Declarative Process Specifications: Reasoning, Discovery, Monitoring

Claudio Di Ciccio[1(✉)] and Marco Montali[2]

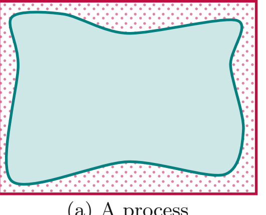
[1] Sapienza University of Rome, Rome, Italy
claudio.diciccio@uniroma1.it
[2] Free University of Bozen-Bolzano, Bolzano, Italy
montali@inf.unibz.it

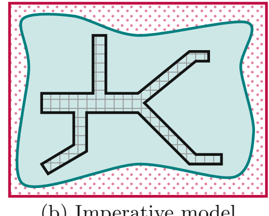
Abstract. The declarative specification of business processes is based upon the elicitation of behavioural rules that constrain the legal executions of the process. The carry-out of the process is up to the actors, who can vary the execution dynamics as long as they do not violate the constraints imposed by the declarative model. The constraints specify the conditions that require, permit or forbid the execution of activities, possibly depending on the occurrence (or absence) of other ones. In this chapter, we review the main techniques for process mining using declarative process specifications, which we call *declarative process mining*. In particular, we focus on three fundamental tasks of (1) reasoning on declarative process specifications, which is in turn instrumental to their (2) discovery from event logs and their (3) monitoring against running process executions to promptly detect violations. We ground our review on Declare, one of the most widely studied declarative process specification languages. Thanks to the fact that Declare can be formalized using temporal logics over finite traces, we exploit the automata-theoretic characterization of such logics as the core, unified algorithmic basis to tackle reasoning, discovery, and monitoring. We conclude the chapter with a discussion on recent advancements in declarative process mining, considering in particular multi-perspective extensions of the original approach.

1 Introduction

Finding a suitable balance between flexibility and control is a long-standing problem in the management of work processes [83]. Among the different approaches striving to achieve this balance, *flexibility by design* suggests to infuse flexibility in the process modeling language at hand. Declarative process modeling languages take this to the extreme: they support the specification of *what* are the relevant constraints on the temporal evolution of the process, without explicitly indicating *how* process instances should be routed to satisfy such constraints. In comparison with imperative approaches that produce "closed" representations (i.e., only those process executions explicitly foreseen in the model are allowed), declarative approaches yield "open" representations (i.e., every process execution is implicitly allowed, as long as it does not incur in the violation of some constraint).

© The Author(s) 2022
W. M. P. van der Aalst and J. Carmona (Eds.): Process Mining Handbook, LNBIP 448, pp. 108–152, 2022.
https://doi.org/10.1007/978-3-031-08848-3_4


(a) A process


(b) Imperative model

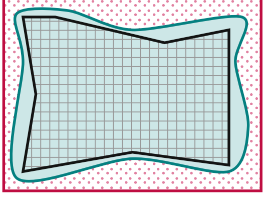

(c) Declarative specification

Fig. 1. Intuitive representation of the difference between imperative process models and declarative process specifications in the space of all execution traces. Diagram (a) represents a real process, which isolates the *allowed* (green, solid fill) behaviors from the *forbidden* (red, dotted fill) ones. Diagram (b) shows an *imperative process model* that stays within the boundaries of the process, but misses many allowed behaviors. Diagram (c) shows a *declarative process specification* that well approximates the boundaries of the process: it accepts only traces that are allowed by the process, and includes all the traces accepted by the imperative model in (b). (Color figure online)

Figure 1 depicts an intuitive representation of the difference between classical imperative process models and declarative process specifications, considering execution traces that are forbidden by the real process, allowed by the real process, and captured by the designed process specification. Imperative models (such as those based on Petri nets and related formalisms) are suited to explicitly capture control-flow patterns like sequences, choices, concurrent sections, and loops. Those patterns, in turn, lend themselves to characterize a subset of the allowed traces, but struggle in covering the whole space of execution paths in the case of loosely structured, flexible processes. In other words, they favor control over flexibility. Contrariwise, declarative specifications strive to balance flexibility and control by attempting to characterize constraints that well-separate the allowed behaviors from the forbidden ones. In other words, declarative process specifications allow us to capture not only *what is expected* to occur, but also *what should not happen*. This helps in better approximating the boundaries of the real process, containing (and extending) those captured via imperative process models.

The idea of adopting a constraint-based, declarative approach to regulate dynamic systems has been originally brought forward in different communities: in data management, to express cascaded transactional updates [26]; in multiagent systems, to regulate agent interaction protocols [88]; and in business process management, to capture subprocesses that foresee loosely-coupled control-flow conditions on their activities [85]. This idea was further developed within BPM in consequent years, leading to a series of declarative, constraint-based process modeling languages, with two prominent exponents: Declare [76] and Dynamic Condition-Response Graphs [49]. Common to all such approaches is the usage of *linear temporal/dynamic logics* (i.e., temporal/dynamic logics for sequences of events) to formally describe specifications, and the exploitation of corresponding reasoning mechanisms to tackle a variety of concrete tasks along the entire

process lifecycle, from design and model analysis to runtime execution and data analysis.

In this chapter, we focus on *declarative process mining*, that is, process mining where the input or output models are specified using declarative, constraint-based languages. Concretely, we employ the DECLARE language, but all the presented ideas seamlessly apply any language that can be formalized using logics over *finite* traces [30], which are indeed at the core of DECLARE. Focusing on finite traces reflects the intuition that every process instance is expected to complete in a finite number of steps. This aspect has a significant impact on the corresponding operational techniques, as these logics admit an automata-theoretic characterization that is based on standard *finite-state automata* [27,30], instead of automata on infinite structures, which are needed when such logics are interpreted over infinite traces.

Leveraging automata-based techniques paired with suitable measures relating traces, events and constraints, we review three interconnected fundamental declarative process mining tasks:

Reasoning – to uncover relationships among different constraints, and check key properties of DECLARE specifications;
Discovery – to extract a DECLARE specification that suitably characterizes the traces contain in an event log;
Monitoring – to provide operational decision support [63] by checking at run-time whether a running process execution satisfies a DECLARE specification, promptly detecting and reporting violations.

All the presented techniques are integrated in the MINERful process discovery technique[1] [40] and the RuM toolkit[2] [4].

The chapter is organized as follows. Section 2 introduces the declarative process specification language DECLARE alongside a running example to which we will refer throughout the remainder of the chapter. Section 3 provides the fundamental notions upon which the core techniques for reasoning, discovery and monitoring on declarative specifications are based. We define the formal semantics of DECLARE and discuss the core reasoning tasks for declarative specifications in Sect. 4. Section 5 explains the core notions of declarative process discovery and monitoring. Section 6 discusses the latest advances in the field of declarative process specification mining. Finally, Sect. 7 concludes this chapter with final remarks and a summary of the core concepts illustrated herein.

[1] https://github.com/cdc08x/MINERful.
[2] https://rulemining.org.

Table 1. A set of Declare constraints among those that are typically used for process mining, with their textual description, graphical notation, and examples fulfilling or violating them.

Constraint	Explanation	Examples				Notation
Existence constraints						
Init(a)	a is the *first* to occur	✓⟨a, c, c⟩	✓⟨a, b, a, c⟩	×⟨c, c⟩	×⟨b, a, c⟩	Init a
AtLeastOne(a)	a occurs at least *once*	✓⟨b, c, a, c⟩	✓⟨b, c, a, a, c⟩	×⟨b, c, c⟩	×⟨c⟩	1..* a
AtMostOne(a)	a occurs at most *once*	✓⟨b, c, c⟩	✓⟨b, c, a, c⟩	×⟨b, c, a, a, c⟩	×⟨b, c, a, c, a, a⟩	0..1 a
End(a)	a is the *last* to occur	✓⟨b, c, a⟩	✓⟨b, a, c, a⟩	×⟨b, c⟩	×⟨b, a, c⟩	End a
Relation constraints						
RespondedExistence(a, b)	If a occurs in the trace, then b occurs as well	✓⟨b, c, a, a, c⟩	✓⟨b, c, c⟩	×⟨c, a, a, c⟩	×⟨a, c, c⟩	a b
Response(a, b)	If a occurs, then b occurs after a	✓⟨c, a, a, c, b⟩	✓⟨b, c, c⟩	×⟨c, a, a, c⟩	×⟨b, a, c, c⟩	a b
AlternateResponse(a, b)	Each time a occurs, then b occurs afterwards, and no other a recurs in between	✓⟨c, a, c, b⟩	✓⟨a, b, c, a, c, b⟩	×⟨c, a, a, c, b⟩	×⟨b, a, c, a, c, b⟩	a b
ChainResponse(a, b)	Each time a occurs, then b occurs immediately afterwards	✓⟨c, a, b, b⟩	✓⟨a, b, c, a, b⟩	×⟨c, a, c, b⟩	×⟨b, c, a⟩	a b
Precedence(a, b)	b occurs only if preceded by a	✓⟨c, a, c, b, b⟩	✓⟨a, c, c⟩	×⟨c, c, b, b⟩	×⟨b, a, c, c⟩	a b
AlternatePrecedence(a, b)	Each time b occurs, it is preceded by a and no other b can recur in between	✓⟨c, a, c, b, a⟩	✓⟨a, b, c, a, a, c, b⟩	×⟨c, a, c, b, b, a⟩	×⟨a, b, b, a, b, c, b⟩	a b
ChainPrecedence(a, b)	Each time b occurs, then a occurs immediately beforehand	✓⟨a, b, c, a⟩	✓⟨a, b, a, a, b, c⟩	×⟨b, c, a⟩	×⟨b, a, a, c, b⟩	a b
Mutual relation constraints						
CoExistence(a, b)	If b occurs, then a occurs, and vice versa	✓⟨c, a, c, b, b⟩	✓⟨b, c, c, a⟩	×⟨c, a, c⟩	×⟨b, c, c⟩	a b
Succession(a, b)	a occurs if and only if it is followed by b	✓⟨c, a, c, b, b⟩	✓⟨a, c, c, b⟩	×⟨b, a, c⟩	×⟨b, c, c, a⟩	a b
AlternateSuccession(a, b)	a and b if and only if the latter follows the former, and they alternate each other in the trace	✓⟨c, a, c, b, a, b⟩	✓⟨a, b, c, a, b, c⟩	×⟨c, a, a, c, b, b⟩	×⟨b, a, c⟩	a b
ChainSuccession(a, b)	a and b occur if and only if the latter immediately follows the former	✓⟨c, a, b, a, b⟩	✓⟨c, c, c⟩	×⟨c, a, c, b⟩	×⟨c, b, a, c⟩	a b
Negative relation constraints						
NotCoExistence(a, b)	a and b never occur together	✓⟨c, c, c, b, b, b⟩	✓⟨c, c, a, c⟩	×⟨a, c, c, b, b⟩	×⟨b, c, a, c⟩	a b
NotSuccession(a, b)	b cannot occur after a	✓⟨b, b, c, a, a⟩	✓⟨c, b, b, c, a⟩	×⟨a, a, c, b, b⟩	×⟨a, b, b⟩	a b
NotChainSuccession(a, b)	a and b cannot occur contiguously	✓⟨a, c, b, a, c, b⟩	✓⟨b, b, a, a⟩	×⟨a, b, c, a, b⟩	×⟨c, a, b, c⟩	a b

2 Declare: A Gentle Introduction

Declare is a language and graphical notation providing an extendible repertoire of templates to formulate constraints. The origin of the approach traces back to the PhD work by Pesic [75], and the parallel and consequent study in the PhD work by Montali [67]. Notably, Declare actually stems from three initial lines of research, respectively focused on the declarative specification of business processes (cf. the ConDec language [78]), service choreographies (cf. the DecSerFlow language [70,94]), and clinical guidelines (cf. the CigDec language [72]). These lines were then unified into a single research thread. The term Declare was used for the first time in [76].

Table 1 shows a set of Declare constraints we use throughout this chapter. The whole, core set of Declare templates has been inspired by a catalogue of temporal logic patterns used in model checking for a variety of dynamic systems from different application domains [41].

Formally, we define a declarative process specification as follows.

Definition 1 (Declarative process specification). *A* declarative process specification *is a tuple* $\mathrm{DS} = (\textsc{Rep}, \mathrm{Act}, K)$ *where*

- $\textsc{Rep}$ *is a finite non-empty set of* templates*, where each template is a predicate* $\kappa(x_1, \ldots, x_m) \in \textsc{Rep}$ *on variables* $x_1, \ldots, x_m$ *(with* $m \in \mathbb{N}$ *the* arity *of* κ*),*
- Act *is a finite non-empty set of* activities*,*
- K *is a finite set of* constraints*, namely pairs* $(\kappa(x_1, \ldots, x_m), \kappa)$ *where* $\kappa(x_1, \ldots, x_m)$ *is a template from* $\textsc{Rep}$*, and* κ *is a mapping that, for every* $i \in \{1, \ldots, m\}$ *assigns variable* x_i *with an activity* $\kappa(x_i) = a_i \in \mathrm{Act}$*; we compactly denote such a constraint with* $\kappa(a_1, \ldots, a_m)$*.* ◁

Example 1 (A Declare process specification). Figure 2 portrays an example of declarative specification for the admission process of an international Bachelor's program. This example considers the Declare repertoire of templates. The process begins with the creation of an account in the university portal (henceforth, c). To specify that c is the initial task, we write Init(c), graphically depicted with the Init label in the tag on top of the activity box. Init is a unary template and Init(c) assigns its variable with activity c. Unary templates in Declare are also known as *existence* templates. We indicate that not more than one account can be created per process run with AtMostOne(c). In the diagram, it is indicated with the 0..1 label in the tag.

To register for a selection round (r), an account must have been created before (Precedence(c, r)). Precedence is a binary template and Precedence(c, r), graphically depicted as c ──▶● r, assigns c and r to its first and second variable, respectively. Binary templates in Declare are commonly named as *relation* templates.

Every registration to a selection round (r) gives access to a uniquely corresponding evaluation phase (v). After r, v eventually follows and no other registrations are allowed until v completes. We write AlternateResponse(r, v), graphically depicted as r ●═══▶ v. The evaluation requires r to

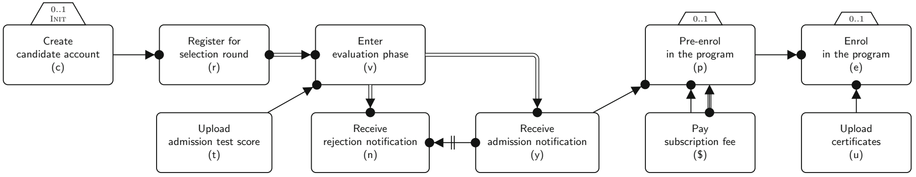


Fig. 2. The Declare map of the admission process at a university.

be completed before and v will not recur unless a new registration is issued: AlternatePrecedence(r, v), . Typically, if both AlternateResponse(r, v) and AlternatePrecedence(r, v) hold true, we compactly represent them jointly with the *mutual* relation constraint AlternateSuccession(r, v) . An admission test score has to be uploaded in the platform to access the evaluation phase: Precedence(t, v). Evaluation phases are necessary for the committee to return rejections (n) and notifications of admission (y), thus AlternatePrecedence(v, y) and AlternatePrecedence(v, n) hold.

After the admission has been notified, the candidate will not receive a rejection any longer – NotResponse(y, n), drawn in Fig. 2 as . NotResponse(y, n) falls under the category of the *negative* relation constraints, as the occurrence of y *disables* n in the remainder of the process execution.

Only if candidates receive a notification of admission, they will be entitled to pre-enrol in the program (Precedence(y, p)). The candidates are considered as pre-enrolled immediately after they pay the subscription fee (ChainResponse($, p), shown as follows in the diagram:). Also, candidates cannot be considered as pre-enrolled if they have not paid the subscription fee: Precedence($, p). Not more than one pre-enrolment is allowed per candidate: AtMostOne(p). To enrol in the program (e), the candidate must have pre-enrolled – Precedence(p, e) – and uploaded the necessary school and language certificates – Precedence(u, e).

So far, we have been attaching an informal semantics to Declare and its templates. In the next section, we provide a more systematic and formal characterization.

3 Formal Background

Considering that Declare templates have been originally defined starting from a catalogue of Linear Temporal Logic (LTL) patterns [41], it is not surprising that temporal logics have been used to characterize the semantics of Declare since the very beginning. However, the fact that Declare specifications are interpreted over finite-length executions calls for the use of Linear Temporal Logic on Finite Traces (LTL_f) [30]. This indeed leads to a setting that is radically

different, both semantically and algorithmically, from the traditional one where formulae are interpreted using LTL over infinite, recurring behaviors [29].

A complete formalization of DECLARE templates, also including an alternative formalization using a logic programming-based approach, can be found in [68]. It was later refined in [29]. In his PhD thesis, Di Ciccio was the first to provide a semantics based on regular expressions [36]. These two themes were later unified in [28], leading to a richer framework that is able to declaratively capture constraints and metaconstraints, that is, constraints predicating over the possible/certain satisfaction and violation of other constraints.

In this section, we provide some necessary background on LTL_f and its extension with past-tense temporal operators, as well as on the automata-theoretic characterization for this logic. We then use this framework to formalize DECLARE and reason automatically on DECLARE specifications. Thereupon, we reflect upon the most recent advances of research in attempting at capturing not only the formal semantics of constraints, but also how they pragmatically interact with relevant events.

