Declare constraints $\mathscr{C}_i$ whose support on $\mathscr{L}_i$ is above 50%. Then, we define four models, $\mathscr{C}_i^{I}, \mathscr{C}_i^{II}, \mathscr{C}_i^{III}, \mathscr{C}_i^{IV}$, containing respectively the first 25%, 50%, 75%, and 100% of the constraints in a random shuffling of $\mathscr{C}_i$, such that $\mathscr{C}_i^{I} \subset \mathscr{C}_i^{II} \subset \mathscr{C}_i^{III} \subset \mathscr{C}_i^{IV}$. 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 logs to a set of facts. The translation time is included in the measurement times and is comparable with the time taken by D4Py.

Execution environment. The experiments in this section were executed on an Intel(R) Xeon(R) Gold 5118 CPU @ 2.30 GHz, 512 GB RAM machine, using Clingo version 5.4.0, Python 3.10, D4Py 1.0 and `pyrunlim`[2] to measure resources usage. Experiments were run sequentially. All data and scripts to reproduce our experiments are available in the repository.

5.1 Conformance checking and query checking on real-world logs

Conformance checking. We consider the conformance checking tasks $(\mathscr{L}_i, \mathscr{M})$, with $\mathscr{M} \in \{\mathscr{C}_i^{I}, \mathscr{C}_i^{II}, \mathscr{C}_i^{III}, \mathscr{C}_i^{IV}\}$, over the considered logs and its Declare models. Figure 7 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 D4Py on considered tasks. ASP$_{\mathscr{A}}$ and D4Py perform similarly, whereas ASP$_{\mathscr{S}}$ is less efficient. It must also be noticed that D4Py, beyond computing whether a trace is compliant or not with a given constraint, also stores additional information such as the number of times a constraint is violated, or activate, while the ASP encodings do not. However, the automata and direct encoding

2 https://github.com/alviano/python/tree/master/pyrunlim

Table 3. *Runtime in seconds for conformance checking on $\mathscr{C}^{IV}$*

Log	ASP$_{\mathscr{D}}$	D4Py	ASP$_{\mathscr{A}}$	ASP$_{\mathscr{S}}$
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 4. *Cumulative runtime in seconds for query checking tasks*

Log	ASP$_{\mathscr{D}}$	D4Py	ASP$_{\mathscr{A}}$	ASP$_{\mathscr{S}}$
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

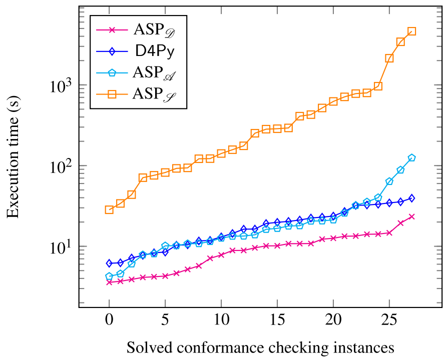


Fig. 7. Conformance checking cactus.

can be straightforwardly extended in such sense; in particular, similar “book-keeping” in the direct encoding is performed by the `fail/3` and `witness/3` atoms.

Query checking. We consider the query checking instances $(t, \mathscr{L}_i, s)$ 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 $\mathscr{L}_i$ is a log. Figure 8 summarizes the results in a cactus plot, and Table 4 aggregates the same data on the log dimension, best runtime in bold. ASP$_{\mathscr{D}}$ is again the best method overall, outperforming other ASP-based methods with the exception of ASP$_{\mathscr{A}}$ on the RP log tasks. Again, ASP$_{\mathscr{A}}$ and D4Py perform similarly and ASP$_{\mathscr{S}}$ is the worst.

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*

	Conformance Checking				Query Checking			
Log	ASP$_{\mathscr{D}}$	D4Py	ASP$_{\mathscr{A}}$	ASP$_{\mathscr{S}}$	ASP$_{\mathscr{D}}$	D4Py	ASP$_{\mathscr{A}}$	ASP$_{\mathscr{S}}$
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

Discussion. In the comparative evaluation of the different methods for conformance and query checking tasks, our direct encoding approach ASP$_{\mathscr{D}}$ showed better performance compared to both D4Py and other ASP-based methods, as reported in Table 3 and Table 4 (and in Figures 7 and 8). As shown in our experiments, ASP$_{\mathscr{D}}$ not only outperformed in terms of runtime the other methods but also offers a valuable alternative when considering overall efficiency. Notably, ASP$_{\mathscr{A}}$ and D4Py showed similar performances, with ASP$_{\mathscr{A}}$ slightly outperforming in certain instances, particularly in the RP log, but ASP$_{\mathscr{S}}$ exhibited less efficiency in nearly all the logs.

From the point of view of memory consumption (Table 5), D4Py proved to be the most efficient in query checking tasks. This can be attributed to its imperative implementation that allows for an *"iterate and discard"* approach to candidate assignments, avoiding the need for their explicit grounding required by ASP-based techniques. ASP$_{\mathscr{S}}$ is the least efficient method regarding memory usage, consistently across both types of tasks and all logs. We conjecture the significant increase in maximum memory usage is the primary factor contributing to the runtime performance degradation in both tasks, that might make the encodings an interesting benchmark for compilation-based ASP systems (Mazzotta et al., 2022; Cuteri et al., 2023; Dodaro et al., 2024; Cuteri et al., 2023). In fact, ASP$_{\mathscr{S}}$ proved to be the less efficient in both memory consumption and running time. Furthermore, we observe that in conformance checking ASP$_{\mathscr{D}}$ is more efficient w.r.t. memory consumption when compared to D4Py, and, when combined, these two methods collectively show better memory efficiency when compared to other ASP-based methods, translating in lower running times.

The obtained results clearly indicates the lower memory requirements of D4Py, demonstrating its applicability in resource-constrained environments. However, the compact and declarative nature of ASP provides an efficient means to implement Declare constraints, as demonstrated by the performance of ASP$_{\mathscr{D}}$, which is especially suitable in environments where memory is less of a constraint and execution speed is a key factor. Furthermore, the ASP-based approaches can be more readily extended, in a pure declarative way, to perform, for example, query-checking of multiple Declare constraints in a matter of few extra rules.