3.1 Linear Temporal Logic on Finite Traces

LTL_f has the same syntax of LTL [80], but is interpreted on finite traces. In this chapter, in particular, we consider the LTL dialect including past modalities [56] for declarative process specifications as in [18].

From now on, we fix a finite set Σ representing an alphabet of propositional symbols describing (names of) activities available in the domain under study. A (finite) *trace* $t = \langle a_1, \ldots, a_n \rangle \in \Sigma$ of length $|t| = n$ is a finite sequence of activities, where the presence of activity a_i at instant i of the trace represents an *event* that witnesses the occurrence of a_i at instant i – which we also write $t(i) = a_i$. Notice that *at each instant we assume that one and only one activity occurs.* Using standard notation from regular expressions, the set Σ^* denotes the overall set of traces whose constitutive events refer to activities in Σ.

Definition 2 (Syntax of LTL_f). *Well-formed formulae are built from Σ, the unary temporal operators $\bigcirc$ (next) and $\ominus$ (yesterday), and the binary temporal operators $\mathbf{U}$ (until) and $\mathbf{S}$ (since) as follows:*

$$\varphi ::= \mathsf{a} \mid (\neg\varphi) \mid (\varphi_1 \wedge \varphi_2) \mid (\bigcirc \varphi) \mid (\varphi_1 \ \mathbf{U} \ \varphi_2) \mid (\ominus \varphi) \mid (\varphi_1 \ \mathbf{S} \ \varphi_2)$$

where $\mathsf{a} \in \Sigma$. ◁

Definition 3 (Semantics of LTL_f, satisfaction, validity, entailment). *An LTL_f formula φ is inductively* satisfied *in some instant i $(1 \leq i \leq n)$ of a trace t of length $n \in \mathbb{N}$, written $t, i \vDash \varphi$, if the following holds:*

- $t, i \vDash \mathsf{a}$ *iff $t(i)$ is assigned with* a;
- $t, i \vDash \neg\varphi$ *iff* $t, i \nvDash \varphi$;
- $t, i \vDash \varphi_1 \wedge \varphi_2$ *iff* $t, i \vDash \varphi_1$ *and* $t, i \vDash \varphi_2$;
- $t, i \vDash \bigcirc \varphi$ *iff* $i < n$ *and* $t, i+1 \vDash \varphi$;

- $t, i \models \ominus \varphi$ *iff* $i > 1$ *and* $t, i-1 \models \varphi$*;*
- $t, i \models \varphi_1 \ \mathbf{U} \ \varphi_2$ *iff* $t, j \models \varphi_2$ *with* $i \leq j \leq n$*, and* $t, k \models \varphi_1$ *for all* k *s.t.* $i \leq k < j$*;*
- $t, i \models \varphi_1 \ \mathbf{S} \ \varphi_2$ *iff* $t, j \models \varphi_2$ *with* $1 \leq j \leq i$*, and* $t, k \models \varphi_1$ *for all* k *s.t.* $j < k \leq i$*.*

A formula φ is satisfied by *a trace* t *(equivalently,* t satisfies φ*), written* $t \models \varphi$*, iff* $t, 1 \models \varphi$*. A formula* φ *is: (i)* satisfiable *if it has a satisfying trace from* Σ^**; (ii)* valid *if every trace in* Σ^* *satisfies it. A formula* φ_1 entails *formula* φ_2*, written* $\varphi_1 \models \varphi_2$*, if, for every trace* t *of length* $n \in \mathbb{N}$ *and every* i *s.t.* $1 \leq i \leq n$*, if* $t, i \models \varphi$ *then* $t, i \models \psi$*.* $\triangleleft$

Since LTL_f is closed under negation, it is easy to see that a formula φ is valid if and only if $\neg\varphi$ is unsatisfiable.

It is worth noting that, in LTL_f, the next operator is interpreted as the so-called *strong* next: $\bigcirc \varphi$ requires that the next instant exists within the trace, and that at such next instant φ holds. This has an important consequence: differently from LTL, in LTL_f formula $\neg \bigcirc \varphi$ is *not* equivalent to $\bigcirc \neg\varphi$. This is because $\neg \bigcirc \varphi$ is true in an instant of a finite trace either when that instant has no successor, or the next instant exists and in such a next instant φ does not hold. More on this can be found in [29].

From the basic operators above, the following can be derived:

- Classical boolean abbreviations $\mathbf{true}, \mathbf{false}, \vee, \rightarrow$;
- Constant $\mathbf{end} \equiv \neg \bigcirc \mathbf{true}$, denoting the last instant of a trace;
- Constant $\mathbf{start} \equiv \neg \ominus \mathbf{true}$, denoting the first instant of a trace;
- $\Diamond \varphi \equiv \mathbf{true} \ \mathbf{U} \ \varphi$ indicating that φ eventually holds true in the trace (hence, before or at **end**);
- $\varphi_1 \ \mathbf{W} \ \varphi_2 \equiv (\varphi_1 \ \mathbf{U} \ \varphi_2) \vee \Box \varphi_1$, which relaxes $\mathbf{U}$ as φ_2 may never hold true;
- $\diamondminus \varphi \equiv \mathbf{true} \ \mathbf{S} \ \varphi$ indicating that φ holds true at some instant before the current one (i.e., after **start** in the trace);
- $\Box \varphi \equiv \neg \Diamond \neg\varphi$ indicating that φ holds true from the current instant till **end**;
- $\boxminus \varphi \equiv \neg \diamondminus \neg\varphi$ indicating that φ holds true from **start** to the current instant.

Example 2. Let $t = \langle a, b, b, c, d, e \rangle$ be a trace and φ_1, φ_2 and φ_3 three LTL_f formulae defined as follows: $\varphi_1 \doteq d$; $\varphi_2 \doteq \Diamond b$; $\varphi_3 \doteq \Box(b \rightarrow \Diamond d)$. We have that $t, 1 \not\models \varphi_1$ whereas $t, 5 \models \varphi_1$; $t, 1 \models \varphi_2$ whereas $t, 5 \not\models \varphi_2$; $t, 1 \models \varphi_3$ and $t, 5 \models \varphi_3$ (in fact, $t, i \models \varphi_3$ for any instant $1 \leq i \leq n$). $\triangleleft$

3.2 Finite-State Automata

One of the central features of LTL_f is that a finite state automaton (FSA) [22] $\mathscr{A}(\varphi)$ can be computed such that for every trace t we have that $t \models \varphi$ iff t is in the language recognized by $\mathscr{A}(\varphi)$, as illustrated in [18,28,30,38]. We include the main notions next, recalling that focusing on deterministic FSAs is without loss of generality, as over finite traces every non-deterministic FSAs can be determinized [50].

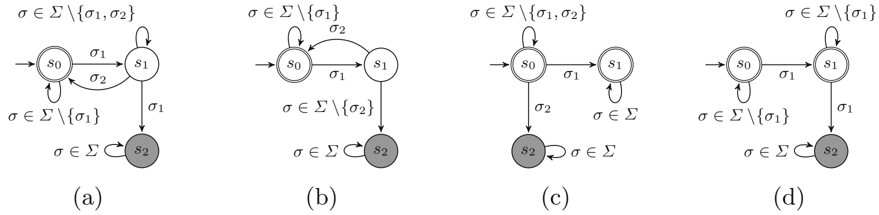


Fig. 3. Examples of constraint FSAs.

Definition 4 (Finite state automaton (FSA)). *A (deterministic) finite state automaton (FSA) is a tuple $A = (\Sigma, S, \delta, s_0, S_F)$, where:*

- *Σ is a finite set of symbols;*
- *S is a finite non-empty set of states;*
- *$\delta : S \times \Sigma \rightarrow S$ is the* transition function, *i.e., a partial function that, given a starting state and a (labeled) transition, returns the target state;*
- *s_0 is the initial state;*
- *$S_F \subseteq S$ is the set of final (accepting) states $s_F \in S_F$*

◁

In the remainder of the chapter, we assume that δ is left-total and surjective on $S \setminus \{s_0\}$, that is, the transition function is defined for every state and symbol, and every state is on a path from the initial one – with the possible exception of the initial state itself. An FSAs that is left-total is called *untrimmed*. Notice that these two requirements are without loss of generality: every FSA can be converted into an equivalent FSA that is left-total and surjective. In particular, to make an FSAs untrimmed, it is sufficient to: *(i)* introduce a non-final trap state $s_\perp$; *(ii)* for every state s and symbol a' such that $\delta(s, a')$ is *not* defined, enforce $\delta(s, a') = s_\perp$; *(iii)* connect $s_\perp$ to itself for every symbol, setting $\delta(s_\perp, a) = s_\perp$ for every $a \in \Sigma$.

Example 3. Figure 3 depicts four FSAs. States are represented as circles and transitions as arrows. Accepting states are decorated with a double line. The initial state is indicated with a single, unlabeled incoming arc. For instance, Fig. 3(a) is such that $\Sigma \supseteq \{\sigma_1, \sigma_2\}$, $S = \{s_0, s_1, s_2\}$, $S_\text{F} = \{s_0\}$, $\delta(s_0, \sigma_1) = s_1$ and $\delta(s_1, \sigma_1) = s_2$. ◁

Definition 5 (Runs and traces of an FSA). *Let $A = (\Sigma, S, \delta, s_0, S_F)$ be an FSA as per Definition 4. A* computation *π of A is a finite sequence alternating states and activities $s_0 \xrightarrow{\sigma_0} \ldots \xrightarrow{\sigma_{n-1}} s_n$ that starts from the initial state s_0 is such that for every $0 \leq i < n$, we have $\delta(s_i, \sigma_i) = s_{i+1}$. If π terminates in a final state, that is, $s_n \in S_F$, then it is a* run, *and* induces *a corresponding trace $\sigma_0, \ldots, \sigma_{n-1}$ over Σ^* obtained from π by only keeping the symbols that label the transitions.* ◁

Example 4. In Fig. 3(a), $\pi_1 = s_0 \xrightarrow{\sigma_1} s_1$, $\pi_2 = s_0 \xrightarrow{\sigma_2} s_0 \xrightarrow{\sigma_1} s_1 \xrightarrow{\sigma_1} s_2$, and $\pi_3 = s_0 \xrightarrow{\sigma_1} s_1 \xrightarrow{\sigma_2} s_2 \xrightarrow{\sigma_1} s_0$ are three examples of computations. However, only π_3 is a run because $s_0 \in S_\mathrm{F}$ whereas $s_1, s_2 \notin S_\mathrm{F}$. Notice that, in Fig. 3, we additionally highlight with a grey background colour those states that cannot be in a step of a run – that is, from which accepting states cannot be reached (e.g., s_2 in Fig. 3(a)). ◁

Definition 6 (Accepted trace, language of an FSA). *A trace $t \in \Sigma^*$ is* accepted *by FSA $A = (\Sigma, S, \delta, s_0, s_F)$ if there is a run of A inducing t. The* language *$\mathscr{L}(A)$ of A is the set of traces accepted by A.* ◁

Example 5. For the FSA in Fig. 3(a), the language contains the trace $t_1 = \langle \sigma_1, \sigma_2, \sigma_1 \rangle$, since a run exists over this sequence of labels (i.e., π_3 above), whereas $t_2 = \langle \sigma_2, \sigma_1 \rangle$ is not part of the language. ◁

Automata Product. FSAs are closed under the (synchronous) product operation $\times$ [81]. The (cross-)product $A \times A'$ of two FSAs A and A' is an FSA that accepts the intersection of languages (sets of accepted traces) of each operand: $\mathscr{L}(A \times A') = \mathscr{L}(A) \bigcap \mathscr{L}(A')$. It is defined as follows.

Definition 7 (Automata product). *The* product FSA *of two FSAs $A = (\Sigma, S, \delta, s_0, S_F)$ and $A' = (\Sigma, S', \delta', s'_0, S'_F)$ over the same alphabet Σ is the FSA $A \times A' = (\Sigma, S^\times, \delta^\times, s_0^\times, S_F^\times)$, where the set $S^\times \subseteq S \times S'$ of states (obtained from the cartesian product of the states in A and A'), its initial state $s_0^\times$, its final states $S_F^\times$, and the transition function $\delta^\times$, are defined by simultaneous induction as follows:*

- *$s_0^\times = \langle s_0, s'_0 \rangle \in S^\times$;*
- *For every state $\langle s_1, s'_1 \rangle \in S^\times$, state $s_2 \in S$, state $s'_2 \in S'$, and label $\ell \in \Sigma$, if $\delta(s_1, \ell) = s_2$ and $\delta'(s'_1, \ell) = s'_2$ then: (i) $\langle s_2, s'_2 \rangle \in S^\times$, (ii) $\delta^\times(\langle s_1, s'_1 \rangle, \ell) = \langle s_2, s'_2 \rangle$, (iii) if $s_2 \in S_F$ and $s'_2 \in S'_F$, then $\langle s_2, s'_2 \rangle \in S_F^\times$.*
- *Nothing else is in $S_F^\times$, $S^\times$, and $\delta^\times$.*

◁

Notice that the FSA constructed with Definition 7 can be manipulated using language-preserving automata operations, such as in particular *minimization* [50].

The product operation $\times$ is commutative and associative. The identity element for $\times$ over alphabet Σ is $A^\mathrm{I} = (\Sigma, \{s_0\}, s_0, \{s_0\} \times \Sigma \times \{s_0\}, \{s_0\})$ – depicted in Fig. 4(a). It accepts all traces over Σ: $\mathscr{L}(A^\mathrm{I}) = \mathbb{P}(\Sigma^*)$ as any sequence of transitions labeled by symbols in Σ corresponds to a run for A^I. The absorbing element is $A^\emptyset = (\Sigma, \{s_0\}, s_0, \{s_0\} \times \Sigma \times \{s_0\}, \emptyset)$ and is illustrated in Fig. 4(b). It does not accept any trace at all: $\mathscr{L}(A^\emptyset) = \emptyset$ as any sequence of transitions labeled by symbols in Σ corresponds to a computation ending in a non-accepting state.

4 Reasoning

Equipped with the notions acquired thus far, we can now discuss the core reasoning tasks that are associated to declarative process specifications. To this end, we begin this section by describing the semantics of Declare in detail.

(a) Identity element (b) Absorbing element

Fig. 4. Finite state automata acting as identity element and absorbing element for the automata cross-product operation.

4.1 Semantics of Declare

The semantics of a Declare template $\text{K}(x_1, \ldots, x_m)$ is given as an LTL$_f$ formula $\varphi_{\text{K}(x_1,\ldots,x_m)}$ defined over variables $x_1, \ldots, x_m$ instead of activities. Given the free variables x and y, e.g., Response(x, y) corresponds to $\Box(x \rightarrow \Diamond y)$, witnessing that whenever x occurs, then y is expected to occur at some later instant. Table 2 shows the LTL$_f$ formulae of some templates of the Declare repertoire. The formalization of a constraint is then obtained by grounding the LTL$_f$ formula of its template.

Definition 8 (Constraint formula, satisfying trace). *The* formula *of constraint* $\text{K}(a_1, \ldots, a_m)$, *written* $\varphi_{\text{K}(a_1,\ldots,a_m)}$, *is the* LTL$_f$ *formula obtained from* $\varphi_{\text{K}(x_1,\ldots,x_m)}$ *by replacing* x_i *with* a_i *for each* $1 \leq i \leq m$. *A trace* t satisfies $\text{K}(a_1, \ldots, a_m)$ *if* $t \models \varphi_{\text{K}(a_1,\ldots,a_m)}$; *otherwise, we say that* t violates $\text{K}(a_1, \ldots, a_m)$. ◁

Example 6. Considering Table 2, we have $\varphi_{\text{Response}(\mathsf{a},\mathsf{b})} = \Box(\mathsf{a} \rightarrow \Diamond \mathsf{b})$, and $\varphi_{\text{Response}(\mathsf{b},\mathsf{c})} = \Box(\mathsf{b} \rightarrow \Diamond \mathsf{c})$. Traces $\langle \mathsf{b} \rangle$ and $\langle \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{a}, \mathsf{c}, \mathsf{b} \rangle$ satisfy Response(a, b), while $\langle \mathsf{a} \rangle$ and $\langle \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{a}, \mathsf{c} \rangle$ do not. ◁

A Declare specification is then formalized by conjoining all its constraint formulae, thus obtaining a direct, declarative notion of *model trace*, that is, a trace that is accepted by the specification.

Definition 9 (Specification formula, model trace). *The* formula *of* Declare *specification* $\text{DS} = (\text{Rep}, \text{Act}, K)$, *written* φ_{DS}, *is the* LTL$_f$ *formula* $\bigwedge_{\text{K} \in K} \varphi_{\text{K}}$. *A trace* $t \in \text{Act}^*$ *is a* model trace *of* DS *if* $t \models \varphi_{\text{DS}}$; *in this case, we say that* t *is* accepted by DS, *otherwise that* t *is* rejected by DS. ◁

Constructing constraint and specification formulae is, however, not enough. When one reads $\Box(\mathsf{a} \rightarrow \Diamond \mathsf{b})$ following the textual description given above, the formula gets intepreted as "whenever a occurs, then b is expected to occur at some later instant". This formulation intuitively hints at the fact that the occurrence of a *activates* the Response(a, b) constraint, requiring the *target* b to occur. In turn, we get that a trace not containing any occurrence of a is *less interesting* than a trace containing occurrences of a, each followed by one or more occurrences of b: even though both traces satisfy Response(a, b), the first trace never "interacts"

Table 2. Semantics of some DECLARE constraints.