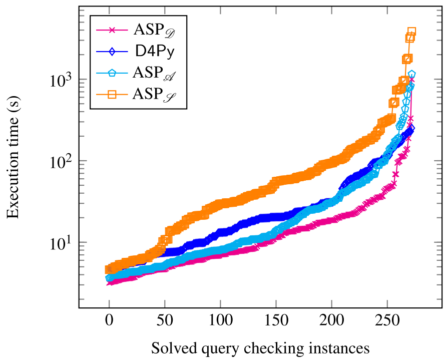


Fig. 8. Query checking cactus plot.

An intuitive reason for such a memory usage gap in the conformance checking tasks is the following. As noted in the encodings section, both the automata encoding and the syntax tree encoding rely on a 4-ary predicate ($true(TID, C, X, T)$, $cur\_state(TID, C, X, T)$ respectively), that keeps track of subformulae evaluation on each instant of the trace and current state in the constraint DFA for each instant of the trace. In the case of the automaton encoding, X does not index subtrees but states of the automaton – all the other terms have the same meaning. During the grounding of the logic programs, this yields a number of symbolic atoms that scales linearly w.r.t. the total number of events in the log (i.e.,, the sum of lenghts of the traces in the log) and linearly in the number of nodes in the syntax tree/states in the automaton. The gap between the formula encoding and the automata encoding is explained by the fact that the LTL$_\mathrm{f}$ to symbolic automaton translation, albeit 2-EXP in the worst case, results usually in *more compact* automatons in the case of the Declare patterns. For example, considering the Response template: its formula definition is $\mathsf{G}(a \rightarrow \mathsf{X}b)$, with a size (number of subformulae) of 5, while its automaton as 2 states. Furthermore, since Declare assumes simplciity, (e.g., has $|\pi_i| = 1$), the symbolic automaton can be further simplified for some constraints. This yields, overall, *less states w.r.t the number of nodes in the parse tree* – that reduces the number of ground symbols. Total number of events is the same for both encodings, since both encodings share the same input fact schema. On the other hand, as we sketched in Section 3, the direct encoding explicitly models *how constraints are violated*, thus for most templates, it produces less atoms, since we yield at most one `witness/3` atom for each occurrence of the constraint activation, and at most one `fail/2` atom for each constraint and each trace. Table 6 reports the number of generated symbols on the three different encodings to conformance check $\mathscr{C}^{\mathrm{IV}}$ on the Sepsis Cases event log, where (see Table 3) conformance checking shows a noticeable wide gap in runtime between the encodings, although remaining manageable runtime-wise for all the three ASP encodings. This confirms our intuitive explanation. In the case of the query checking task, being a classic guess & check ASP encoding, the performance depends on many factors, and it is difficult to pinpoint – size of the search space (quadratic in $|\mathscr{A}|$), order of branching while searching the model, number of constraints grounded, the nature itself of traces in the log. In this context, trade-offs between runtime and memory consumption are expected.

Table 6. *Metrics comparison for conformance checking of $\mathscr{C}^{IV}$ over the sepsis cases event log, with ASP encodings ASP$_{\mathscr{D}}$, ASP$_{\mathscr{S}}$, and ASP$_{\mathscr{A}}$. Execution times do not take into account XES input parsing and output parsing (as in Table 3, but are performed directly on facts representation of the input. Thus, reported times slightly differ from Table 3, although being executed on the same log*

	Method		
Metric	ASP$_{\mathscr{D}}$	ASP$_{\mathscr{S}}$	ASP$_{\mathscr{A}}$
Symbols	117176	4593770	1329322
Rules	143106	5953435	1345738
Execution time (s)	0.59	123.55	10.75
Max Memory (MB)	60.94	5756.16	300.81

5.2 Behavior on longer traces

Lastly, we analyze how the automata-based and direct ASP encodings scale w.r.t the length of the traces. We generate synthetic event logs (details in the appendix) of 1000 traces of fixed length (up to 1000 events) for each constraint, and we perform conformance checking with respect to a single constraint, chosen among the *Response* and *Precedence* hierarchies. Table 7 reports the result of our experiment. We compare only automata and direct encodings, since the previous analysis make it evident the syntax tree encoding is subpar due to memory consumption. The direct encoding yields a slight, but consistent, advantage time-wise w.r.t the automata encoding, and about half of peak memory consumption. Recall this is for a *single* constraint, and usually Declare models are composed of multiple Declare constraints, so this gap is expected to widen even more in conformance checking real Declare models. This is consistent with our previous experiments, see Table 3.

Moreover, we include a comparison between the direct encoding and D4Py over the same synthetic logs. For each synthetic log in Table 7, a point (x, y) in the scatter plot of Figure 9 means that the conformance checking task is solved in x seconds using the direct encoding and y seconds using D4Py.[3] Again, these results are consistent with observed behavior over real-world event logs.

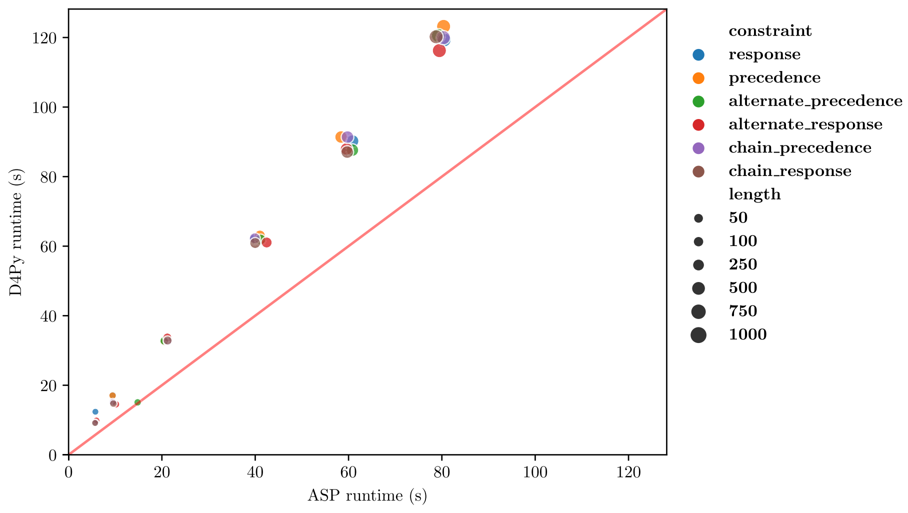
6 Conclusion

Declare is a declarative process modeling language, which describes processes by sets of temporal constraints. Declare specifications can be expressed as LTL$_p$ formulae, and traditionally have been evaluated by executing the equivalent automata (Di Ciccio and Montali, 2022), regular expressions (Di Ciccio and Mecella, 2015), or procedural approaches (de Leoni and van der Aalst, 2013). Translation-based approaches (on automata, or syntax trees) are at the foundation of existing ASP-based solutions (Chiariello et al., 2022; Kuhlmann et al., 2023). This paper proposes a novel direct

3 Note that time measurements of Figure 9 and Table 7 are not directly comparable, since in Table 7 input logs were stored as ASP facts, while in Figure 9 input logs were XES files.

Table 7. *Comparing metrics between different ASP encodings, with increasing trace lenghts. An entry is an ordered pair (t, m) where t is the runtime (seconds), m the memory peak (MBs)*

	Response		Precedence	
$\|\pi\|$	$\mathrm{ASP}_{\mathscr{D}}$	$\mathrm{ASP}_{\mathscr{A}}$	$\mathrm{ASP}_{\mathscr{D}}$	$\mathrm{ASP}_{\mathscr{A}}$
50	(0.898 s, 175.922 MB)	(1.603 s, 463.297 MB)	(1.055 s, 69.926 MB)	(1.186 s, 316.082 MB)
100	(1.877 s, 103.254 MB)	(2.429 s, 326.863 MB)	(1.660 s, 110.941 MB)	(2.458 s, 318.984 MB)
250	(4.249 s, 216.777 MB)	(7.589 s, 573.191 MB)	(4.263 s, 316.070 MB)	(6.489 s, 660.438 MB)
500	(9.111 s, 401.844 MB)	(15.604 s, 871.145 MB)	(8.701 s, 385.801 MB)	(16.096 s, 845.754 MB)
750	(13.121 s, 505.500 MB)	(29.809 s, 1427.172 MB)	(12.873 s, 497.191 MB)	(28.641 s, 1409.781 MB)
1000	(18.693 s, 784.520 MB)	(40.228 s, 1725.414 MB)	(17.416 s, 764.488 MB)	(39.589 s, 1717.375 MB)
	Alt. Response		**Alt. Precedence**	
$\|\pi\|$	$\mathrm{ASP}_{\mathscr{D}}$	$\mathrm{ASP}_{\mathscr{A}}$	$\mathrm{ASP}_{\mathscr{D}}$	$\mathrm{ASP}_{\mathscr{A}}$
50	(1.397 s, 215.266 MB)	(1.679 s, 461.453 MB)	(1.016 s, 635.871 MB)	(1.751 s, 741.715 MB)
100	(2.101 s, 164.242 MB)	(2.476 s, 461.355 MB)	(6.232 s, 635.867 MB)	(3.045 s, 741.715 MB)
250	(4.636 s, 207.594 MB)	(7.735 s, 562.719 MB)	(4.653 s, 207.570 MB)	(8.123 s, 855.000 MB)
500	(9.419 s, 404.379 MB)	(16.354 s, 869.719 MB)	(9.121 s, 401.465 MB)	(17.967 s, 995.680 MB)
750	(14.057 s, 503.562 MB)	(30.529 s, 1429.836 MB)	(14.646 s, 737.258 MB)	(31.910 s, 1577.168 MB)
1000	(19.415 s, 783.363 MB)	(43.013 s, 1726.945 MB)	(20.054 s, 823.027 MB)	(49.133 s, 1714.512 MB)
	Chain Response		**Chain Precedence**	
$\|\pi\|$	$\mathrm{ASP}_{\mathscr{D}}$	$\mathrm{ASP}_{\mathscr{A}}$	$\mathrm{ASP}_{\mathscr{D}}$	$\mathrm{ASP}_{\mathscr{A}}$
50	(1.116 s, 69.902 MB)	(1.245 s, 349.805 MB)	(0.922 s, 166.375 MB)	(1.245 s, 452.973 MB)
100	(1.818 s, 349.098 MB)	(2.563 s, 587.199 MB)	(1.798 s, 166.371 MB)	(2.610 s, 452.969 MB)
250	(4.972 s, 349.805 MB)	(6.891 s, 655.785 MB)	(4.678 s, 206.789 MB)	(7.862 s, 559.523 MB)
500	(9.527 s, 378.543 MB)	(16.778 s, 859.781 MB)	(9.882 s, 405.285 MB)	(16.104 s, 850.094 MB)
750	(13.120 s, 720.094 MB)	(27.913 s, 1539.625 MB)	(13.165 s, 744.195 MB)	(28.160 s, 1605.547 MB)
1000	(17.960 s, 772.133 MB)	(40.541 s, 1731.586 MB)	(17.837 s, 794.250 MB)	(41.960 s, 1739.156 MB)



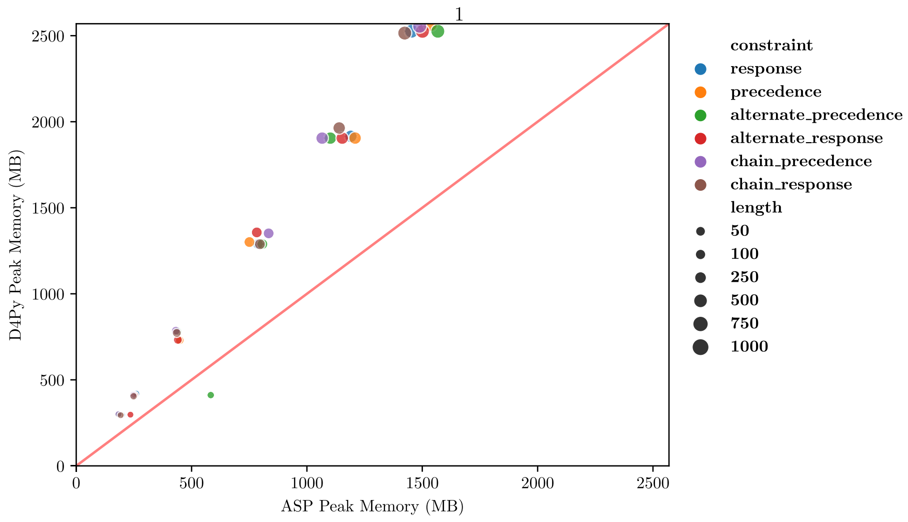


Fig. 9. Comparison of runtime (upper) and peak memory usage (lower) for the ASP direct encoding and D4Py, on synthetic logs used in Table 7.