Template	**LTL$_f$expression [18,30]**	**Activation**	**Target**
Existence constraints			
ATLEASTONE(x)	$\Box(\mathbf{start} \rightarrow \Diamond x)$	**start**	$\Diamond x$
ATMOSTONE(x)	$\Box(x \rightarrow \neg \bigcirc \Diamond x)$	x	$\neg \bigcirc \Diamond x$
INIT(x)	$\Box(\mathbf{start} \rightarrow x)$	**start**	x
END(x)	$\Box(\mathbf{end} \rightarrow x)$	**end**	x
Relation constraints			
RESPONDEDEXISTENCE(x, y)	$\Box(x \rightarrow \Diamond y \vee \diamondminus y)$	x	$\Diamond y \vee \diamondminus y$
RESPONSE(x, y)	$\Box(x \rightarrow \Diamond y)$	x	$\Diamond y$
ALTERNATERESPONSE(x, y)	$\Box(x \rightarrow \bigcirc(\neg x \;\mathbf{U}\; y))$	x	$\bigcirc(\neg x \;\mathbf{U}\; y)$
CHAINRESPONSE(x, y)	$\Box(x \rightarrow \bigcirc y)$	x	$\bigcirc y$
PRECEDENCE(x, y)	$\Box(y \rightarrow \diamondminus x)$	y	$\diamondminus x$
ALTERNATEPRECEDENCE(x, y)	$\Box(y \rightarrow \ominus(\neg y \;\mathbf{S}\; x))$	y	$\ominus(\neg y \;\mathbf{S}\; x)$
CHAINPRECEDENCE(x, y)	$\Box(y \rightarrow \ominus x)$	y	$\ominus x$
Negative relation constraints			
NOTRESPONDEDEXISTENCE(x, y)	$\Box(x \rightarrow (\Box \neg y \wedge \boxminus \neg y))$	x	$\Box \neg y \wedge \boxminus \neg y$
NOTRESPONSE(x, y)	$\Box(x \rightarrow \Box \neg y)$	x	$\Box \neg y$
NOTCHAINRESPONSE(x, y)	$\Box(x \rightarrow \neg \bigcirc y)$	x	$\neg \bigcirc y$
NOTPRECEDENCE(x, y)	$\Box(y \rightarrow \boxminus \neg x)$	y	$\boxminus \neg x$
NOTCHAINPRECEDENCE(y, x)	$\Box(y \rightarrow \neg \ominus x)$	y	$\neg \ominus x$

with RESPONSE(a, b), while the second does. This relates to the notion of vacuous satisfaction in LTL [51] and that of interestingness of satisfaction in LTL$_f$ [39].

The point is, all such considerations are not captured by the formula $\Box(\mathsf{a} \rightarrow \Diamond \mathsf{b})$, but are related to pragmatic interpretation of how it relates to traces. To see this aspect, let us consider that we can equivalently express the formula above as $\Box \neg \mathsf{a} \vee \Diamond(\mathsf{b} \wedge \Box \neg \mathsf{a})$, which now reads as follows: "Either a never happens at all, or there is some occurrence of b after which a never happens". This equivalent reformulation does not put into evidence the activation or the target.

This problem can be tackled in two possible ways. One option is to attempt at an automated approach where activation, target, and interesting satisfaction are semantically, implicitly characterized once and for all at the logical level; this is the route followed in [39]. The main drawback of this approach is that the user cannot intervene at all in deciding how to fine-tune the activation and target conditions. An alternative possibility is instead to ask the user to explicitly indicate, together with the LTL$_f$ formula φ of the template, also two related LTL$_f$ formulae expressing activation and target conditions for φ. This latter approach, implicitly adopted in [69] and then explicitly formalized in [18], gives more control to the user on how to *pragmatically* interpret constraints. We follow this latter approach.

Intuitively, the *activation* of a constraint is a triggering condition that, once made true, expects that the *target* condition is satisfied by the process execution.

Contrariwise, if the constraint is not activated, the satisfaction of the target is not enforced. All in all, to properly constitute an activation-target pair for an LTL_f formula φ, we need them to satisfy the condition that whenever the current instant is such that the activation is satisfied, φ must behave equivalently to the target (thus requiring its satisfaction). This is formally captured as follows.

Definition 10 (Activation and target of a constraint). *The* activation *and* target *of a constraint* K *over activities* Act *are two* LTL_f *formulae* ${}_{\square}\mathrm{K}$ *and* $\mathrm{K}_{\blacktriangleright}$ *such that for every trace* $t \in \mathrm{Act}^*$ *we have that:*

$$t \models \varphi_{\mathrm{K}} \quad \textit{iff} \quad t \models \Box\left({}_{\square}\mathrm{K} \rightarrow (\mathrm{K}_{\blacktriangleright})\right)$$

Table 2 shows activations and targets for each constraint, inspired by the work of Cecconi et al. [18]. In the next example, we explain the rationale behind some of the constraint formulations in the table.

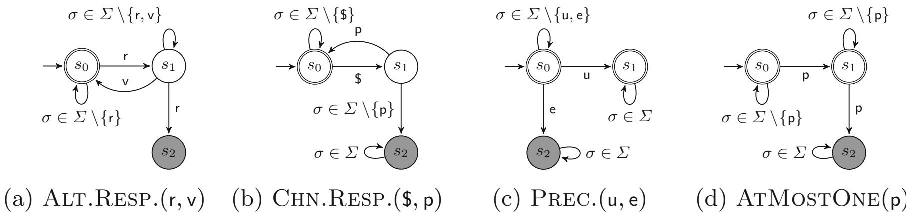
Example 7. Consider $\textsc{ChainResponse}(\$, \mathsf{p})$, dictating that whenever $\$$ occurs, then p is the activity occurring next. We have $\varphi_{\textsc{ChainResponse}(\$,\mathsf{p})} = \Box(\$ \rightarrow \bigcirc\, \mathsf{p})$. Then, by Definition 10, we can directly fix ${}_{\square}\textsc{ChainResponse}(\$, \mathsf{p}) = \$$, and $\textsc{ChainResponse}(\$, \mathsf{p})_{\blacktriangleright} = \bigcirc\, \mathsf{p}$, respectively witnessing that every occurrence of $\$$ triggers the constraint, with a target requiring the consequent execution of p in the next instant. Similarly, for $\textsc{Precedence}(\$, \mathsf{p})$ we have $\varphi_{\textsc{Precedence}(\$,\mathsf{p})} = \Box\,(\mathsf{p} \rightarrow \diamondminus\, \$)$, and in turn, by Definition 10, ${}_{\square}\varphi_{\textsc{Precedence}(\$,\mathsf{p})} = \mathsf{p}$ and $\varphi_{\textsc{Precedence}(\$,\mathsf{p})\blacktriangleright} = \diamondminus\, \$$. The case of $\textsc{AtMostOne}(\mathsf{p})$ is also similar. In this case, $\varphi_{\textsc{AtMostOne}(\mathsf{p})}$ formalizes that p cannot occur twice, which in LTL_f can be directly captured by $\neg\Diamond(\mathsf{p} \wedge \bigcirc\Diamond\, \mathsf{p})$. This is logically equivalent to $\Box(\mathsf{p} \rightarrow \neg\bigcirc\Diamond\, \mathsf{p})$, which directly yields ${}_{\square}\textsc{AtMostOne}(\mathsf{p}) = \mathsf{p}$ and $\textsc{AtMostOne}(\mathsf{p})_{\blacktriangleright} = \neg\bigcirc\Diamond\, \mathsf{p}$.

A quite different situation holds instead for the other *existence* constraints. Take, for example, $\textsc{AtLeastOne}(\mathsf{a})$, requiring that a occurs at least once in the execution. This can be directly encoded in LTL_f as $\Diamond\, \mathsf{a}$. This formulation, however, does not help to individuate the activation and target of the constraint. Intuitively, we may disambiguate this by capturing that since the constraint requires the presence of a from the very beginning of the execution, the constraint is indeed activated at the beginning, i.e., when **start** holds, imposing the satisfaction of the target $\Diamond\, \mathsf{a}$. This intuition is backed up by Definition 10, using the semantics of **start** and noticing the following logical equivalences:

$$\Diamond\, \mathsf{a} = \mathbf{start} \rightarrow \Diamond\, \mathsf{a} = \Box\,(\mathbf{start} \rightarrow \Diamond\, \mathsf{a})$$

This explains why the latter formulation is employed in Table 2. ◁

Declarative Constraints as FSAs. Crucial for our techniques is that every LTL_f formula φ can be encoded into a corresponding FSA (in the sense of Definition 4) A_φ that recognizes all and only those traces that satisfy the formula. This can be done through different algorithmic techniques. A direct approach



(a) ALT.RESP.(r, v) (b) CHN.RESP.(\$, p) (c) PREC.(u, e) (d) ATMOSTONE(p)

Fig. 5. Example FSAs of DECLARE constraints.

that transforms an input formula into a non-deterministic FSAs is presented in [28,29]; notice that the so-obtained FSAs can then be determinized and minimized using standard techniques [50,99]. A fortiori, given a DECLARE specification $\text{DS} = (\text{REP}, \text{Act}, K)$, we proceed as follows:

- We pair each constraint $\kappa \in K$ to a corresponding, so-called *local automaton* A_κ. This automaton is the FSA A_{φ_κ} of the constraint formula φ_κ, and is used to characterize all and only those traces that satisfy κ;
- We pair the whole specification to a so-called *global automaton* A_{DS}, that is, the FSA $A_{\varphi_{\text{DS}}}$ of the constraint formula φ_{DS}. It thus recognizes all and only the model traces of DS. Recall that, as introduced in Definition 9, φ_{DS} is the conjunction of the formulae of the constraints in K, and thus the language $\mathscr{L}(A_{\text{DS}})$ corresponds to $\bigcap_{\kappa \in K} \mathscr{L}(A_\kappa)$. By definition of automata product, this means that $\mathscr{L}(A_{\text{DS}})$ can be obtained by computing the product of the local automata of the constraints in K.

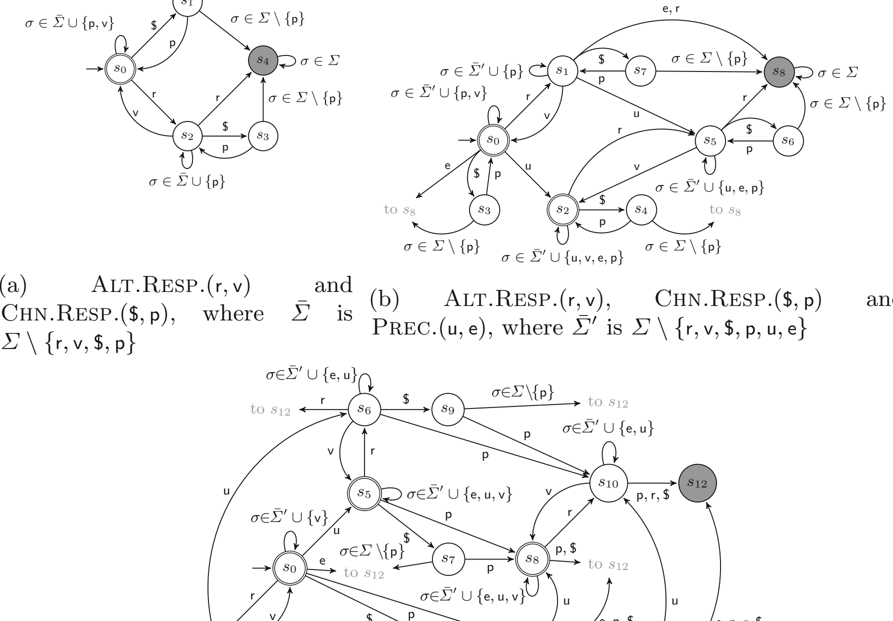
Figure 5 shows four local automata for constraints taken from our running example: ALTERNATERESPONSE(r, v), CHAINRESPONSE(\$, p), PRECEDENCE(u, e) and ATMOSTONE(p). Examples of global automata are instead given in Fig. 6.

In the remainder of this chapter, we will extensively use local and global automata for reasoning, discovery, and monitoring. Though out of scope for this chapter, it is also worth mentioning that the automata-based approach has also been used for simulation of DECLARE models and thereby the production of event logs from declarative specifications [37], and also to define enactment engines for DECLARE specifications [76,97].

4.2 Reasoning on DECLARE Specifications

Reasoning on a DECLARE specification is necessary to understand which model traces are supported and, in turn, to ascertain its correctness. Reasoning is also key to unveil how constraints interact with each other, and check whether activations and targets are properly defined. As we will see, this is instrumental not only to analyze specifications, but it is also an integral part of declarative process mining.

In general, reasoning on declarative specifications is of particular importance: while they enjoy flexibility, they typically do not explicitly indicate how execu-



(a) Alt.Resp.(r, v) and Chn.Resp.($\mathsf{\$}, \mathsf{p}$), where $\bar{\Sigma}$ is $\Sigma \setminus \{\mathsf{r}, \mathsf{v}, \mathsf{\$}, \mathsf{p}\}$

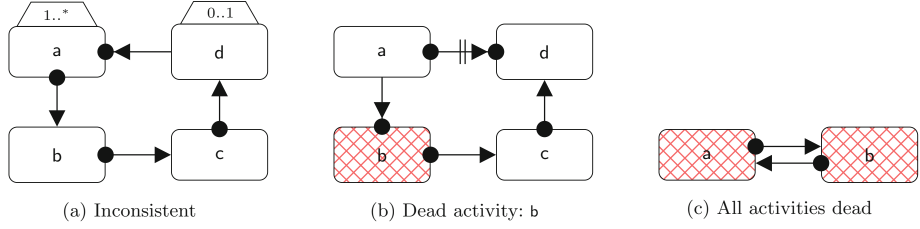
(b) Alt.Resp.(r, v), Chn.Resp.($\mathsf{\$}, \mathsf{p}$) and Prec.($\mathsf{u}, \mathsf{e}$), where $\bar{\Sigma}'$ is $\Sigma \setminus \{\mathsf{r}, \mathsf{v}, \mathsf{\$}, \mathsf{p}, \mathsf{u}, \mathsf{e}\}$

(c) Alt.Resp.(r, v), Chn.Resp.($\mathsf{\$}, \mathsf{p}$), Prec.($\mathsf{u}, \mathsf{e}$) and AtMostOne($\mathsf{p}$), where $\bar{\Sigma}'$ is $\Sigma \setminus \{\mathsf{r}, \mathsf{v}, \mathsf{\$}, \mathsf{p}, \mathsf{u}, \mathsf{e}\}$ (for the sake of readability, a few transitions to s_{12} are omitted)-

Fig. 6. Global automata for the interplay of Declare constraints.

tion has to be controlled. We have seen how this phenomenon concretely manifests itself in the context of Declare: traces conforming to the specification (that is, *model traces*) are only implicitly described as those that satisfy all the given constraints. Constraints, in turn, may be quite diverse from each other (e.g., indicating what *is expected* to occur, but also what should *not* happen) and, even more importantly, may affect each other in subtle, difficult to detect ways. This phenomenon is known, in the literature that studies the cognitive impact of languages and notations, under the name of *hidden dependencies* [47]. Hidden dependencies in Declare have been studied in [32,70], and their impact on understandability and interpretability of declarative process models has spawned a dedicated line of research, started in [48].

We detail next key reasoning tasks in the context of Declare, substantiating how hidden dependencies enter into the picture. We show that all such reasoning tasks can be homogeneously tackled by a single check on the global automaton of the specification under study.



(a) Inconsistent

(b) Dead activity: b

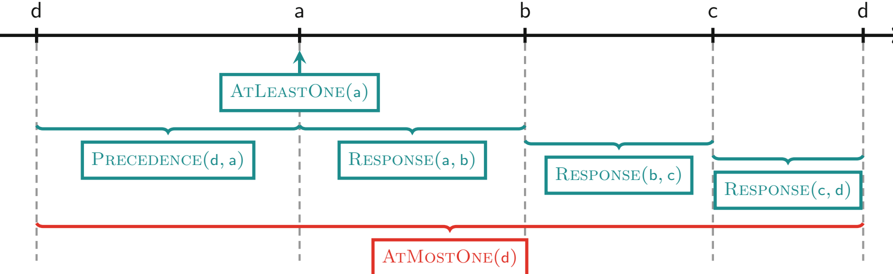
(c) All activities dead

Fig. 7. Examples of incorrect DECLARE specifications.

Specification Consistency. This is the most fundamental task, defined as follows.

Definition 11 (Consistent specification). *A* DECLARE *specification* DS *is* consistent *if there exists at least one model trace for* DS*.* ◁

Example 8. Consider the DECLARE specification in Fig. 7(a). The specification is inconsistent. This is not due to conflicting constraints insisting on the same activity, but due to hidden dependencies arising from the interplay of multiple constraints. To see why the specification is inconsistent, we can try to construct a trace that satisfies some of the constraints in the model, until we reach a contradiction (i.e., the "trace pattern" constructed so far violates a constraint of the specification). This is graphically shown next:



The picture clearly depicts that ATLEASTONE(a) triggers:

- on the one hand PRECEDENCE(d, a), calling for a preceding occurrence of d;
- on the other hand, in cascade, RESPONSE(a, b), RESPONSE(b, c), and RESPONSE(c, d), calling for a later occurrence of d.