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, albeit limited to Declare, outperforms other ASP-based methods in terms of execution time and peak memory consumption and compares favourably with dedicated libraries. Thus, ASP provides a compact, declarative, and efficient way to implement Declare constraints in the considered tasks. Interesting future avenues of research are to investigate whether this approach can be extended to *data-aware* (de Leoni and van der Aalst, 2013) variants of

Declare that take into account data attributes associated to events in a trace, to probabilistic extensions of Declare (Alman et al., 2022; Alviano et al., 2024; Vespa et al., 2024) that associate uncertainty (in terms of probabilities) to constraints in a declarative process model, as well as other Declare-based Process Mining tasks.

Acknowledgements

We thank the anonymous reviewers for their careful reading of our manuscript and their many insightful comments and suggestions.

This work was partially supported by the Italian Ministry of Research (MUR) under PRIN project PINPOINT – CUP H23C22000280006; partially by the HypeKG Project of Italian Ministry of Universities and Research (MUR) within Progetti di Rilevante Interesse Nazionale (PRIN) 2022 Program under Grant 2022Y34XNM (CUP: H53D23003710006); partially by the DISTORT Project of Italian Ministry of Universities and Research (MUR) within Progetti di Rilevante Interesse Nazionale Prin 2022 PNRR Program under Grant P2022KHTX7 (CUP: H53D230 08170001) and European Union – Next Generation EU; partially by the PNRR projects FAIR "Future AI Research" – Spoke 9 – WP9.1 and WP9.2 – CUP H23C22000860006, through the NRRP MUR Program funded by the NextGenerationEU, by the project "Borgo 4.0," POR Campania FESR 2014-2020, by the French government as part of France 2030 (Grant agreement n°ANR-19-PI3A-0004), the European Union – Next Generation EU as part of the France Relance, and The Italian Ministry of Enterprise Industry and Made in Italy (MIMIT) under project EI-TWIN n. F/310168/05/X56 CUP B29J24000680005.

Competing interests

The author(s) declare none.

References

Alman, A., Maggi, F. M., Montali, M. and Peñaloza, R. 2022. Probabilistic declarative process mining. *Inf. Syst.* 109, 102033.

Alviano, M., Ielo, A. and Ricca, F. 2024. Efficient compliance computation in probabilistic declarative specifications. (*ICLP Workshops. CEUR Workshop Proceedings*, 3799, CEUR–WS.org.

Apt, K. R., Blair, H. A. and Walker, A. 1988. Towards a theory of declarative knowledge, *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann, pp. 89–148.

Brewka, G., Eiter, T. and Truszczynski, M. 2011. Answer set programming at a glance. *Communications of the Acm* 54, 12, 92–103.

Burattin, A., Maggi, F. M. and Sperduti, A. 2016. Conformance checking based on multi-perspective declarative process models. *Expert Systems with Applications* 65, 194–211.

Cabalar, P., Kaminski, R., Morkisch, P. and Schaub, T. 2019. Telingo = ASP + time, *LPNMR. Lecture Notes in Computer Science*, 11481, Springer, pp. 256–269.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Maratea, M., Ricca, F. and Schaub, T. 2020. Asp-core-2 input language format. *Theory Pract. Log. Program.* 20, 2, 294–309.

Cecconi, A., De Giacomo, G., Di Ciccio, C., Maggi, F. M. and Mendling, J. 2022. Measuring the interestingness of temporal logic behavioral specifications in process mining. *Inf. Syst.* 107, 101920.

Chan, W. 2000. Temporal-logic queries, *Computer Aided Verification*, Emerson, E. A. and Sistla, A. P. (ed.), Berlin Heidelberg, Berlin, Heidelberg: Springer, pp. 450–463.

Chesani, F., Francescomarino, C. D., Ghidini, C., Grundler, G., Loreti, D., Maggi, F. M., Mello, P., Montali, M. and Tessaris, S. 2022. Optimising business process discovery using answer set programming, *LPNMR. Lecture Notes in Computer Science*, 13416, Springer, pp. 498–504.

Chesani, F., Francescomarino, C. D., Ghidini, C., Loreti, D., Maggi, F. M., Mello, P., Montali, M. and Tessaris, S. 2023. Process discovery on deviant traces and other stranger things. *IEEE Trans. Knowl. Data Eng.* 35, 11, 11784–11800.

Chesani, F., Lamma, E., Mello, P., Montali, M., Riguzzi, F. and Storari, S. 2009. Exploiting inductive logic programming techniques for declarative process mining. *Trans. Petri Nets Other Model. Concurr.* 2, 278–295.

Chiariello, F., Fionda, V., Ielo, A. and Ricca, F. 2024. A direct ASP encoding for declare. Proceedings, M. Gebser and Sergey, I., Lecture Notes in Computer Science, *Practical Aspects of Declarative Languages - 26th International Symposium, PADL. 2024*, Springer, London, UK, 14512, 116–133, Lecture Notes in Computer Science, January 15-16, 2024.

Chiariello, F., Maggi, F. M. and Patrizi, F. 2022. Asp-based declarative process mining, *AAAI.* AAAI Press, pp. 5539–5547.

Chiariello, F., Maggi, F. M. and Patrizi, F. 2023. From LTL on process traces to finite-state automata, *BPM. CEUR WP*, 3469, pp. CEUR–WS.org,127-131.

Chinosi, M. and Trombetta, A. 2012. BPMN: an introduction to the standard. *Comput. Stand. Interfaces* 34, 1, 124–134.

Cuteri, A., Mazzotta, G. and Ricca, F. 2023. Compilation-based techniques for evaluating normal logic programs under the well-founded semantics. (*CILC. CEUR Workshop Proceedings*, 3428, CEUR–WS.org.

De Giacomo, G., De Masellis, R. and Montali, M. 2014. Reasoning on LTL on finite traces: insensitivity to infiniteness, *AAAI.* AAAI Press, 1027

De Giacomo, G. and Favorito, M. 2021. Compositional approach to translate ltlf/ldlf into deterministic finite automata, *ICAPS.* AAAI Press, pp. 122–130.