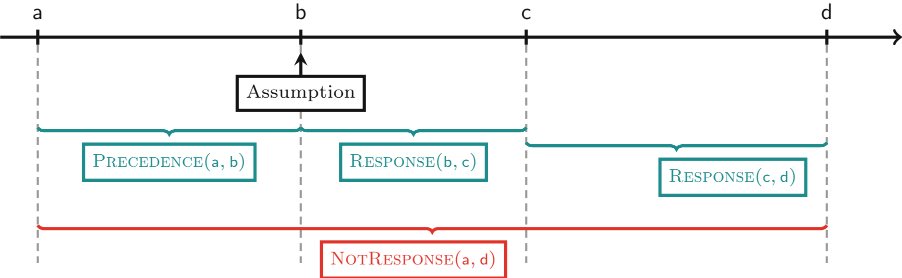
Considering the interplay of the involved constraints, d is required to occur in different instants, hence twice, in turn violating ATMOSTONE(d). ◁

By definition of model trace, it is immediate to see that DS *is consistent if and only if the* LTL$_f$ *specification formula* φ_{DS} *is satisfiable.* This, in turn, can be algorithmically verified by first constructing the global automaton A_{DS}, and then checking whether such an automaton is empty (i.e., it does not recognize any trace). Specifically, φ_{DS} *is* satisfiable *if and only if* A_{DS} *is non-empty.*

Detection of Dead Activities. This task amounts to check whether a DECLARE specification is over-constrained, in the sense that it contains an activity that can never be executed (in that case, such an activity is called *dead*).

Definition 12 (Dead activity). *Let* $\text{DS} = (\text{REP}, \text{Act}, K)$ *be a* DECLARE *specification. An activity* $\mathsf{a} \in \text{Act}$ *is* dead *in* DS *if there is no model trace of* DS *where* a *occurs.* ◁

Example 9. Consider the DECLARE specification in Fig. 7(b). The specification is consistent; as an example, trace $\langle \mathsf{c}, \mathsf{d} \rangle$ is a model trace. However, none of its model traces can foresee the execution of b. This can be seen if one tries to construct a trace containing an occurrence of b. The result is the following:



It is apparent that the presence of b requires a previous occurrence of a and, indirectly, a future occurrence of d, violating NOTRESPONSE(a, d). This shows that b is a dead activity.

Consider now the specification in Fig. 7(c). The situation here is trickier. The specification is consistent, as it accepts the empty trace (where no activity is executed, and hence none of the two response constraints present in the specification gets activated). However, none of the two activities a and b present therein can occur. As soon as this happens, the combination of the two response constraints cannot be *finitely* satisfied. In fact, an occurrence of a requires a later occurrence of b, which in turn requires a later occurrence of a, and so on and so forth, indefinitely. In other words, in every instant, one between RESPONSE(a, b) and RESPONSE(b, a) must be active and waiting for a later occurrence of its target, in a future instant. Since every instant must have a next instant, it is not possible to construct a satisfying (finite) trace. ◁

Dead activity detection can be directly reduced to (in)consistency of a specification. Specifically, activity a is dead in a DECLARE specification $\text{DS} = (\text{REP}, \text{Act}, K)$ if and only if the specification $(\text{REP}, \text{Act}, K \cup \{\text{ATLEASTONE}(\mathsf{a})\})$, obtained from DS by forcing the existence of a is inconsistent (i.e., its specification formula is not satisfiable).

Valid Activation and Target. To ensure that a DECLARE constraint K comes with a valid activation $_{\square}\text{K}$ and target $\text{K}_{\blacktriangleright}$ for its formula φ_{K}, we can directly apply Definition 10 and check whether the LTL_f formula $\varphi_{\text{K}} \leftrightarrow \Box({}_{\square}\text{K} \rightarrow \text{K}_{\blacktriangleright})$ is valid, that is, whether its negation is not satisfiable.

Checking Relations Between Constraints/Specifications. We establish two key relations between constraints/specifications. The first is that of *subsumption* between templates, leveraging the entailment relation between LTL_f formulae to constraints. We formally define it as follows.

Definition 13 (Subsumption). *Let* $\textsc{k}(x_1,\ldots,x_m), \textsc{k}'(x_1,\ldots,x_m) \in \textsc{Rep}$ *two templates.* $\textsc{k}(x_1,\ldots,x_m)$ subsumes $\textsc{k}'(x_1,\ldots,x_m)$ *(in symbols,* $\textsc{k}(x_1,\ldots,x_m) \sqsubseteq \textsc{k}'(x_1,\ldots,x_m)$*) if, given any mapping* κ *assigning* $x_1,\ldots,x_m$ *with activities* $a_1,\ldots,a_m \in \text{Act}$, $\varphi_{\textsc{k}(a_1,\ldots,a_m)} \models \varphi_{\textsc{k}'(a_1,\ldots,a_m)}$. $\triangleleft$

This relation can be checked by verifying that $\varphi_{\textsc{k}(a_1,\ldots,a_m)} \rightarrow \varphi_{\textsc{k}'(a_1,\ldots,a_m)}$ is valid, that is, the negated formula $\varphi_{\textsc{k}(a_1,\ldots,a_m)} \wedge \neg\varphi_{\textsc{k}'(a_1,\ldots,a_m)}$ is not satisfiable for any $a_1,\ldots,a_m \in \text{Act}$. For example, $\textsc{Alt.Prec.}(x,y) \sqsubseteq \textsc{Precedence}(x,y)$ as the former requires that y can occur only if preceded by x (just as the latter) *and* y does not recur in between. Therefore, every event that satisfies the former must satisfy the latter too. In the following, we shall lift this notion to constraints too (e.g., we say that $\textsc{AlternatePrecedence}(\mathsf{y},\mathsf{p})$ subsumes $\textsc{Precedence}(\mathsf{y},\mathsf{p})$).

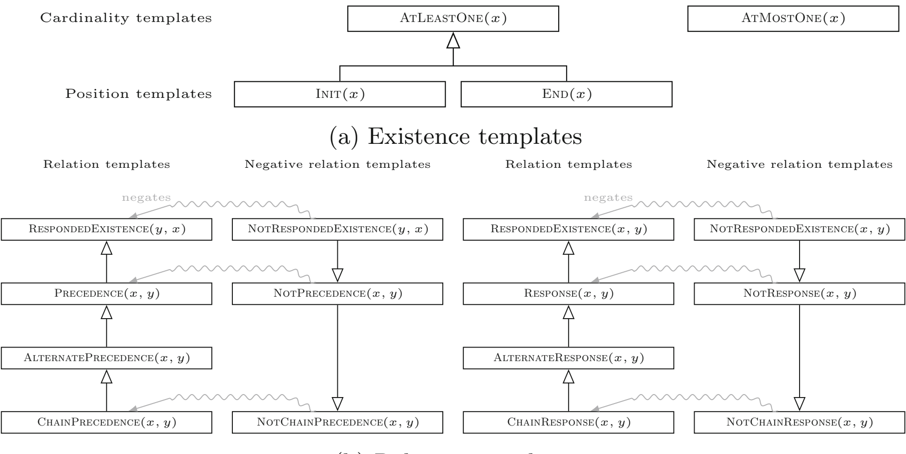
By Definition 8 and Definition 9, since both Declare constraints and specifications correspond to LTL_f formulae, we can use subsumption for a twofold purpose:

- Consider two candidate constraints $\textsc{k}_1$ and $\textsc{k}_2$. If $\textsc{k}_1 \sqsubseteq \textsc{k}_2$, then we know that adding $\textsc{k}_1$ to a Declare specification will make the addition of $\textsc{k}_2$ irrelevant, and that adding $\textsc{k}_1$ or $\textsc{k}_2$ will determine whether the specification is more or less constraining.
- Consider a candidate constraint $\textsc{k}$ and a target specification DS. If the former logically entails the latter, $\varphi_{\text{DS}} \models \varphi_{\textsc{k}}$, then $\textsc{k}$ is *redundant* in DS, and it makes no sense to include it in DS.

The second relation characterizes constraints that are the negated version of each other. Let $\textsc{k}_1$ and $\textsc{k}_2$ be two Declare constraints, coming with activation formulae ${}_{\square}\textsc{k}_1$ and ${}_{\square}\textsc{k}_2$ and target formulae $\textsc{k}_{1\blacktriangleright}$ and $\textsc{k}_{2\blacktriangleright}$, respectively. We say that $\textsc{k}_1$ and $\textsc{k}_2$ are the *negated versions* of one another if their activations are logically equivalent, that is ${}_{\square}\textsc{k}_1 \leftrightarrow {}_{\square}\textsc{k}_2$, and their targets are incompatible, that is, $\textsc{k}_{1\blacktriangleright} \wedge \textsc{k}_{2\blacktriangleright}$ is false. An example is that of Response vs NotResponse.

Consider now the situation where a decision must be taken concerning which of two candidate constraints $\textsc{k}_1$ and $\textsc{k}_2$ can be added to a Declare specification. Knowing that $\textsc{k}_1$ and $\textsc{k}_2$ are the negated versions of one another indicates that they should not *both* be added to the specification, as including them both would make the specification inconsistent as soon as the two constraints are activated.

As we will see in the next section, these notions become key when dealing with declarative process mining, and in particular the discovery of Declare specifications from event logs. Figure 8 graphically depicts how the main Declare constraint templates relate to each other in terms of subsumption and negated versions.



(a) Existence templates

(b) Relation templates

Fig. 8. The subsumption map of Declare templates. Templates are indicated with solid boxes. The subsumption relation is depicted as a line starting from the subsumed template and ending in the subsuming one, with an empty triangular arrow recalling the UML IS-A graphical notation. The negative templates are graphically linked to the corresponding relation templates by means of wavy grey arcs.

5 Declarative Process Mining

Declarative process constraints depict the interplay of every activity in the process with the rest of the activities. As a consequence, the behavioural relationships that hold among activities can be analysed with a local focus on each one [9], as a projection of the whole process behaviour on a single element thereof. The constraints pertaining to a single activity thus be seen as its footprint in the global behaviour of the process. We shall interchangeably interpret Declare constraints as *(i)* behavioural relations between activities in a process specification or *(ii)* rules exerted on the occurrence of events in traces. Notice that the latter is a different approach than the former, typically used for process modelling as originally conceived by the seminal work of Pesic et al. [77]. The former is instead the basis for declarative process mining. In the following, we describe how process specifications can be discovered and monitored.

5.1 Declarative Process Discovery

Declarative process discovery refers to the inference of those constraints that significantly rule the behaviour of a process, based upon an input event log. The problem can be framed in two distinct ways:

- A *discriminative discovery problem*, reminiscent of a classification task. This requires to split the input event log in two partitions, one containing “positive” examples and the second containing “negative” examples. Discovery

Algorithm 1: Overview of the discovery algorithm

```
Input: L ∈ B(U*_act), the event log to be analyzed;
       Rep, a finite set of Declare templates to be considered to express the discovered specification;
       Act ⊆ U_act, a finite set of activities to be included in the discovered specification;
       conf_t^min, supp_t^min, conf_e^min, supp_e^min, the minimum thresholds for trace-based confidence and
       support, and event-based confidence and support, respectively (default for all four parameters: 0.0);
Output: DS, a declarative process specification
1  K ← {κ(a_1, ..., a_m) : κ ∈ Rep, a_1; ..., a_m ∈ Act, a_i ≠ a_j with 1 ≤ i, j ≤ m}
                 /* candidate constraints: templates assigned with any pair of distinct activities */
2  foreach κ ∈ K                                                          /* compute measures */
3   do
4     c_t ← conf_t(κ, L); s_e ← supp_t(κ, L); c_e ← conf_e(κ, L); s_e ← supp_e(κ, L)
5     if c_t ≤ conf_t^min or s_t ≤ supp_t^min or c_e ≤ conf_e^min or s_e ≤ supp_e^min then
6        K ← K \ {κ}                            /* remove constraints with a measure below the threshold */
7  foreach κ ∈ K                  /* remove constraints as per subsumption hierarchy and negated v. */
8   do
9     foreach κ' ∈ K s.t. κ' ⊑ κ                              /* for every κ' that subsumes κ in K */
10     do
11        if allm(κ', L) ≤ allm(κ, L)                        /* if the measures of κ' are ≤ those of κ */
12         then
13           K ← K \ {κ}
14        else K ← K \ {κ'}
15    foreach κ' ∈ DS s.t. κ' is the negated version of κ do
16        if allm(κ', L) < allm(κ, L) then K ← K \ {κ}
17        else K ← K \ {κ'}
18 return DS = (Rep, Act, K)
```

amounts to find a suitable DECLARE specification that correctly reconstructs the classification, that is, accepts all positive examples and reject all negative ones.

- A *standard discovery problem* – also known as *specification mining* in the software engineering literature [53]. This calls for the individuation of which DECLARE constraints best describe the traces in the log, considering all of them as "positive" examples.

The first discovery algorithm for DECLARE treated discovery as a discriminative problem, exploiting inductive logic programming to tackle it [20,52]. In parallel, Goedertier et al. [46] brought forward techniques to generate negative examples from positive ones. Interestingly, this line of investigation recently received again the attention of the community [19,89].

Declarative process discovery framed as a standard discovery problem finds its two main exponents in *Declare Miner* [58] and *MINERful* [40], which have been then extended with an arsenal of techniques to improve the quality and correctness of the discovered specifications. We follow the second thread, summarizing the main ideas exploited therein, though reshaping the core concepts in an attempt to embrace the wider plethora of declarative process discovery techniques and the advancements they brought [8,18,59].

Process discovery in a declarative setting typically consists of the following phases:

1) The initial setup, i.e., the selection of *(i)* the templates to be sought for, *(ii)* the activities to be considered for the candidate constraints instantiating those templates, and *(iii)* the minimum thresholds for constraint interestingness measures to retain a candidate constraint;

2) The computation of interestingness measures for all the constraints that instantiate the given templates;
3) The simplification of the returned specification, through *(i)* the removal of constraints whose measures do not reach the user-specified thresholds, *(ii)* the pruning of the redundant constraints from the set, and *(iii)* the removal of one constraint for every pair of constraints that are the negated version of one another.

Algorithm 1 gives a bird-eye view of the approach in pseudocode. As we can observe, interestingness measures are crucial to determine the degree to which constraints are satisfied in the log. They have been introduced to indicate the level of reliability and relevance of constraints discovered from event logs, originally devised in the field of association rule mining [3] and adapted to the declarative process discovery context [17,65]. Among them, we recall support and confidence. Intuitively, support is a normalized measure quantifying how often the constraint is satisfied in the event log. Confidence considers the number of satisfactions with respect to the occurrences of the activations. We define them formally as follows.

Definition 14 (Trace-based measures). *Let L be a non-empty simplified event log with at least a non-empty trace, and* κ *a declarative constraint as per Definition 1. We define the trace-based support* $\mathrm{supp_t}$ *and the trace-based confidence* $\mathrm{conf_t}$ *as follows:*

$$\mathrm{supp_t}(\kappa, L) = \frac{\sum\limits_{t \in L : t \models \Diamond(_\square\kappa) \wedge \kappa} L(t)}{\sum\limits_{t \in L} L(t)}; \tag{1}$$

$$\mathrm{conf_t}(\kappa, L) = \frac{\sum\limits_{t \in L : t \models \Diamond(_\square\kappa) \wedge \kappa} L(t)}{\max\left\{1, \sum\limits_{t \in L : t \models \Diamond(_\square\kappa)} L(t)\right\}}. \tag{2}$$

◁

We remark that the condition at the numerator that the trace has to satisfy not only the constraint κ but also eventually its activation, i.e., $t \models \Diamond(_\square\kappa) \wedge \kappa$, serves the purpose of avoiding to count "vacuous satisfactions" discussed in Sect. 4.1. For example, while trace $\langle \mathsf{b}, \mathsf{c} \rangle$ satisfies ChainResponse(a, b), it does so vacuously, in the sense that it never activates the constraint. This intuitively means that ChainResponse(a, b), albeit satisfied, it cannot be interestingly used to describe the behaviour encoded in the trace. We recall that with $L(t)$ denotes the multiplicity of occurrences of t in the log L (see [1], Sect 3.1). The max term at the denominator of the formulation of confidence serves the purpose of avoiding a division by zero in case no trace satisfies $\Diamond(_\square\kappa)$.

Declare Miner first introduced the trace-based measures to discover specifications from logs, counting traces that (non-vacuously) satisfy constraints as a

whole. MINERful, instead, advocated also the adoption of measures that lie at the level of granularity of events. The similarities and differences between the two measuring schemes and the role of explicit activations and targets to tackle vacuity has been later systematized in [18]. The motivation behind the use of event-based measures is the ability to give a differently weight to traces violating the constraints in more than one instant: with trace-based measures, e.g., both traces $\langle \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{c}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{c} \rangle$ and $\langle \mathsf{b}, \mathsf{a}, \mathsf{c}, \mathsf{a}, \mathsf{c}, \mathsf{a}, \mathsf{a}, \mathsf{a}, \mathsf{a}, \mathsf{a}, \mathsf{a}, \mathsf{c} \rangle$ would count as single violations for $\textsc{ChainResponse}(\mathsf{a}, \mathsf{b})$. However, only the last occurrence of a out of ten leads to violation in the first trace, whereas all eight occurrences of a lead to violation in the second trace. Next, we formally capture the notion of event-based measures.