De Giacomo, G. and Vardi, M. Y. 2013. Linear temporal logic and linear dynamic logic on finite traces, *IJCAI. IJCAI/AAAI*, pp. 854–860.

de Leoni, M. and van der Aalst, W. M. P. 2013. Data-aware process mining: discovering decisions in processes using alignments, *SAC. ACM*, pp. 1454–1461.

Di Ciccio, C. and Mecella, M. 2015. On the discovery of declarative control flows for artful processes. *ACM Trans. Manag. Inf. Syst.* 5, 1-24, 37–37.

Di Ciccio, C. and Montali, M. 2022. Declarative process specifications: reasoning, discovery, monitoring, *Process Mining Handbook. Lecture Notes in Business Information Processing*, 448, Springer, pp. 108–152.

Dodaro, C., Mazzotta, G. and Ricca, F. 2023. Compilation of tight ASP programs, *ECAI. Frontiers in Artificial Intelligence and Applications*, 372, IOS Press, pp. 557–564.

Dodaro, C., Mazzotta, G. and Ricca, F. 2024. Blending Grounding and Compilation for Efficient ASP Solving. *IN Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning*, 317–328.

Donadello, I., Riva, F., Maggi, F. M. and Shikhizada, A. 2022. Declare4py: a python library for declarative process mining, *BPM (PhD/Demos). CEUR WP*, 3216, pp. CEUR–WS.org,117-121.

Dwyer, M. B., Avrunin, G. S. and Corbett, J. C. 1999. Patterns in property specifications for finite-state verification, *ICSE. ACM*, pp. 411–420.

Erdem, E., Gelfond, M. and Leone, N. 2016. Applications of answer set programming. *Ai Magazine* 37, 3, 53–68.

Finkbeiner, B. and Sipma, H. 2004. Checking finite traces using alternating automata. *Form. Meth.Syst. Des.* 24, 2, 101–127.

Fionda, V. and Greco, G. 2018. LTL on finite and process traces: complexity results and a practical reasoner. *Journal of Artificial Intelligence Research* 63, 557–623.

Fionda, V., Ielo, A. and Ricca, F. 2024. Ltlf2asp: ltlf bounded satisfiability in ASP, *LPNMR. Lecture Notes in Computer Science*, 15245, Springer, pp. 373–386.

Fionda, V., Ielo, A., Ricca, F. and KR. 2023-September 2-8, 2023. Logic-based composition of business process models. Marquis, P., Son, T. C. and Kern-Isberner, G., *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning*, Rhodes, Greece, pp. 272–281, *2023*

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2012. Answer set solving in practice, *Synthesis Lectures On Artificial Intelligence and Machine Learning.* Morgan & Claypool Publishers.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.* 19, 1, 27–82.

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

Greenman, B., Prasad, S., Stasio, A. D., Zhu, S., Giacomo, G. D., Krishnamurthi, S., Montali, M., Nelson, T. and Zizyte, M. 2024. Misconceptions in finite-trace and infinite-trace linear temporal logic, *FM (1). Lecture Notes in Computer Science*, 14933, Springer, pp. 579–599.

Greenman, B., Saarinen, S., Nelson, T. and Krishnamurthi, S. 2023. Little tricky logic: misconceptions in the understanding of LTL. *Art Sci. Eng. Program.* 7, 2.

Ielo, A., Law, M., Fionda, V., Ricca, F., De Giacomo, G. and Russo, A. 2023. Towards ilp-based ltlf passive learning, *ILP. (To Appear)*

Ielo, A., Ricca, F. and Pontieri, L. 2022. Declarative mining of business processes via ASP, *PMAI@IJCAI. CEUR WP*, 3310, pp. CEUR–WS.org,105-108.

Kuhlmann, I. and Corea, C. 2024. Inconsistency measurement in ltl f based on minimal inconsistent sets and minimal correction sets, *SUM. Lecture Notes in Computer Science*, 15350, Springer, pp. 217–232.

Kuhlmann, I., Corea, C. and Grant, J. 2023. Non-automata based conformance checking of declarative process specifications based on ASP, *Business Process Management Workshops. Lecture Notes in Business Information Processing*, 492, Springer, pp. 396–408.

Lamma, E., Mello, P., Riguzzi, F. and Storari, S. 2007. Applying inductive logic programming to process mining, *Inductive Logic Programming, 17th International Conference, ILP. 2007*, Revised Selected Papers, H. Blockeel, Ramon, J., Shavlik, J. W. and Tadepalli, P., 4894. Corvallis, OR, USA: Springer, pp. 132–146, Lecture Notes in Computer Science, June 19-21, 2007

Law, M. 2023. Conflict-driven inductive logic programming. *Theory Pract. Log. Program.* 23, 2, 387–414.

Lloyd, J. W. 1984. *Foundations of Logic Programming. Symbolic Computation.* Springer, Berlin, Heidelberg.

Lopes, I. F. and Ferreira, D. R. 2019. A survey of process mining competitions: the BPI challenges 2011-2018, *Business Process Management Workshops. Lecture Notes in Business Information Processing*, 362, Springer, pp. 263–274.

Mannhardt, F. and Blinde, D. 2017. Analyzing the trajectories of patients with sepsis using process mining. Gulden, J., Nurcan, S., Reinhartz-Berger, I., Guédria, W., Bera, P., Guerreiro, S., Fellmann, M. and Weidlich, M., CEUR Workshop Proceedings, *Joint Proceedings of the Radar tracks at the 18th International Working Conference on Business Process Modeling, Development and Support (BPMDS), and the 22nd International Working Conference on Evaluation and Modeling Methods for Systems Analysis and Development (EMMSAD), and the 8th International Workshop on Enterprise Modeling and Information Systems Architectures (EMISA) co-located with the 29th International Conference on Advanced Information Systems Engineering 2017 (CAiSE 2017)*, Essen, Germany, 1859, CEUR–WS.org,72-80, CEUR Workshop Proceedings, June 12-13, *2017*

Mazzotta, G., Ricca, F. and Dodaro, C. 2022. Compilation of aggregates in ASP systems, *AAAI*. AAAI Press, pp. 5834–5841.

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

Pnueli, A. 1977. The temporal logic of programs, *FOCS*. IEEE Computer Society, pp. 46–57.

Räim, M., Di Ciccio, C., Maggi, F. M., Mecella, M. and Mendling, J. 2014. Log-based understanding of business processes through temporal logic query checking. (*OTM Conferences. Lecture Notes in Computer Science*, Springer, 8841, 75–92.

Rovani, M., Maggi, F. M., de Leoni, M. and van der Aalst, W. M. P. 2015. Declarative process mining in healthcare. *Expert Systems with Applications* 42, 23, 9236–9251.

Son, T. C., Baral, C., Nam, T. H. and McIlraith, S. A. 2006. Domain-dependent knowledge in answer set planning. *ACM Trans. Comput. Log.* 7, 4, 613–657.

van der Aalst, W. M. P. 1998. The application of petri nets to workflow management. *J. Circuits Syst. Comput.* 8, 01, 21–66.

van der Aalst, W. M. P. 2022. Process mining: a 360 degree overview, *PM Handbook. LNIP*, 448, Springer, pp. 3–34.

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.

Van der Aalst, W. M. and Weijters, A. J. 2004. Process mining: a research agenda. *Computers in Industry* 53, 3, 231–244.

Vespa, M., Bellodi, E., Chesani, F., Loreti, D., Mello, P., Lamma, E. and Ciampolini, A. 2024. Probabilistic compliance in declarative process mining. (*PMAI@ECAI. CEUR Workshop Proceedings*, 3779, CEUR–WS.org,11-22.

Weske, M. 2024. *Business Process Management - Concepts, Languages, Architectures.* Springer, Berlin, Heidelberg.

Appendix A Validation

In this paper, we do not investigate whether there exists an automatic translation from arbitrary LTL_f or LTL_p into ASP rules (beyond indirect techniques), but we focus solely on the Declare constraints, providing ad-hoc encodings written by hand w.r.t their informal semantics. To validate the correctness of these encodings, in our use case, we applied a bounded model checking-like approach, searching for a "behavioral counterexample" between our encoding of each particular constraint and a *ground truth* logic program - which corresponds to the logic program that captures the behavior of the state machine which is equivalent to the LTL_p definition of the constriant at hand. That is, given a Declare constraint c, we consider its LTL_p definition φ_c and its corresponding DFA $\mathscr{M}_{\varphi_c}$, and its direct encoding P_c. In particular, if we are able to find a trace π such that $\pi \models P_c$, but $\pi \notin \mathscr{L}(\pi)$ - or viceversa, $\pi \not\models P_c$ but $\pi \in \mathscr{L}(\pi)$ - then π is a witness of the fact that our direct encoding P_c for the Declare constraint c encodes a wrong behavior (accepting a trace it should not accept, or rejecting a trace it should accept). We tested our encodings for counterexamples of length up to 20 over the $\{a, b, *\}$ alphabet, where a plays the role of the constraint activation, b its target, and $*$ a placeholder for "other characters". As discussed in the automata encoding section, if two propositional symbols do not appear in a LTL_p formula, they are interchangeable in a trace and won't alter the satisfaction of the trace.

```
#const t=20.
time(0..t-1).
activity("*").
activity(A) :- bind(_,_,A).
1 { trace(1,T,A): activity(A) } 1 :- time(T).
:- #count{M: sat(M,C,_)} != 1, constraint(C, _).
```

We adapt the conformance checking encoding, adding a (constant) extra term to `sat/2` predicate (in each encoding involved in the check procedure), to distinguish which evaluation method yields the $sat(\cdot,\cdot)$ atom. That is, instead of $sat(c, tid)$ to model that constraint c holds true over trace tid, we use the atoms $sat(automata, c, tid)$, $sat(adhoc, c, tid)$ and $sat(ltlf, c, tid)$ to distinguish satisfiability of the constraint c expressed in the automaton encoding, direct and syntax-tree encoding respectively. The above program is to be evaluated together with two distinct conformance checking encoding from Section 3, 4. The choice generates the search space for a LTL_p trace. The constraint discards answer sets (e.g., LTL_p traces) that are evaluated *in the same way* by two distinct encodings encodings. That is, it discards traces that are accepted by both encodings or rejected by boths encodings. This logic program yields a model if and only if two LTL_p semantics' encoding are inconsistent with each other. We tested our direct encoding, searching for counterexamples of length up to 20 over the $\{a, b, *\}$ against the automaton encoding, and we didn't find any behavioral counterexample.

Appendix B Synthetic Log Generation

To generate synthetic traces for our experiment, we use the following logic program. Each stable model corresponds to a unique trace. For each constraint in Table 1, we generate a log of 1000 traces (half positive, half negative) over an alphabet of 15 activities. To generate positive traces, we set the external atom *negative* to false, to generate negative traces we set the *negative* external atom to true using the Clingo Python API. The constant t represents the length of each trace, it is set as runtime as well. The input facts are the encoding of a Declare constraint, along with the logic program which encodes the semantics (either the automata, syntax tree or direct one). In our experiments, synthetic logs are generated with the direct encoding. To avoid *uninteresting* traces, were constraint are *vacuously* true due to absence of activation/target, we impose that both activation and target should appear at least once.

```
activity("a_0"; ...; "a_14").
#external negative.
#const t=-1.
:- t < 0.
time(0..t-1).

{ trace(0,T,A): activity(A) } = 1 :- time(T).
:- not trace(0,_,A), bind(0, arg_0, A).
:- not trace(0,_,A), bind(0, arg_1, A).
positive :- not negative.
:- not sat(_,0,0), positive.
:- sat(_,0,0), negative.
#show.
#show trace/3.
```

Appendix C An example use case for Declare-based process mining

This is an example application of Declare, regarding the *Sepsis Cases Event Log* (Mannhardt and Blinde, 2017). The log contains events logged by the information system of a dutch hopsital, concerning patients with a diagnosis/suspected diagnosis of sepsis. Here is the official description of the event log: *This real-life event log contains events of sepsis cases from a hospital. Sepsis is a life threatening condition typically caused by an infection. One case represents the pathway through the hospital. The events were recorded by the ERP (Enterprise Resource Planning) system of the hospital. There are about 1000 cases with in total 15,000 events that were recorded for 16 different activities. Moreover, 39 data attributes are recorded, for example the group responsible for the activity, the results of tests and information from checklists. Events and attribute values have been anonymized. The time stamps of events have been randomized, but the time between events within a trace has not been altered.* Suppose we are interested in conformance checking the log traces against the Declare model containing the constraints:

- $Precedence(Antibiotics, IVLiquid)$
- $ExclusiveChoice(ReleaseA, ReleaseB)$
- $ChainPrecedence(ERTriage, AdmissionIC)$