Definition 15 (Event-based measures). *Let L be a non-empty simplified event log with at least a non-empty trace, and κ a declarative constraint as per Definition 1. We define the event-based support* $\mathrm{supp_e}$ *and the event-based confidence* $\mathrm{conf_e}$ *as follows:*

$$\mathrm{supp_e}(\kappa, L) = \frac{\sum_{t \in L} |\{a_i \in t : a, i \models ({}_{\square}\kappa \wedge \kappa_{\blacktriangleright})\}| \times L(t)}{\sum_{t \in L} |t| \times L(t)}; \tag{3}$$

$$\mathrm{conf_e}(\kappa, L) = \frac{\sum_{t \in L} |\{a_i \in t : a, i \models ({}_{\square}\kappa \wedge \kappa_{\blacktriangleright})\}| \times L(t)}{\max\left\{1, \sum_{t \in L} |\{a_i \in t : a, i \models {}_{\square}\kappa\}| \times L(t)\right\}}. \tag{4}$$

◁

Again, the condition at the numerator that events satisfy both activation and target of the constraint is intended to avoid including vacuous satisfactions in the sum. The max term at the denominator of confidence is intended to avoid a division by zero in case no event satisfies ${}_{\square}\kappa$.

For the sake of readability, we shall denote with $\mathrm{allm}(\kappa, L)$ the tuple containing all computed measures for a constraint κ on the event log L: $\mathrm{allm}(\kappa, L) = (\mathrm{supp_t}(\kappa, L), \mathrm{conf_t}(\kappa, L), \mathrm{supp_e}(\kappa, L), \mathrm{conf_e}(\kappa, L))$. Given two constraints κ_1 and κ_2, we write $\mathrm{allm}(\kappa_1, L) \leq \mathrm{allm}(\kappa_2, L)$ if $\mathrm{supp_t}(\kappa_1, L) \leq \mathrm{supp_t}(\kappa_2, L)$, $\mathrm{conf_t}(\kappa_1, L) \leq \mathrm{conf_t}(\kappa_2, L)$, $\mathrm{supp_e}(\kappa_1, L) \leq \mathrm{conf_t}(\kappa_2, L)$, and $\mathrm{conf_e}(\kappa_1, L) \leq \mathrm{conf_t}(\kappa_2, L)$. We write $\mathrm{allm}(\kappa_1, L) \leq \mathrm{allm}(\kappa_2, L)$ if $\mathrm{allm}(\kappa_1, L) \leq \mathrm{allm}(\kappa_2, L)$ and $\mathrm{allm}(\kappa_2, L) \leq \mathrm{allm}(\kappa_1, L)$.

Example 10 (An event log for the specification in Example 1). Let $\mathcal{U}_{act} \doteq \{\mathsf{c}, \mathsf{r}, \mathsf{v}, \mathsf{t}, \mathsf{n}, \mathsf{y}, \$, \mathsf{p}, \mathsf{e}, \mathsf{u}\} \cup \{@\}$ be an alphabet of activities. We interpret @ as an email exchange, which can occur at any stage during the process. The other activities in $\mathcal{U}_{act}$ are those that were considered in the process specification in Example 1. Let the following event log be built on $\mathcal{U}_{act}$:

Table 3. Measures computed for the relation constraints of Example 1 from the event log of Example 10.

Constraint	Event-based		Trace-based	
	Confidence	Support	Confidence	Support
PRECEDENCE(c, r)	1	0.129	1	1
ALTERNATEPRECEDENCE(r, v)	1	0.129	1	1
ALTERNATERESPONSE(r, v)	0.997	0.129	0.996	0.996
PRECEDENCE(t, v)	0.997	0.129	0.996	0.996
ALTERNATEPRECEDENCE(v, n)	1	0.059	1	0.461
ALTERNATEPRECEDENCE(v, y)	1	0.084	1	0.856
NOTRESPONSE(y, n)	1	0.084	1	0.856
PRECEDENCE(y, p)	1	0.07	1	0.715
PRECEDENCE($, p)	1	0.07	1	0.715
CHAINRESPONSE($, p)	1	0.07	1	0.715
ATMOSTONE(p)	1	1	1	1
PRECEDENCE(p, e)	1	0.07	1	0.715
ATMOSTONE(e)	1	1	1	1
PRECEDENCE(u, e)	0.985	0.069	0.985	0.704

$L = [t_1^{200}, t_2^{100}, t_3^{100}, t_4^{80}, t_5^{80}, t_6^{4}, t_7^{2}, t_8^{2}]$ where

$t_1 = \langle \mathsf{c}, \mathsf{t}, \mathsf{r}, \mathsf{v}, \mathsf{y}, \$, \mathsf{p}, \mathsf{u}, \mathsf{e} \rangle$ $\quad t_2 = \langle \mathsf{c}, \mathsf{t}, \mathsf{t}, \mathsf{r}, \mathsf{v}, \mathsf{n}, \mathsf{t}, \mathsf{r}, \mathsf{v}, \mathsf{y}, \$, \mathsf{p}, \mathsf{u}, \mathsf{e} \rangle$

$t_3 = \langle \mathsf{c}, \mathsf{t}, \mathsf{r}, \mathsf{t}, \mathsf{v}, \mathsf{y}, \mathsf{u}, \$, \mathsf{p}, \mathsf{e} \rangle$ $\quad t_4 = \langle \mathsf{c}, \mathsf{t}, @, \mathsf{t}, \mathsf{r}, \mathsf{v}, \mathsf{n}, @, \mathsf{r}, \mathsf{v}, \mathsf{n} \rangle$

$t_5 = \langle \mathsf{c}, \mathsf{r}, \mathsf{t}, \mathsf{t}, \mathsf{v}, \mathsf{n}, \mathsf{y}, @ \rangle$ $\quad t_6 = \langle \mathsf{c}, \mathsf{t}, \mathsf{r}, \mathsf{t}, \mathsf{v}, @, @, \mathsf{y}, \$, \mathsf{p}, @, \mathsf{e} \rangle$

$t_7 = \langle \mathsf{c}, @, \mathsf{r}, \mathsf{v}, \mathsf{y}, \$, \mathsf{p}, @, \mathsf{e} \rangle$ $\quad t_8 = \langle \mathsf{c}, \mathsf{t}, \mathsf{r}, \mathsf{r}, \mathsf{v}, @, \mathsf{n} \rangle$

We observe that the log above does not fully comply with the specification. Indeed, *(i)* trace t_8 violates ALTERNATERESPONSE(r, v), as the candidate managed to register twice before evaluation (notice the occurrence of two consecutive r's before v); *(ii)* t_7 violates PRECEDENCE(t, v) and PRECEDENCE(u, e), as the candidate must have sent the admission test score and the necessary enrolment documents via email rather than via the system (see the occurrence of @ in place of t in the second instant and in place of u later in the trace); finally, *(iii)* trace t_6 violates PRECEDENCE(u, e), as the candidate must have submitted the enrolment documents via email in that case too (notice the absence of task u and the presence of @ in its stance). ◁

Example 11. With the example above, we have that both the trace-based support and trace-based confidence of ALT.PREC.(r, v), e.g., equate to 1.0: $\text{supp}_{\mathsf{t}}(\text{PRECEDENCE}(\mathsf{c}, \mathsf{r}), L) = \text{conf}_{\mathsf{t}}(\text{PRECEDENCE}(\mathsf{c}, \mathsf{r}), L) = 1.0$. This is because in all traces the activator (i.e., r) occurs and the constraint is not violated in any trace. Instead, $\text{supp}_{\mathsf{t}}(\text{ALT.PREC.}(\mathsf{v}, \mathsf{n}), L) = \frac{100+80+80+2}{568} \approx 0.461$

and $\mathrm{conf_t}(\textsc{Alt.Prec.}(\mathsf{v},\mathsf{n}), L) = 1.0$. The trace-based support is lower than the trace-based confidence because the activator (n) occurs in 262 traces out of 568 (i.e., in the 100 instances of t_2, the 80 instances of t_4, the 80 instances of t_5, and the 2 instances of t_8). Similarly, $\mathrm{conf_e}(\textsc{Precedence}(\mathsf{c},\mathsf{r}), L) = 1.0$ and $\mathrm{conf_e}(\textsc{Alt.Prec.}(\mathsf{v},\mathsf{n}), L) = 1.0$. The measures do not change for event-based and trace-based confidence because every activation of the two constraints above leads to a satisfaction. In contrast, $\mathrm{supp_e}(\textsc{Precedence}(\mathsf{c},\mathsf{r}), L) = \frac{1\times200+2\times100+1\times100+2\times80+1\times80+1\times4+1\times2+2\times2}{9\times200+14\times100+10\times100+11\times80+8\times80+12\times4+9\times2+7\times2} = \frac{750}{5800} \approx 0.129$. ◁

It is worth noting that discovery approaches such as Declare Miner [58] and Janus [18] adopt (variations of) local constraint automata to count the satisfactions of constraints. MINERful [40] and DisCoveR [8] resort to occurrence statistics of activities gathered from the event log, more closely to the procedural discovery algorithms discussed in [2].

By definition of confidence and support (trace- or event-based), and as exemplified above, we observe that trace-based confidence is an upper bound for trace-based support and event-based confidence is an upper bound for event-based support. Next, we illustrate how the discovery algorithm operates with our running example.

Example 12. Table 3 shows the event-based and trace-based measures computed on the basis of our running example for every constraint in the original specification – phase (2) of the discovery procedure described above. They belong to the output of the discovery algorithm running on the event log of Example 10 set at phase (1) to seek for *(i)* all templates from the Declare repertoire in Table 2 *(ii)* over activities $\{\mathsf{c},\mathsf{r},\mathsf{v},\mathsf{t},\mathsf{n},\mathsf{y},\$,\mathsf{p},\mathsf{e},\mathsf{u}\}$, with *(iii)* minimum event-based confidence of 0.95. We remark that also $\textsc{AlternatePrecedence}(\mathsf{y},\mathsf{p})$, $\textsc{ChainPrecedence}(\$,\mathsf{p})$, $\textsc{AlternatePrecedence}(\mathsf{p},\mathsf{e})$ and $\textsc{AlternatePrecedence}(\mathsf{c},\mathsf{p})$, $\textsc{NotChainPrecedence}(\mathsf{y},\mathsf{p})$ and $\textsc{NotChainResponse}(\mathsf{y},\mathsf{p})$, among others, fulfil those criteria and thus are part of the returned set. ◁

To increase the information brought by a discovered model, not only we prune the constraints whose measures lie below the given threshold values. Also, we take into account the subsumption hierarchy illustrated in Fig. 8. In addition, we retain in the constraint set only one among pairs that are a negated version of one another. If we kept both, the model would turn the activation in common into a dead activity (see Sect. 4.2).

Example 13. Figure 9 illustrates the result of the pruning phase (3) based on subsumption and choice of constraints that are the negated version of one another, based on the event log of Example 10. We observe that $\textsc{AlternatePrecedence}(\mathsf{y},\mathsf{p})$ has the same measures as $\textsc{Precedence}(\mathsf{y},\mathsf{p})$, and we know that $\textsc{Precedence}(\mathsf{y},\mathsf{p})$ is subsumed by $\textsc{AlternatePrecedence}(\mathsf{y},\mathsf{p})$ (see Sect. 4.2); as we are interested in more restrictive constraints that reduce the space of possible process runs to more closely define its behaviour, we retain the former and discard the latter. Keeping both would introduce a redundancy,

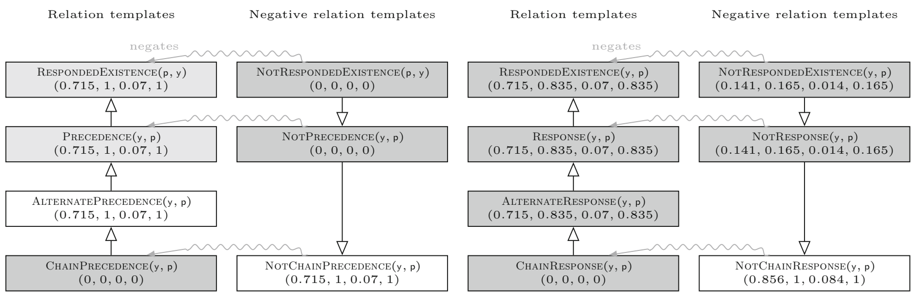


Fig. 9. The subsumption map of relation Declare constraints in a discovery context. The graphical notation follows Fig. 8. Gray boxes denote constraints that have measures below the minimum thresholds. Light-gray boxes indicate constraints that are subsumed by others with equivalent measures.

and retaining only the latter would omit detailed information as not only p must be preceded by y, but also p cannot recur unless y occurs again. By the same line of reasoning, we prefer retaining Init(c) to AtMostOne(c) in the result specification. The same concepts apply with ChainPrecedence($, p), to be preferred over Precedence($, p) and AlternatePrecedence(p, e) in place of Precedence(p, e), among others. Notice that Precedence(y, p), Precedence($, p) and Precedence(p, e) were in the given specification of our running example but, we conclude, are not the most restrictive constraints that could be used in the specification, as the discovery algorithm evidences. ◁

To conclude, we remark that not all redundancies can be found with the sole subsumption-hierarchy based pruning. The subsumption hierarchy, indeed, checks constraints that are exerted on the same activities – e.g., AlternatePrecedence(y, p) and Precedence(y, p). Therefore, we need a more powerful redundancy checking mechanism, seeking for constraints that are entailed by the remainder of the specification's constraint set (see Sect. 4.2).

Example 14. The confidence of AlternatePrecedence(v, p) is 1.0 in the event log of our running example. Yet, it does not add information to the discovered specification as it is redundant, logically entailed by the other constraints – in particular, AlternatePrecedence(r, v), AlternatePrecedence(v, y), Precedence(y, p) and AtMostOne(p). ◁

To verify this, we can resort to language inclusion via automata product as in [38]: the language of the product of the four constraint automata is not smaller than the language accepted by the intersection of the second, third and fourth constraint automata. Here, we do not enter the details of the algorithms that detect redundancies at such a deeper level but provide an example of its rationale. The interested reader can find further details in [24,38].

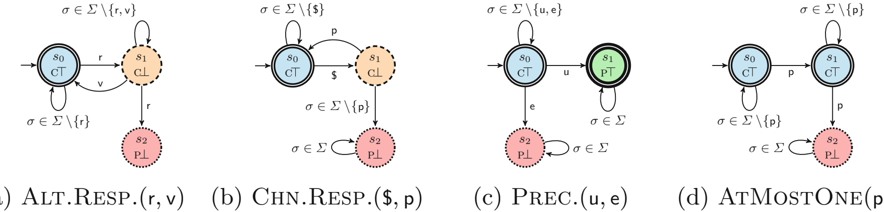


Fig. 10. Example FSAs adapted for the monitoring of constraints. Non-final states indicating current violation (c⊥) are dashed and filled in orange; non-final states indicating permanent violation (p⊥) are dotted and filled in red; final states indicating current satisfaction (c⊤) are thin-solid and filled in blue; final states indicating permanent satisfaction (p⊤) are thick-solid and filled in green. (Color figure online)

5.2 Declarative Process Monitoring

(Compliance) process monitoring aims at tracking running process executions to check their conformance to a reference process model, with the purpose of detecting and reporting deviations as soon as possible [57]. It constitutes one of the main tasks of operational decision support [92, Ch. 10], which characterizes process mining applied at runtime to running process executions.

Declarative process monitoring employs a declarative specification (in our case, described using Declare) as reference model for monitoring. The central fact in monitoring that process instances are running, that is, their generated traces evolve over time, calls for a finer-grained understanding of the state of constraints and of the whole specification. We illustrate this intuitively in the next example.

Example 15. Consider the excerpt in Fig. 11 of our admission process running example, and an evolving trace that, once completed, corresponds to the following sequence: $\langle \$, \mathsf{p}, \mathsf{u}, \$, \mathsf{p} \rangle$. Let us replay the trace from the beginning.

1. At the beginning, all constraints are *satisfied*, but they are so for sure only *currently*, as events may occur making them violated. For example, a registration without a consequent evaluation would lead to violating AlternateResponse(r, v), whereas an enrolment without a prior upload of certificates would lead to a violation of Precedence(u, e).
2. Upon the occurrence of $, constraint ChainResponse($, p) becomes pending or, to be more precise, *currently violated*, as paying demands a pre-enrolment occurring immediately after.
3. The execution of p brings ChainResponse($, p) back to *currently satisfied*, as it does not require the occurrence of further events, but may do so in the future in case of another payment.
4. Upon the occurrence of u, constraint Precedence(u, e) becomes *permanently satisfied*, as enrolment is now enabled, and there is no way to continue the execution leading to a violation of the constraint.

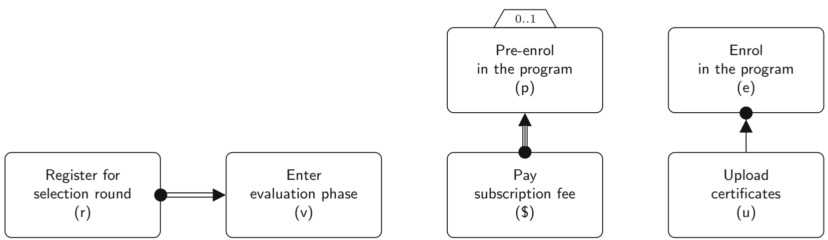


Fig. 11. Excerpt of the DECLARE specification in Fig. 2.

5. This is indeed what happens with the next occurrence of \$, which makes CHAINRESPONSE(\$, p) *currently violated*.
6. The second pre-enrolment has the effect of bringing CHAINRESPONSE(\$, p) once again back to *currently satisfied*. However, it has also the effect of *permanently violating* ATMOSTONE(p), as the number of occurrences of p has exceeded the upper bound allowed by ATMOSTONE(p), and there is no way of fixing the violation.

◁

As witnessed by the example, the state of each constraint can be described in a fine-grained way by considering on the one hand the trace accumulated so far (i.e., the prefix of the whole, still unknown, execution), and by pondering on the other hand about the possible, future continuations. To do so in a formal way, we appeal to the literature on runtime-verification for linear temporal logics, and in particular to the RV-LTL semantics, originally introduced in [11] over infinite traces. This semantics was adopted for the first time in the context of LTL_f over finite traces in [64,66], in order to define an operational technique for DECLARE monitoring. This led to deeper investigations on the usage of RV-LTLto characterize the relevance of a trace to a declarative specification [39], and to finally obtain a formally grounded, comprehensive framework for monitoring [27,28].

We now define the RV-LTL semantics for LTL_f. In the definition, we denote the concatenation of trace t_1 with t_2 as $t_1 \cdot t_2$.

Definition 16 (RV-LTL states). *Consider an* LTL_f *formula* φ *over* Σ, *and a trace* t *over* Σ^*. *We say that* φ is in (RV-LTL) state s after t, *written* $[t \models \varphi]_{\text{RV}} = v$, *if:*

(Permanent satisfaction) *(i)* $v = \text{P}\top$, *(ii) the current trace satisfies* φ $(t \models \varphi)$, *and (iii) every possible suffix keeps* φ *satisfied (for every trace* $t' \in \Sigma^*$, *we have* $t \cdot t' \models \varphi$*).*

(Permanent violation) *(i)* $v = \text{P}\bot$, *(ii) the current trace violates* φ $(t \not\models \varphi)$, *and (iii) every possible suffix keeps* φ *violated (for every trace* $t' \in \Sigma^*$, *we have* $t \cdot t' \not\models \varphi$*).*

(Current satisfaction) *(i)* $v = \text{C}\top$, *(ii) the current trace satisfies* φ $(t \models \varphi)$, *and (iii) there exists a suffix that leads to violate* φ *(for some trace* $t' \in \Sigma^*$, *we have* $t \cdot t' \not\models \varphi$*).*

(Current violation) *(i)* $v = \textsc{c}\bot$, *(ii) the current trace violates* φ *(*$t \not\models \varphi$*), and (iii) there exists a suffix that leads to satisfy* φ *(for some trace* $t' \in \Sigma^*$*, we have* $t \cdot t' \models \varphi$*).*

We also say that t conforms to φ *if* $[t \models \varphi]_{\text{RV}} = \textsc{p}\top$ *or* $[t \models \varphi]_{\text{RV}} = \textsc{c}\top$ *(i.e., stopping the execution in* t *satisfies the formula).* ◁

By inspecting the definition, we can directly see that monitoring is at least as hard as LTL$_f$ satisfiability/validity checking. To see this, consider what happens at the *beginning of an execution*, where the current trace is empty. By applying Definition 16 to this special case, and by recalling the notion of satisfiability/validity of an LTL$_f$ formula, we in fact get that an LTL$_f$ formula φ is:

- permanently satisfied if φ is valid;
- permanently violated if φ is unsatisfiable;
- currently satisfied if the two formulae $\varphi \wedge \textbf{end}$ and $\neg\varphi$ are both satisfiable;
- currently violated if the two formulae $\neg\varphi \wedge \textbf{end}$ and φ are both satisfiable.

To perform monitoring according to the RV-LTL states from Definition 16, we can once again exploit the automata-theoretic characterization of LTL$_f$. In particular, given an LTL$_f$ formula φ, we construct its FSA A_φ, and *color* the automaton states according to the RV-LTL semantics. As introduced in [64] and then formally verified in [28], this can be simply done as follows. Consider a state s in of A_φ. We label it by:

- $\textsc{p}\top$, if s is final and all the states reachable from s in A_φ are final as well; if A_φ is minimized, this means that s only reaches itself.
- $\textsc{p}\bot$, if s is non-final and all the states reachable from s in A_φ are non-final as well; if A_φ is minimized, this means that s only reaches itself.
- $\textsc{c}\top$, if s is final and can reach a non-final state in A_φ.
- $\textsc{c}\bot$, if s is non-final and can reach a final state in A_φ.

Figure 10 shows some examples of colored constraint automata, obtained by considering the constraint formulae of some Declare constraints from our running example. To monitor the state evolution of a constraint, one has simply to dynamically play the evolving trace on its colored local automaton, returning the updated RV-LTL label as soon as a new event is processed. Doing so on the local automata in Fig. 10 for trace $\langle \mathsf{\$}, \mathsf{p}, \mathsf{u}, \mathsf{\$}, \mathsf{p} \rangle$ formally reconstructs what discussed in Example 15.

However, this is not enough to *promptly detect violations* as soon as they manifest in the traces. This has been extensively discussed in [28,66], and is at the very core of the power of temporal logic-based techniques for monitoring. We use again Example 15 to illustrate the problem.

Example 16. Consider Example 15 and the following question: is step 6 the earliest at which a violation can be detected? Clearly, if we focus on each constraint in isolation, the answer is affirmative. To see this formally, we play trace $\langle \mathsf{\$}, \mathsf{p}, \mathsf{u}, \mathsf{\$}, \mathsf{p} \rangle$ on the four colored local automata of Fig. 10, obtaining the following runs:

- For AlternateResponse(r, v), we have $s_0 \xrightarrow{\$} s_0 \xrightarrow{p} s_0 \xrightarrow{u} s_0 \xrightarrow{\$} s_0 \xrightarrow{p} s_0$; no violation is encountered.
- For ChainResponse($, p), we have $s_0 \xrightarrow{\$} s_1 \xrightarrow{p} s_0 \xrightarrow{u} s_0 \xrightarrow{\$} s_1 \xrightarrow{p} s_0$; no violation is encountered.
- For Precedence(u, e), we have $s_0 \xrightarrow{\$} s_0 \xrightarrow{p} s_0 \xrightarrow{u} s_1 \xrightarrow{\$} s_1 \xrightarrow{p} s_1$; no violation is encountered.
- For AtMostOne(p), we have $s_0 \xrightarrow{\$} s_0 \xrightarrow{p} s_1 \xrightarrow{u} s_1 \xrightarrow{\$} s_1 \xrightarrow{p} s_2$; a violation is encountered in the last reached state.

The answer changes if we consider the whole Declare specification that contains all such constraints at once. In fact, by taking into account the interplay of constraints, we can detect a violation already at step 5, i.e., after the second occurrence of payment. This is because, after that step, the two constraints ChainResponse($, p) and AtMostOne(p) enter into a *conflict*, that is, no continuation of the current trace can lead to satisfy them both. In fact, after trace ⟨$, p, u, $⟩, constraint ChainResponse($, p) is currently violated, waiting for a consequent occurrence of p; however, constraint AtMostOne(p), which is currently satisfied, becomes permanently violated upon a further occurrence of p. ◁

As we have seen, the early detection of violations cannot always be caught by considering the colored local automata of constraints in isolation. However, it can be systematically detected by taking into account the colored global automaton of the whole specification.

Example 17. Figure 12 shows the colored global automaton of the Declare specification in Fig. 11. By playing the trace ⟨$, p, u, $, p⟩ therein, we obtain the following run: $s_0 \xrightarrow{\$} s_1 \xrightarrow{p} s_4 \xrightarrow{u} s_8 \xrightarrow{\$} s_{12} \xrightarrow{p} s_{12}$. Clearly, the violation state s_{12} is already reached in step 5, i.e., just after the second payment. ◁

All in all, we can then monitor an evolving trace against a Declare specification as follows:

- Each constraint is encoded into the corresponding colored local automaton, used to track the state evolution of the constraint itself.
- The whole specification is encoded into the corresponding colored global automaon, used to track the evolution of the whole specification, and in particular to early-detect violations.
- At runtime, every new event occurrence is delivered in parallel to all the automata, updating each of them by executing the corresponding transition and entering into the next state, at the same time returning the associated RV-LTL label.

Figure 13 shows the result of applying this technique to our running example.

An alternative approach, which is exploited in [64], is to compute, as done before, the global automaton as the cross-product of local automata, remembering, in each global state, the RV-LTL labels of all local states from which such

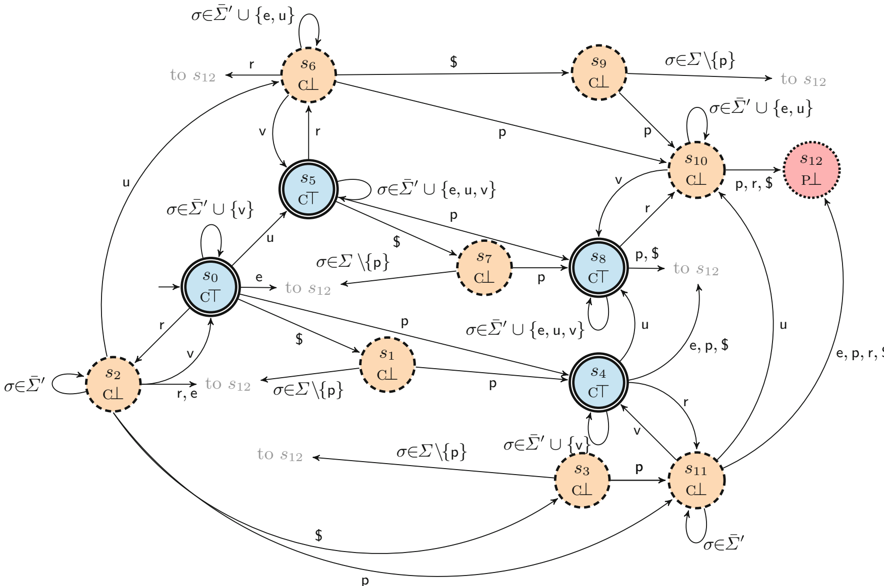


Fig. 12. The colored global automaton automaton obtained as the (colored) cross-product of constraints in Fig. 10 as shown in Fig. 6(c), the states of which are decorated with the four RV-LTL truth values.

a global state has been produced. In addition, no minimization step is applied on the resulting automaton. Once colored, this non-minimized, global colored automaton combines in a single device the contribution of all local monitors and that of the global monitor.

5.3 A Note on Conformance Checking

In this section, we have focused on monitoring evolving traces against DECLARE specifications. This can be seen as a form of *online conformance checking*, aiming at detecting deviations at execution time. This technique can be seamlessly lifted to handle the standard conformance checking task, where conformance is evaluated on an event log containing full traces of already completed process executions (cf. [16]). In this setting, the global automaton is not needed anymore, as a-posteriori it is not relevant to compute the earliest moment of a violation, but only to properly detect it at the trace level. The usage of local automata, one per constraint, is enough, and also has the advantage of producing an informative feedback that indicates, trace by trace, how many (and which) constraints are satisfied or violated. Finer-grained feedbacks like those based on the computation of trace alignments have been extensively applied for procedural models (cf. [16]), and can be also recasted in the declarative setting, aligning the log traces with the (closest) model traces accepted by the global automaton

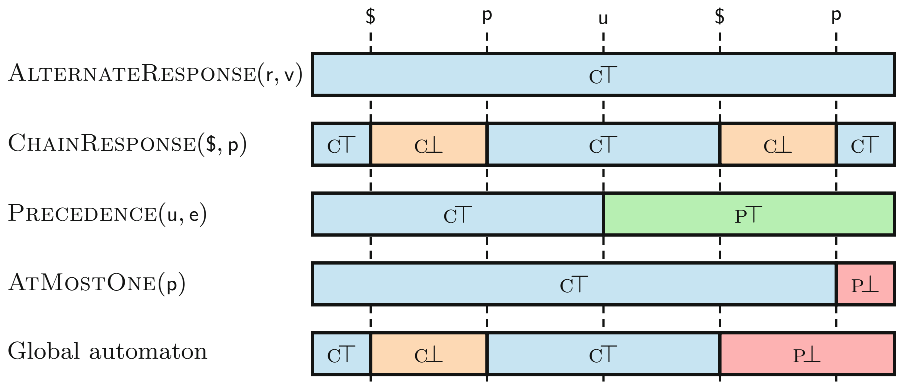


Fig. 13. Monitoring with local and global colored automata, showing a case where the global automaton detects a violation before it actually manifests on a single constraint.

of the Declare specification of interest. This is an active line of research, which started from the seminal approach in [31].

6 Recent Advances and Outlook

We close this chapter by reporting about the most recent advances in the field of declarative process mining revolving around Declare, describing the current frontier of research, and highlighting open challenges.

6.1 Beyond Declare Patterns

As we have seen in Sect. 3, a Declare specification consists of a repertoire of constraint templates grounded on specific activities. At the same time, such templates come with a logic-based semantics given in terms of LTL_f. A natural question is then: can the techniques described in this chapter be used for the *entire* LTL_f logic? This means, more precisely, considering the situation where each constraint corresponds to an arbitrary LTL_f formula while, as usual, the specification formula is constructed by putting in conjunction the LTL_f formulae of all its constituting constraints.

To answer this question, one has to separate the *logical* and *pragmatic* aspects involved in the different tasks we have been introducing. We do so focusing on reasoning, discovery, and monitoring.

Reasoning. As discussed in Sect. 4.2, all the reasoning tasks we have considered in this chapter can be lifted to the whole LTL_f logic. Indeed, they are reduced to LTL_f satisfiability/validity checking, which in turn can be tackled by checking (non-)emptiness of FSAs. The situation may change if one wants to provide more advanced debugging or diagnosis functionalities – for example, to return the most

relevant conflicting set(s) of constraints that are causing inconsistencies or dead activities. While these types of problem can also be attacked at the level of the entire logic [25,79], focusing only on pre-defined patterns becomes necessary if one wants to involve humans in the loop or define preferences over constraints in the case where multiple explanations exist [25]. Considering specific patterns is also relevant when studying the computational complexity of reasoning on pattern combinations [44,45,91], or the scalability and effectiveness of reasoning tools [44,45,71,97].

Discovery. As pointed out in Sect. 5.1, two distinct process discovery problems are typically tackled in a declarative setting: discriminative discovery and specification mining.

The case of discriminative discovery is tightly related to classification and machine learning, allowing one to rely on general learning algorithms for declarative process mining. Such algorithms tackle general logical frameworks, such as Horn clauses in inductive logic programming or full temporal logics in model learning, and can thus go far beyond a pre-defined set of templates, either targeting full LTL$_f$ [15,82] or enriching the discoverable DECLARE templates with further key dimensions, such as metric temporal constraints, event attributes, and data conditions [21,23].

As shown in Sect. 5.1, standard discovery stands as a radically different problem, since the input event log provides a uniform set of (positive) examples, while no negative example is given. This calls for suitable metrics to measure *how well* a set of constraints characterizes the behaviour contained in the log. In the approach described in this chapter, such metrics are defined starting from the notions of constraint activation and target, which are template-specific. Attempts have been conducted to lift some of these notions (in particular that of activation and "relevant" satisfaction [39]) to full LTL$_f$, but further research is needed to target the discovery of arbitrary LTL$_f$ formulae from event logs. Notice that while full LTL$_f$ discovery would enrich the expressiveness of the discovered specifications, it would on the other hand pose the issue of *understandability*: end users may struggle when confronted with arbitrary temporal formulae, while they are facilitated when pre-defined templates are used.

Monitoring. As we have discussed in Sect. 5.2, DECLARE monitoring is tackled using automata, and consequently seamlessly work for arbitrary LTL$_f$ formulae. As for advanced debugging techniques, the same considerations done for reasoning also hold for monitoring. For example, the detection of minimal conflicting sets of constraints in the case of early detection of violations caused by the interplay of multiple constraints can be tamed at the level of the full logic [66], but would require to focus on patterns if one wants to formulate preferences or incorporate human feedback [25].

Remarkably, working with FSAs allows us to define monitors for temporal formulae that go even beyond LTL$_f$. In fact, LTL$_f$ is as expressive as star-free regular expressions, while automata are able to capture full regular expressions and, in turn, finite-trace temporal logics incorporating in a single formalism

LTL_f and regular expressions, such as Linear Dynamic Logic over finite traces (LDL_f) [30]. Working with LDL_f in our setting has the specific advantage that we can express and monitor *metaconstraints*, that is, constraints that predicate on the RV-LTL truth values of other constraints [27,28].

6.2 Dealing with Uncertainty

In the conventional definition of a DECLARE specification, constraints are interpreted as being *certain*: every model trace is expected to satisfy all constraints contained in the specification. Such an interpretation is too restrictive in scenarios where the specification should accommodate:

- constraints describing common behaviours, expected to hold in the majority, but not all cases;
- constraints describing exceptional, outlier behaviours that rarely occurs but should be not judged as violating the specification.