The model states that the activities $ReleaseA$, $ReleaseB$ (e.g., releasing a patient from the hospital with different types of diagnosis) are mutually exclusive. Admission in the intensive care unit ($AdmissionIC$) should be immediately preceded by a triage ($ERTriage$); The IV Liquid ($IVLiquid$) exam should not be performed before patients undergo $Antibiotics$. A Declare model could be either designed by a domain expert, according to for example its clinical practice, or clinical guidelines, or discovered automatically from data. Conformance checking allows to identify which traces are compliant, or non-compliant, with the given model, and understand which constraints are violated. To run this example:

```
clingo example_1/*
```

Suppose instead we are interested in all the activities that immediately follow the admission in intensive care ($AdmissionIC$) in at most 30% of the log. This could be useful for clinicians to analyze if clinical practices are correctly being followed. This amounts to performing the query checking task $ChainResponse(AdmissionIC, ?x)$. To run this example:

```
clingo example_2/*
```

Its output is variable bindings that satisfiy the query checking task. In both commands, the `filter.lp` file contains only projection rules onto the "output atoms". All the `*.lp` files are written according to the descriptions in Section 3 and are available at our repository. More details about the log and other process mining techniques on this log are showcased in (Mannhardt and Blinde, 2017), although they do not make use of Declare, but non-declarative process mining techniques based on Petri Nets. A detailed application of Declare-based process mining techniques in the healthcare domain is available in (Rovani et al., 2015), showcasing a case study concerning gastric cancer clinical guidelines and practice.

Appendix D Encodings

All the encodings are available under the folders `ltlf_base` (syntax tree encoding), `automata` (automata encoding), and `asp_native` (direct encoding) in our repository. Each folder contains a `templates.lp` file, which are the set of facts that encode the Declare templates as sketched in Section 3 (e.g., automata transition tables and syntax tree reification - notice this file is empty for the direct encoding), and a `semantics.lp` file which encode logic programs to evaluate automata runs over traces (for the automata encoding), temporal logic operators' semantics as normal rules (for the syntax tree encoding) and the direct translation of Declare into ASP rules for the direct encoding. For completeness, we report here the contents of `asp_native semantics.lp`, along with pictures that map each constraints' failure conditions to ASP rules.

For all constraints, a constraint c holds over a trace tid if we derive the $sat(c, tid)$ atom, that is, if we are unable to show a failure condition for c over tid:

```
sat(C, TID) :- constraint(C,_), trace(TID,_,_), not fail(C,TID).
```

D.1 response-based templates

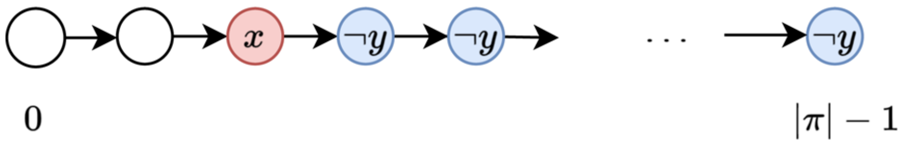
D.1.1 response

Listing 1. *Response* template.

```
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).
```

$$\mathsf{F}(x \wedge \neg \mathsf{F} y)$$



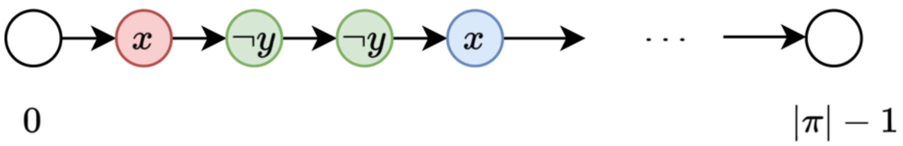
D.1.2 alternate response

Listing 2. *AlternateResponse* template.

```
witness(C,T,TID) :-
  constraint(C, "Alternate Response"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y),
  trace(TID,T,X),
  T'' = #min{Q: trace(TID,Q,X), Q > T},
  trace(TID,T',Y), T'' > T', T' > T.

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

$$\mathsf{F}(x \wedge \mathsf{X_w}(x \mathsf{R} \neg y))$$



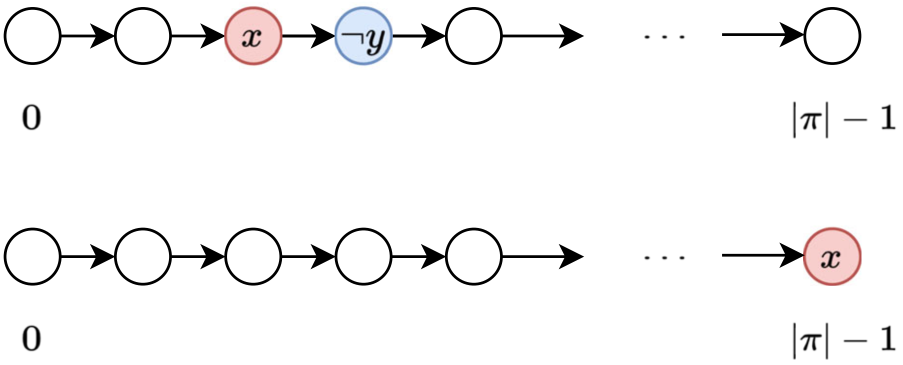
Notice that the case where the x does not occur is covered by the **#min** behavior in `clingo`, which yields the term **#sup** when **x** does not occur in the trace, hence the arithmetic literal `T'' >T'` always evaluates to true in this case.

D.1.3 chain response

Listing 3. *ChainResponse* template.

```
fail(C,TID) :-
  constraint(C, "Chain Response"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y),
  trace(TID,T,X),
  not trace(TID,T+1,Y).
```

$$\mathsf{F}(x \wedge \mathsf{X_w} \neg y)$$



Here, a single rule covers both failure cases, as *not trace*$(tid, t+1, y)$ literal is true both when the $t+1$ time instant in the trace does not exist, as well as when the $\pi_{i+1} \neq y$.

D.2 precedence-based templates

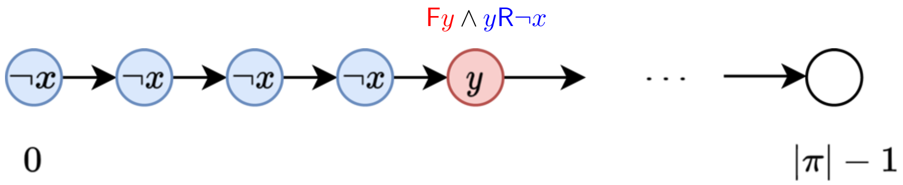
D.2.1 precedence

Listing 4. *Precedence* template.