To deal with this form of *uncertainty*, DECLARE has been recently extended with *probabilistic constraints* [62]. In this framework, every probabilistic constraint comes with:

- a constraint formula φ (specified, as in the standard case, using LTL_f);
- a comparison operator $\odot \in \{=, \neq, <, \leq, >, \geq\}$;
- a number $p \in [0, 1]$.

The interpretation of this constraint is that φ holds in a random trace generated by the process with a probability that is $\odot p$. In frequentist terms, this can be in turn interpreted as follows: given a log of the process, the ratio of traces satisfying φ must be $\odot p$.

Since a DECLARE specification contains multiple constraints, one has to consider how different probabilistic constraints interact with each other. In particular, n probabilistic constraints yield up to 2^n possible so-called *scenarios*, each highlighting which probabilistic constraints hold and which are violated. Reasoning over such scenarios has to be conducted by suitably mixing their temporal and probabilistic dimensions. The former handles which combinations of constraints and their violations (i.e., which scenarios) are consistent, while the latter lifts the probability conditions attached of single constraints to discrete probability distributions over the possible scenarios.

To carry out this form of combined reasoning, probabilistic constraints are formalized in a well-behaved fragment of the logic introduced in [61]. As it turns out, logical and probabilistic reasoning are loosely coupled in this fragment, and can be carried out resorting to standard finite-state automata and systems of linear inequalities. This approach has been used as the basis for defining a new family of *probabilistic declarative process mining* techniques [6].

6.3 Mixed-Paradigm Models

In Fig. 1, we have intuitively contrasted declarative specifications and imperative models. The distinction of these two approaches is in reality not so crisp. In fact, a single process may contain parts that are more suitably captured using imperative languages, and parts that can be better described as declarative specifications. Take, for instance, a clinical guideline mixing administrative and therapeutic subprocesses [73].

To capture such *hybrid* processes, one needs a multi-paradigm approach that can combine imperative and declarative constructs in a single process model. One of the first proposals doing so is [85], where an imperative process can contain activities that are internally structured using so-called *pockets of flexibility* specified using declarative temporal constraints over a given set of tasks.

This layered approach has been further developed in [90], which brings forward a hierarchical model where each sub-process can be specified either as an imperative or declarative component. Discovery of hierarchical hybrid process models has been subsequently tackled in [87].

Multi-paradigm approaches providing a tighter integration between imperative and declarative components have also been studied. In [33], process models combining Petri nets and Declare constraints at the same modelling level are introduced and studied, singling out methodologies and techniques to handle the intertwined state space emerging from their interaction. Conformance checking for these mixed-paradigm models is extensively assessed in [95]. A different approach is brought forward in [5], where a Declare specification is used to express global constraints that "glue together" multiple imperative processes concurrently executed over the same instances. Automata-based techniques extending those illustrated in Sect. 5.2 are introduced to provide integrated monitoring functionalities dealing at once with the local processes and the global constraints.

At the current stage, further research is needed along the illustrated lines towards a solid theory and corresponding algorithmic techniques for *hybrid, mixed-paradigm process mining*.

6.4 Multi-perspective Declare Specifications

Throughout the chapter, we have considered pure control-flow specifications, where a process is captured solely in terms of its constitutive activities and of behavioural constraints separating legal from undesired executions. While the control-flow provides the main process backbone, other equally important perspectives should also be taken into account as suggested already in [1]:

- The *resource* perspective deals with the actors that are responsible for executing tasks within the process.
- The *time* perspective focusses on quantitative temporal conditions on when tasks can/must be scheduled and executed, and on their expected durations.
- The *data* perspective captures how data objects and their attributes influence and are manipulated during the process execution.

Several works have investigated the extension of DECLARE with additional perspectives. From the formal point of view, this requires to extend the logic-based formalization of DECLARE with features that can capture resources, metric time, data, and conditions thereof, in turn resorting to variants of metric and/or first-order formalisms over finite traces [10,14,69,74]. It is important to stress that such features may be blurred, considering that data support (if equipped with suitable datatypes and conditions) may be used to predicate over resources and time as well.

Such multi-perspective features have been extensively embedded into DECLARE or related approaches (see, for example, [13,69,98] for constraints with metric time and [42] for constraints with metric time and resources). Next, we focus in more detail on the data dimension.

When it comes to data, two main lines of research can be identified. The first one deals with standard "case-centric" processes extended with event and case data. The second one focuses instead on "multi-case" processes, wherein constraints are expressed over multiple objects and their mutual relations. We briefly discuss each line separately.

Declarative Process Specifications with Event/Case Data. Within a process, activities may be equipped with data attributes that, at execution time, are grounded to actual data values by the involved resources. This means that events witnessing the occurrence of task instances come with a data payload. In addition, each process instance may evolve its own case data in response to the execution of activities.[3] Such case data may be stored in different ways, e.g., as key-value pairs or a full-fledged relational database. In this setting, it becomes crucial to extend DECLARE with so-called *data-aware constraints*, that is, constraints enriched with data-aware conditions over activities. The simple but illustrative example described next motivates why this is needed.

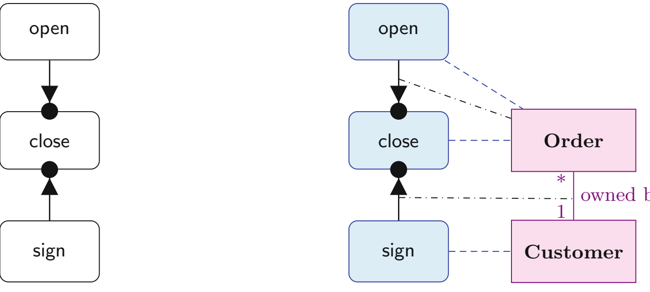
Example 18. We focus on a process where payments are issued by customers through a pay activity, which comes with an attribute indicating the paid amount, in Euros. Two consequent activities check and emit are executed to respectively inspect a payment and emit a receipt.

Let a log for this process contain multiple repetitions of the following traces:

$$t_1 = \langle \mathsf{pay(amount{=}50)}, \mathsf{emit} \rangle \qquad t_2 = \langle \mathsf{pay(amount{=}300)}, \mathsf{check}, \mathsf{emit} \rangle$$
$$t_3 = \langle \mathsf{pay(amount{=}20)} \rangle \qquad t_4 = \langle \mathsf{pay(amount{=}100)}, \mathsf{emit}, \mathsf{check} \rangle$$
$$t_5 = \langle \mathsf{pay(amount{=}90)}, \mathsf{emit} \rangle \qquad t_6 = \langle \mathsf{pay(amount{=}800)}, \mathsf{check} \rangle$$

One may wonder whether RESPONSE(pay, check) is a suitable constraint to explain (part of) the behaviour contained in the log. If considered unrestrictedly, this

[3] For conciseness of presentation, we will not distinguish between event and case data in our discussion, but technically they pose different, albeit tightly related, requirements.



(a) Conventional Declare specification (b) Object-centric Declare specification

Fig. 14. Comparison of conventional vs object-centric Declare.

is not the case, as there are many traces where payment is not followed by any inspection. The situation changes completely if one restricts the scope of the constraint activation only to those payments that involve an amount of 100 or more. ◁

A number of works has brought forward combined techniques to discover Declare constraints equipped with various forms of data conditions [54,60,86], to check conformance for data-aware constraints [12,13], and to handle their monitoring [5,69]. This passage has to be carried out with extreme care, as combining event data and time quickly leads to undecidability of reasoning [14, 34,35]. Therefore, such techniques have to operate in a limited fashion or suitably controlling the expressiveness of data conditions and the way they interact with time.

Object-Centric Declarative Process Specifications. So far, we have discussed the extension of Declare with event or case data. In a more general setting, data may refer to more complex networks of objects and their mutual relations, simultaneously co-evolved by one or multiple processes. In this type of processes, known under the umbrella term of *object-centric processes*, there is no single, pre-defined notion of case, and process executions cannot consequently be represented as flat traces, but call for richer representations (cf. [43]). The following example illustrates why Declare, in its conventional version, cannot be used to capture *object-centric* processes.

Example 19. Consider the fragment of an order-to-cash process, containing three activities: sign (indicating the signature of a GDPR form by the customer), open (the opening of an order), and close (the closing of an order). Two constraints apply to close, defining under which conditions it becomes executable:

- An order can be closed only if *that order* has been opened before.
- An order can be closed only if *its owner* has signed the consent before.

Figure 14(a) shows how these two constraints can be captured in conventional DECLARE. This specification is satisfactory only in the case where each trace refers to a single customer and a single order by that customer. For example, consider the following two traces, respectively referring to an order o_1 by Anne, and an order o_2 by Bob:

$$t_1 = \langle \mathsf{sign}, \mathsf{open}, \mathsf{close} \rangle \qquad\qquad t_2 = \langle \mathsf{open}, \mathsf{close}, \mathsf{sign} \rangle$$

Clearly, t_1 is a model trace, while t_2 is not, as the latter violates PRECEDENCE(sign, close).

However, one may need to consider multiple orders owned by the same or distinct customers, in the common situation where distinct orders may be later bundled together to handle their shipment. In our example, assuming that o_1 and o_2 are later bundled together in a shipment, this would require to combine t_1 and t_2 in a single object-centric trace, suitably extending each event with a reference to the object(s) it operates on. Suppose this would result into:

$$t = \left\langle \begin{array}{l} \mathsf{sign(customer{=}Anne)}, \mathsf{open(order{=}o2)}, \mathsf{open(order{=}o1)}, \\ \mathsf{close(order{=}o1)}, \mathsf{close(order{=}o2)}, \mathsf{sign(customer{=}Bob)} \end{array} \right\rangle$$

The DECLARE specification of Fig. 14(a) becomes now inadequate. In fact, it cannot distinguish which events actually *co-refer* to one another and which do not, so it cannot identify that the first signature by Anne refers to the first occurrence of close, but not to the second one. Hence, it wrongly uses the first occurrence of sign to satisfy PRECEDENCE(sign, close) for both orders. ◁

Fixing the issue described in Example 19 requires the explicitly extension of DECLARE with the ability of expressing how events relate to objects, how objects relate to each other, and in turn to *scope* the application of constraints, expressing that they must be enforced over events that suitably co-refer to each other – either because they operate on the same object, or because they operate on related objects. In our running example, this would call for the following actions:

- introduce the classes of **Order** and **Customer**;
- capture that there is a many-to-one **owned by** association linking orders to customers;
- indicate that sign refers to a customer, and that open and close refer to an order;
- scope PRECEDENCE(open, close) by enforcing that the two involved activities must *co-refer to the same order* (i.e., that an event of activity close for order o can only occur if an event of activity open has previously occurred *for the same order*);
- scope PRECEDENCE(sign, close) by enforcing that the two involved activities must respectively operate with a customer and an order that *co-refer through the **owned by** association* (i.e., that an event of activity close for order o can only occur if an event of activity sign has previously occurred *for the customer who owns* o).

Object-centric behavioral constraints (OCBC) [93] have been brought forward to handle this type of scoping through the integration of Declare specifications and UML class diagrams. Figure 14(b) shows the OCBC specification correctly capturing the constraints of Example 19. The approach is still at its infancy: some first seminal works have been conducted to handle discovery of OCBC specifications from object-centric event logs recording full database transactions [55], and to formalize and reason upon OCBC specifications through temporal description logics [7]. Further research is being carried out to improve the performance of discovery and frame it in the context of object-centric event logs of the form of [1], and to tackle conformance checking and monitoring. This is particularly challenging, as integrating temporal constraints with data models quickly leads to undecidability [7].

7 Conclusion

Throughout this chapter, we have thoroughly reviewed the declarative approach to process specification and mining. The declarative approach aims at limiting the process behavior by defining the boundaries within which its executions can unfold, yet leaving process executors free to explore at runtime which specific executions are generated. This is in contrast with the imperative approach, where process models compactly depict all and only those traces that are admissible. In fact, notice that different (imperative) process models can comply with the same declarative specification, just like different dynamic systems can model ($\models$) a set of temporal rules. In the chapter, we have grounded our discussion on the Declare language, but the introduced concepts are broad enough to be seamlessly applicable to other related approaches.

Specifically, we have first discussed how declarative process specifications can be formalized using Linear Temporal Logic on Finite Traces (LTL_f), and in turn operationally characterized in terms finite state automata (FSAs) for their execution semantics. On this solid formal ground, we have examined the core reasoning tasks that relate to declarative specifications and then delved deeper into the discovery and monitoring of processes according to the declarative paradigm. Interestingly, we have observed that the reasoning tasks are pervasive in all stages of declarative process mining, such as within discovery to avoid producing redundant or inconsistent outputs, and within monitoring to speculatively consider the possible future continuations of the monitored execution. In the last part of the chapter, we have provided a summary of the most recent advances in declarative process mining, focusing in particular on: *(i)* the applicability of declarative process mining techniques and concepts to full temporal logics, going beyond predefined patterns; *(ii)* the incorporation of uncertainty within constraints; *(iii)* the analysis of hybrid models integrating imperative and declarative fragments; *(iv)* multi-perspective constraints incorporating additional dimensions beyond the control-flow, and supporting the declarative specification of object-centric (multi-case) processes. This bird-eye view provides a fair account of the open research challenges in declarative process mining.

Acknowledgments. The authors want to thank Fabrizio Maria Maggi, Wil van der Aalst, Alessio Cecconi, Federico Chesani, Giuseppe De Giacomo, Riccardo De Masellis, Johannes De Smedt, Massimo Mecella, Paola Mello, Jan Mendling, Maja Pesic, Johannes Prescher for the long-standing cooperation and years of joint work that led to this chapter. The work of the authors has received funding by the Italian Ministry of University and Research under the PRIN programme, grant B87G22000450001 (PINPOINT). The work of C. Di Ciccio was partly funded by the Italian Ministry of University and Research under grant "Dipartimenti di eccellenza 2018–2022" of the Department of Computer Science at the Sapienza University of Rome and the Sapienza research project SPECTRA. The work of M. Montali was partly funded by the UNIBZ projects WineID, SMART-APP, QUEST, and VERBA.

References

1. van der Aalst, W.M.P.: Process mining: a 360 degrees overview. In: van der Aalst, W.M.P., Carmona, J. (eds.) Process Mining Handbook. LNBIP, vol. 448, pp. xx–yy. Springer, Cham (2022)
2. van der Aalst, W.M.P.: Foundations of process discovery. In: van der Aalst, W.M.P., Carmona, J. (eds.) Process Mining Handbook. LNBIP, vol. 448, pp. xx–yy. Springer, Cham (2022)
3. Adamo, J.-M.: Data Mining for Association Rules and Sequential Patterns - Sequential and Parallel Algorithms. Springer, New York (2001). https://doi.org/10.1007/978-1-4613-0085-4
4. Alman, A., Di Ciccio, C., Maggi, F.M., Montali, M., van der Aa, H.: RuM: declarative process mining, distilled. In: Polyvyanyy, A., Wynn, M.T., Van Looy, A., Reichert, M. (eds.) BPM 2021. LNCS, vol. 12875, pp. 23–29. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85469-0_3
5. Alman, A., Maggi, F.M., Montali, M., Patrizi, F., Rivkin, A.: Multi-model monitoring framework for hybrid process specifications. In: Franch, X., Poels, G. (eds.) Proceedings of the 34th International Conference on Advanced Information Systems Engineering (CAiSE 2022). Lecture Notes in Computer Science (2022, to appear)
6. Alman, A., Maggi, F.M., Montali, M., Peñaloza, R.: Probabilistic declarative process mining. Inf. Syst. (2012, to appear)
7. Artale, A., Kovtunova, A., Montali, M., van der Aalst, W.M.P.: Modeling and reasoning over declarative data-aware processes with object-centric behavioral constraints. In: Hildebrandt, T., van Dongen, B.F., Röglinger, M., Mendling, J. (eds.) BPM 2019. LNCS, vol. 11675, pp. 139–156. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26619-6_11
8. Back, C.O., Slaats, T., Hildebrandt, T.T., Marquard, M.: Discover: Accurate & efficient discovery of declarative process models. CoRR, abs/2005.10085 (2020)
9. Baier, T., Di Ciccio, C., Mendling, J., Weske, M.: Matching events and activities by integrating behavioral aspects and label analysis. Softw. Syst. Model. **17**(2), 573–598 (2018)
10. Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015)
11. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14:1–14:64 (2011)

12. Bergami, G., Maggi, F.M., Marrella, A., Montali, M.: Aligning data-aware declarative process models and event logs. In: Polyvyanyy, A., Wynn, M.T., Van Looy, A., Reichert, M. (eds.) BPM 2021. LNCS, vol. 12875, pp. 235–251. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85469-0_16
13. Burattin, A., Maggi, F.M., Sperduti, A.: Conformance checking based on multi-perspective declarative process models. Expert Syst. Appl. **65**, 194–211 (2016)
14. Calvanese, D., De Giacomo, G., Montali, M., Patrizi, F.: Verification and monitoring for first-order LTL with persistence-preserving quantification over finite and infinite traces. In: De Raedt, L. (ed.) Proceedings of the 31st International Joint Conference on Artificial Intelligence (IJCAI 2022). ijcai.org (2022, to appear)
15. Camacho, A., McIlraith, S.A.: Learning interpretable models expressed in linear temporal logic. In: Benton, J., Lipovetzky, N., Onaindia, E., Smith, D.E., Srivastava, S. (eds.) Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2018), pp. 621–630. AAAI Press (2019)
16. Carmona, J., van Dongen, B., Weidlich, M.: Conformance checking: foundations, milestones and challenges. In: van der Aalst, W.M.P., Carmona, J. (eds.) Process Mining Handbook. LNBIP, vol. 448, pp. xx–yy. Springer, Cham (2022)
17. Cecconi, A., De Giacomo, G., Di Ciccio, C., Mendling, J.: A temporal logic-based measurement framework for process mining. In: van Dongen et al. [92]
18. Cecconi, A., Di Ciccio, C., De Giacomo, G., Mendling, J.: Interestingness of traces in declarative process mining: the janus LTLp_f approach. In: Weske, M., Montali, M., Weber, I., vom Brocke, J. (eds.) BPM 2018. LNCS, vol. 11080, pp. 121–138. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98648-7_8
19. Chesani, F., et al.: Process discovery on deviant traces and other stranger things. CoRR, abs/2109.14883 (2021)
20. Chesani, F., Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Exploiting inductive logic programming techniques for declarative process mining. In: Jensen, K., van der Aalst, W.M.P. (eds.) Transactions on Petri Nets and Other Models of Concurrency II. LNCS, vol. 5460, pp. 278–295. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00899-3_16
21. Chesani, F., Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Exploiting inductive logic programming techniques for declarative process mining. Trans. Petri Nets Other Model. Concurr. **2**, 278–295 (2009)
22. Chomsky, N., Miller, G.A.: Finite state languages. Inf. Control **1**(2), 91–112 (1958)
23. Corea, C., Deisen, M., Delfmann, P.: Resolving inconsistencies in declarative process models based on culpability measurement. In: Ludwig, T., Pipek, V. (eds.) WI, pp. 139–153. University of Siegen, Germany/AISeL (2019)
24. Corea, C., Delfmann, P.: Quasi-inconsistency in declarative process models. In: Hildebrandt, T., van Dongen, B.F., Röglinger, M., Mendling, J. (eds.) BPM 2019. LNBIP, vol. 360, pp. 20–35. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26643-1_2
25. Corea, C., Nagel, S., Mendling, J., Delfmann, P.: Interactive and minimal repair of declarative process models. In: Polyvyanyy, A., Wynn, M.T., Van Looy, A., Reichert, M. (eds.) BPM 2021. LNBIP, vol. 427, pp. 3–19. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85440-9_1
26. Davulcu, H., Kifer, M., Ramakrishnan, C.R., Ramakrishnan, I.V.: Logic based modeling and analysis of workflows. In: PODS, pp. 25–33. ACM (1998)
27. De Giacomo, G., De Masellis, R., Grasso, M., Maggi, F.M., Montali, M.: Monitoring business metaconstraints based on LTL and LDL for finite traces. In: Sadiq, S.,

Soffer, P., Völzer, H. (eds.) BPM 2014. LNCS, vol. 8659, pp. 1–17. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10172-9_1
28. De Giacomo, G., De Masellis, R., Maggi, F.M., Montali, M.: Monitoring constraints and metaconstraints with temporal logics on finite traces. ACM Trans. Softw. Eng. Methodol. (2022, to appear)
29. De Giacomo, G., De Masellis, R., Montali, M.: Reasoning on LTL on finite traces: insensitivity to infiniteness. In: Brodley, C.E., Stone, P. (eds.) AAAI, pp. 1027–1033. AAAI Press (2014)
30. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: Rossi, F. (ed.) IJCAI, pp. 854–860. IJCAI/AAAI (2013)
31. De Leoni, M., Maggi, F.M., van der Aalst, W.M.: An alignment-based framework to check the conformance of declarative process models and to preprocess event-log data. Inf. Syst. **47**, 258–277 (2015)
32. De Smedt, J., De Weerdt, J., Serral, E., Vanthienen, J.: Discovering hidden dependencies in constraint-based declarative process models for improving understandability. Inf. Syst. **74**(Part 1), 40–52 (2018)
33. De Smedt, J., De Weerdt, J., Vanthienen, J., Poels, G.: Mixed-paradigm process modeling with intertwined state spaces. Bus. Inf. Syst. Eng. **58**(1), 19–29 (2016)
34. Demri, S., Lazic, R.: LTL with the freeze quantifier and register automata. ACM Trans. Comput. Log. **10**(3), 16:1–16:30 (2009)
35. Demri, S., Lazic, R., Nowak, D.: On the freeze quantifier in constraint LTL: decidability and complexity. Inf. Comput. **205**(1), 2–24 (2007)
36. Di Ciccio, C.: On the mining of artful processes. Ph.D. thesis, SAPIENZA, University of Rome, October 2013
37. Di Ciccio, C., Bernardi, M.L., Cimitile, M., Maggi, F.M.: Generating event logs through the simulation of declare models. In: Barjis, J., Pergl, R., Babkin, E. (eds.) EOMAS 2015. LNBIP, vol. 231, pp. 20–36. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24626-0_2
38. Di Ciccio, C., Maggi, F.M., Montali, M., Mendling, J.: Resolving inconsistencies and redundancies in declarative process models. Inf. Syst. **64**, 425–446 (2017)
39. Di Ciccio, C., Maggi, F.M., Montali, M., Mendling, J.: On the relevance of a business constraint to an event log. Inf. Syst. **78**, 144–161 (2018)
40. Di Ciccio, C., Mecella, M.: On the discovery of declarative control flows for artful processes. ACM Trans. Manag. Inf. Syst. **5**(4), 24:1–24:37 (2015)
41. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Boehm, B.W., Garlan, D., Kramer, J. (eds.) ICSE, pp. 411–420. ACM (1999)
42. Elgammal, A., Turetken, O., van den Heuvel, W.-J., Papazoglou, M.: Formalizing and appling compliance patterns for business process compliance. Softw. Syst. Model. **15**(1), 119–146 (2014). https://doi.org/10.1007/s10270-014-0395-3
43. Fahland, D.: Process mining over multiple behavioral dimensions with event knowledge graphs. In: van der Aalst, W.M.P., Carmona, J. (eds.) Process Mining Handbook. LNBIP, vol. 448, pp. xx–yy. Springer, Cham (2022)
44. Fionda, V., Greco, V.: LTL on finite and process traces: complexity results and a practical reasoner. J. Artif. Intell. Res. **63**, 557–623 (2018)
45. Fionda, V., Guzzo, A.: Control-flow modeling with declare: behavioral properties, computational complexity, and tools. IEEE Trans. Knowl. Data Eng. **32**(5), 98–911 (2020)
46. Goedertier, S., Martens, D., Vanthienen, J., Baesens, B.: Robust process discovery with artificial negative events. J. Mach. Learn. Res. **10**, 1305–1340 (2009)

47. Green, T.R.G., Petre, M.: Usability analysis of visual programming environments: a 'cognitive dimensions' framework. Vis. Comp. and Lang. **7**(2), 131–74 (1996)
48. Haisjackl, C., et al.: Understanding declare models: strategies, pitfalls, empirical results. Softw. Syst. Model. **15**(2), 325–352 (2016)
49. Hildebrandt, T.T., Mukkamala, R.R.: Declarative event-based workflow as distributed dynamic condition response graphs. In: PLACES. EPTCS, vol. 69, pp. 59–73 (2010)
50. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley Longman Publishing Co. Inc., Boston (2006)
51. Kupferman, O., Vardi, M.Y.: Vacuity detection in temporal model checking. Int. J. Softw. Tools Technol. Transfer **4**(2), 224–233 (2003)
52. Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Inducing declarative logic-based models from labeled traces. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 344–359. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75183-0_25
53. Lemieux, C., Park, D., Beschastnikh, I.: General LTL specification mining (T). In: Cohen, M.B., Grunske, L., Whalen, M. (eds.) 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, 9–13 November 2015, pp. 81–92. IEEE Computer Society (2015)
54. Leno, V., Dumas, M., Maggi, F.M., La Rosa, M., Polyvyanyy, A.: Automated discovery of declarative process models with correlated data conditions. Inf. Syst. **89**, 101482 (2020)
55. Li, G., de Carvalho, R.M., van der Aalst, W.M.P.: Automatic discovery of object-centric behavioral constraint models. In: Abramowicz, W. (ed.) BIS 2017. LNBIP, vol. 288, pp. 43–58. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59336-4_4
56. Lichtenstein, O., Pnueli, A., Zuck, L.: The glory of the past. In: Parikh, R. (ed.) Logic of Programs 1985. LNCS, vol. 193, pp. 196–218. Springer, Heidelberg (1985). https://doi.org/10.1007/3-540-15648-8_16
57. Ly, L.T., Maggi, F.M., Montali, M., Rinderle-Ma, S., van der Aalst, W.M.P.: Compliance monitoring in business processes: functionalities, application, and tool-support. Inf. Syst. **54**, 209–234 (2015)
58. Maggi, F.M., Bose, R.P.J.C., van der Aalst, W.M.P.: Efficient discovery of understandable declarative process models from event logs. In: Ralyté, J., Franch, X., Brinkkemper, S., Wrycza, S. (eds.) CAiSE 2012. LNCS, vol. 7328, pp. 270–285. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31095-9_18
59. Maggi, F.M., Di Ciccio, C., Di Francescomarino, C., Kala, T.: Parallel algorithms for the automated discovery of declarative process models. Inf. Syst. **74**, 136–152 (2018)
60. Maggi, F.M., Dumas, M., García-Bañuelos, L., Montali, M.: Discovering data-aware declarative process models from event logs. In: Daniel, F., Wang, J., Weber, B. (eds.) BPM 2013. LNCS, vol. 8094, pp. 81–96. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40176-3_8
61. Maggi, F.M., Montali, M., Peñaloza, R.: Temporal logics over finite traces with uncertainty. In: Proceedings of the 34 AAAI Conference on Artificial Intelligence (AAAI 2020), pp. 10218–10225. AAAI Press (2020)
62. Maggi, F.M., Montali, M., Peñaloza, R., Alman, A.: Extending temporal business constraints with uncertainty. In: Fahland, D., Ghidini, C., Becker, J., Dumas, M. (eds.) BPM 2020. LNCS, vol. 12168, pp. 35–54. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58666-9_3

63. Maggi, F.M., Montali, M., van der Aalst, W.M.P.: An operational decision support framework for monitoring business constraints. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 146–162. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28872-2_11
64. Maggi, F.M., Montali, M., Westergaard, M., van der Aalst, W.M.P.: Monitoring business constraints with linear temporal logic: an approach based on colored automata. In: Rinderle-Ma et al. [81], pp. 132–147
65. Maggi, F.M., Mooij, A.J., van der Aalst, W.M.P.: User-guided discovery of declarative process models. In: CIDM, pp. 192–199. IEEE (2011)
66. Maggi, F.M., Westergaard, M., Montali, M., van der Aalst, W.M.P.: Runtime verification of LTL-based declarative process models. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 131–146. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_11
67. Montali, M.: Specification and verification of declarative open interaction models - a logic-based framework. Ph.D. thesis, University of Bologna, Italy (2009)
68. Montali, M.: Specification and Verification of Declarative Open Interaction Models: a Logic-Based Approach. Lecture Notes in Business Information Processing, vol. 56. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14538-4
69. Montali, M., Maggi, F.M., Chesani, F., Mello, P., van der Aalst, W.M.P.: Monitoring business constraints with the event calculus. ACM TIST **5**(1), 17:1–17:30 (2013)
70. Montali, M., Pesic, M., van der Aalst, W.M.P., Chesani, F., Mello, P., Storari, S.: Declarative specification and verification of service choreographies. TWEB **4**(1), 1–62 (2010)
71. Montali, M., et al.: Verification from declarative specifications using logic programming. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 440–454. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89982-2_39
72. Mulyar, N., Pesic, M., van der Aalst, W.M.P., Peleg, M.: Declarative and procedural approaches for modelling clinical guidelines: addressing flexibility issues. In: ter Hofstede, A., Benatallah, B., Paik, H.-Y. (eds.) BPM 2007. LNCS, vol. 4928, pp. 335–346. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78238-4_35
73. Munoz-Gama, J., Martin, N., et al.: Process mining for healthcare: characteristics and challenges. J. Biomed. Inform. **127**, 103994 (2022)
74. Ouaknine, J., Worrell, J.: On the decidability and complexity of metric temporal logic over finite words. Log. Methods Comput. Sci. **3**(1) (2007)
75. Pesic, M.: Constraint-based workflow management systems: shifting control to users. Ph.D. thesis, Technische Universiteit Eindhoven (2008)
76. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: DECLARE: full support for loosely-structured processes. In: EDOC, pp. 287–300 (2007)
77. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: DECLARE: full support for loosely-structured processes. In: EDOC, pp. 287–300. IEEE Computer Society (2007)
78. Pesic, M., van der Aalst, W.M.P.: A declarative approach for flexible business processes management. In: Eder, J., Dustdar, S. (eds.) BPM 2006. LNCS, vol. 4103, pp. 169–180. Springer, Heidelberg (2006). https://doi.org/10.1007/11837862_18
79. Pill, I., Quaritsch, T.: Behavioral diagnosis of LTL specifications at operator level. In: Rossi, F. (ed.) Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013), pp. 1053–1059. IJCAI/AAAI (2013)

80. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57. IEEE (1977)
81. Rabin, M.O., Scott, D.S.: Finite automata and their decision problems. IBM J. Res. Dev. **3**(2), 114–125 (1959)
82. Raha, R., Roy, R., Fijalkow, N., Neider, D.: Scalable anytime algorithms for learning fragments of linear temporal logic. In: Fisman, D., Rosu, G. (eds.) TACAS 2022. LNCS, vol. 13243, pp. 263–280. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_14
83. Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30409-5
84. Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.): Business Process Management. LNCS, vol. 6896. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23059-2
85. Sadiq, S., Sadiq, W., Orlowska, M.: Pockets of flexibility in workflow specification. In: S.Kunii, H., Jajodia, S., Sølvberg, A. (eds.) ER 2001. LNCS, vol. 2224, pp. 513–526. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45581-7_38
86. Schönig, S., Di Ciccio, C., Maggi, F.M., Mendling, J.: Discovery of multi-perspective declarative process models. In: Sheng, Q.Z., Stroulia, E., Tata, S., Bhiri, S. (eds.) ICSOC 2016. LNCS, vol. 9936, pp. 87–103. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46295-0_6
87. Schunselaar, D.M.M., Slaats, T., Maggi, F.M., Reijers, H.A., van der Aalst, W.M.P.: Mining hybrid business process models: a quest for better precision. In: Abramowicz, W., Paschke, A. (eds.) BIS 2018. LNBIP, vol. 320, pp. 190–205. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93931-5_14
88. Singh, M.P.: Distributed enactment of multiagent workflows: temporal logic for web service composition. In: AAMAS, pp. 907–914. ACM (2003)
89. Slaats, T., Debois, S., Back, C.O.: Weighing the pros and cons: process discovery with negative examples. In: Polyvyanyy, A., Wynn, M.T., Van Looy, A., Reichert, M. (eds.) BPM 2021. LNCS, vol. 12875, pp. 47–64. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85469-0_6
90. Slaats, T., Schunselaar, D.M.M., Maggi, F.M., Reijers, H.A.: The semantics of hybrid process models. In: Debruyne, C., et al. (eds.) OTM 2016. LNCS, vol. 10033, pp. 531–551. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48472-3_32
91. Sun, Y., Su, J.: Conformance for DecSerFlow constraints. In: Franch, X., Ghose, A.K., Lewis, G.A., Bhiri, S. (eds.) ICSOC 2014. LNCS, vol. 8831, pp. 139–153. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45391-9_10
92. van der Aalst, W.M.P.: Process Mining - Data Science in Action, 2nd edn. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49851-4
93. van der Aalst, W.M.P., Artale, A., Montali, M., Tritini, S.: Object-centric behavioral constraints: integrating data and declarative process modelling. In: Artale, A., Glimm, B., Kontchakov, R. (eds.) DL. CEUR Workshop Proceedings, vol. 1879. CEUR-WS.org (2017)
94. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: towards a truly declarative service flow language. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 1–23. Springer, Heidelberg (2006). https://doi.org/10.1007/11841197_1
95. van Dongen, B.F., De Smedt, J., Di Ciccio, C., Mendling, J.: Conformance checking of mixed-paradigm process models. Inf. Syst. **102**, 101685 (2021)
96. van Dongen, B.F., Montali, M., Wynn, M.T. (eds.) 2nd International Conference on Process Mining, ICPM 2020, Padua, Italy, 4–9 October 2020. IEEE (2020)

97. Westergaard, M.: Better algorithms for analyzing and enacting declarative workflow languages using LTL. In: Rinderle-Ma et al. [81], pp. 83–98
98. Westergaard, M., Maggi, F.M.: Looking into the future. In: Meersman, R., et al. (eds.) OTM 2012. LNCS, vol. 7565, pp. 250–267. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33606-5_16
99. Zhu, S., Tabajara, L.M., Pu, G., Vardi, M.Y.: On the power of automata minimization in temporal synthesis. In: Proceedings 12th International Symposium on Games, Automata, Logics, and Formal Verification (GandALF 2021). EPTCS, vol. 346, pp. 117–134 (2021)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.