```
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).
```

We use a dedicated rule to model that x never occurs, when it does not exist, the #min aggregate would yield the term #sup.



D.2.2 alternate precedence

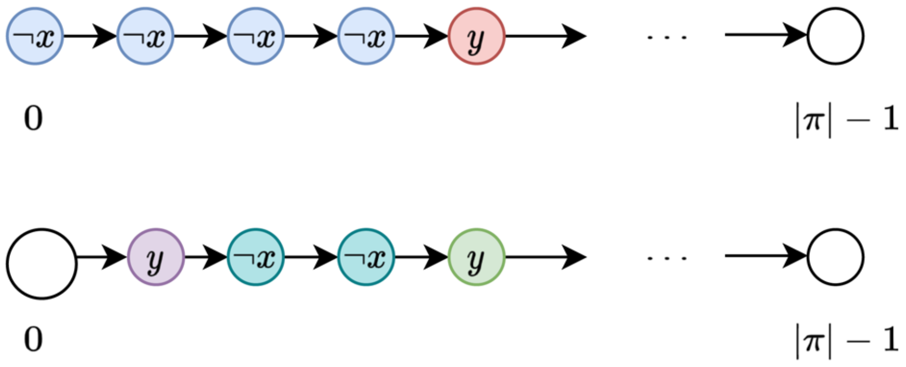
Listing 5. *AlternatePrecedence* template.

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

fail(C,TID) :-
  constraint(C, "Alternate Precedence"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y),
  trace(TID, T0, Y),
  trace(TID, T2, Y),
  T2 > T0,
  #count{Q: trace(TID,Q,X), Q >= T0, Q <= T2} = 0.
```

$$(\mathsf{F}y \wedge y\mathsf{R}\neg x) \vee \mathsf{F}(y \wedge \mathsf{X}(\mathsf{F}y \wedge y\mathsf{R}\neg x))$$



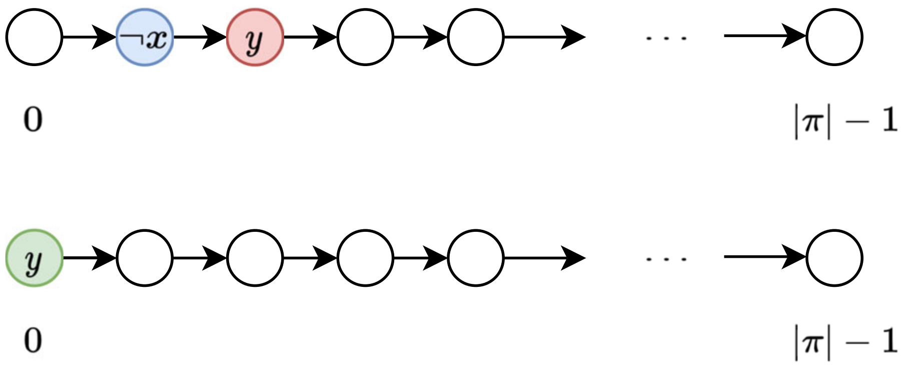
D.2.3 chain precedence

Listing 6. *ChainPrecedence* template.

```
fail(C, TID) :-
  constraint(C, "Chain Precedence"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y),
  trace(TID,T+1,Y),
  trace(TID,T,_),
  not trace(TID,T,X).

fail(C, TID) :-
  constraint(C, "Chain Precedence"),
  bind(C, arg_1, Y),
  trace(TID, 0, Y).
```

$$\mathsf{F}(\mathsf{X}y \wedge \neg x) \vee y$$



D.3 succession-based templates

Recall that Succession templates are defined as the conjunction of the corresponding Precedence, Response templates at the same level of the subsumption hierarchy (see Figure 2). Hence, its failure conditions are the union of the failure conditions of its subformulae.

D.3.1 succession

Listing 7. *Succession* template. *Same failure conditions as* Response, Precedence.

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

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

D.3.2 alternate succession

Listing 8. AlternateSuccession *template. Same failure conditions as* AlternateResponse, AlternatePrecedence.

```
witness(C,T,TID) :-
  constraint(C, "Alternate Succession"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y),
  trace(TID,T,X),
  T'' = #min{Q: trace(TID,Q,X), Q > T},
  trace(TID,T',Y), T'' > T', T' > T.

fail(C,TID) :-
  constraint(C, "Alternate Succession"),
  bind(C, arg_0, X),
  trace(TID,T,X),
  not witness(C,T,TID).

witness(C, T2, TID) :-
  trace(TID, T2, Y),
  T0 = #max{T: trace(TID, T, Y), T2> T},
  trace(TID, T1, X),
  T2 > T1, T1 > T0,
  constraint(C, "Alternate Succession"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y).

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

fail(C, TID) :-
  constraint(C, "Alternate Succession"),
  last(TID, T),
  bind(C, arg_1, Y),
  trace(TID, T, Y).
```

D.3.3 chain succession

Listing 9. *ChainSuccession* template. Same failure conditions as *ChainResponse, ChainPrecedence.*

```
fail(C,TID) :-
  constraint(C, "Chain Succession"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y),
  trace(TID,T,X),
  not trace(TID,T+1,Y).

fail(C, TID) :-
  constraint(C, "Chain Succession"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y),
  trace(TID,T+1,Y),
  trace(TID,T,_),
  not trace(TID,T,X).
```

D.4 choice, existence templates

This set of templates, the most general ones in Declare, at the bottom of the subsumption hierarchy, do not involve temporal operators in their $\mathrm{LTL_p}$ definition, but only atomic operators (e.g., activity occurrences in the whole trace). They are easily seen as projections on the `trace/3` predicate.

Listing 10. *Choice* template.

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

Listing 11. *ExclusiveChoice* template.

```
fail(C, TID) :-
  constraint(C, "Exclusive Choice"),
  bind(C, arg_0, X),
  bind(C, arg_1, Y),
  trace(TID,_,X),
  trace(TID,_,Y).

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

Listing 12. *RespondedExistence* template.

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

Listing 13. *Coexistence* template.

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

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