Customizable Declarative Process Specification Scenarios via Automated Planning

Alessandra Giovannetti[1], Giacomo Acitelli[1], Angelo Casciani[1], Claudio Di Ciccio[2], and Andrea Marrella[1]

Sapienza University of Rome, Italy,
{a.giovannetti,acitelli,casciani,marrella}@diag.uniroma1.it
[2] Utrecht University, The Netherlands, c.diciccio@uu.nl

Abstract. Declarative specifications define the behaviour of processes by means of temporal rules exerting dependencies between the occurrence of activities. Those rules can constitute the system requirements and normative boundaries within which the workflow is framed. The multitude of rules and their interrelations, though, make it reportedly difficult for users to linearly figure out unfoldings that depict the admitted behaviour in the form of explicative scenarios. While simulation engines can produce example traces of customizable length and in varying numerosity, these state-of-the-art tools adopt random approaches that do not permit the selection of a subset of runs exhibiting characteristic traits. In other words, simulations could produce traces that show common behavior, leaving out significant parts of the possible execution paths, and thus substantially hindering a proper validation of specifications. To mitigate this issue, we propose a principled solution that leverages automated planning to produce execution scenarios depicting the specified behavior according to desirable execution costs, diversifying the enacted behavior, or matching a given subsequence of steps. We demonstrate its efficiency and efficacy through experiments demonstrating its scalability and superiority in terms of behavioral coverage with respect to the state-of-the-art simulation tool for declarative process specifications.

Keywords: Automated planning · Simulation · Process science · Declare

1 Introduction

Declarative process specifications express conditions for the execution of activities based on given conditions. An example of such rules is Precedence (Send payment request,Fill in payment information), dictating that if payment information is filled in, then a payment request must have been sent at an earlier stage of the process. These specifications thus enable the definition of process behavior by means of rules that dictate what can, or cannot, be done, without explaining how [37]. Therefore, they are naturally apt for setting the boundaries to a process behavior [17], akin to the requirement specifications that are of interest to reactive systems in software engineering (SE) [27, Ch. 2] and more specifically specifications from a functionality standpoint [40]. Taking inspiration from the SE literature, then, process specifications can be considered as a means to elicit, or *discover* in more Business Process Management (BPM) words [20, Ch. 5], the behavior to be ultimately enacted in a process-aware information system based on domain expertise [2,1,36].

Pre-print copy of the manuscript published by Springer (available at link.springer.com) identified by DOI: 10.1007/978-3-032-37777-7_10

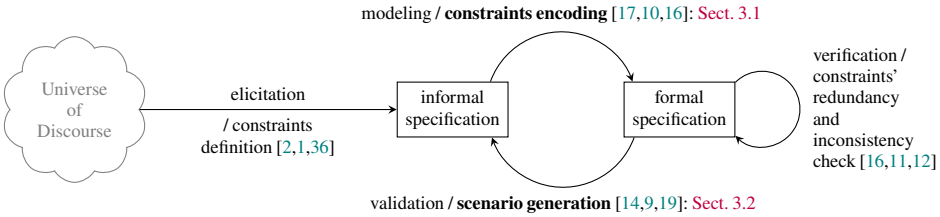


Fig. 1: The process specification cycle, reinterpreted from the *information modeling lifecycle* [23]. The phases considered in this paper are highlighted in bold.

However, the information modeling lifecycle in SE [23] and the quality assurance framework in BPM [20] remind us that the specification should not only be sound, which can be verified through automated techniques, but also validated, i.e., properly representing the desiderata of the domain experts and stakeholders once turned from a universe of discourse to formalized behavioral rules. For validation, declarative specifications alone are reportedly not suitable, because of their intrinsic visual complexity and the demanded cognitive effort to bear in mind multiple rules at once [21,26]. As a more effective approach, research work in BPM has proposed to let the domain experts validate and iteratively refine a specification through exemplary runs that it would accept [42,43,38].

To this end, research in SE resorts to *scenarios* [39], namely step-by-step executions derived from the provided specifications. Such scenarios are, among others, at the core of a well-known methodology for the specification of reactive systems releasing the user from handling formal representations, the play-in/play-out approach [28].

BPM research has provided noticeable contributions in the simulation of declarative process specifications [14,9,19], mainly with the aim of generating event logs to be later processed via process mining. The outputs are thus designed to incorporate a diverse array of possible traces, broad enough to cover an extensive spectrum of execution paths and put mining systems to the test in their capacity to rediscover the original models (the quest for "perfect logs" [9]). Scenarios, instead, serve a different purpose: to exemplify a few runs that are representative of the typical expected process behaviour.

In this paper, we propose an approach based on automated planning [24] tailored for the generation of declarative process specification scenarios. Figure 1 provides a graphical sketch of our investigation, pertaining to the encoding of specification constraints into a planning problem, and the generation of scenarios. To support scenario customization, our solution allows the user to optionally specify the patterns the scenarios should match (e.g., the occurrence of certain tasks or a sequence thereof). Also, we let the user indicate maximum execution costs for scenarios, their ranking, and numerosity. We experimentally demonstrate that our solution is applicable to real-world specifications and that the scenario generation problem remains tractable with inputs of growing size.

The remainder of the paper is organized as follows. Sect. 2 outlines the core notions underpinning our work and provides a running example scenario from the public administration domain. Sect. 3 presents our automata-based manipulation technique and shows how to reduce it to a planning problem. In Sect. 4, we evaluate the scalability of

Pre-print copy of the manuscript published by Springer (available at link.springer.com) identified by DOI: 10.1007/978-3-032-37777-7_10

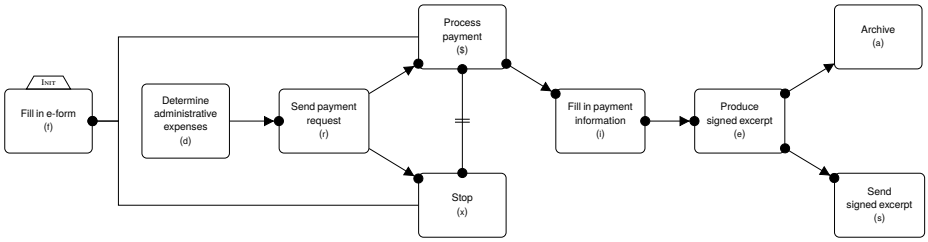


Fig. 2: A DECLARE specification, based on the case study of Schunselaar et al. [37]

our technique. In Sect. 5, we discuss related work. Finally, in Sect. 6 we conclude the paper, pinpoint the technique's limitations, and outline future work.

2 Running example and background

Figure 2 illustrates the running example and motivating scenario for our paper. It is inspired by the real-world specification introduced by Schunselaar et al. for their work on configurable process behavior [37]. The represented process excerpt pertains to the management of requests from the civil registration in a municipality. In particular, it denotes a *compliance model*, namely a set of rules stemming from expert knowledge, and made configurable to apply those in different contexts. The process behavior is there described with DECLARE, a declarative specification language for the representation of (temporally extended) constraints that must be satisfied throughout the process execution [17]. Table 1 illustrate some of those typical constraints. A DECLARE specification consists of a set of activities involved in a process and a collection of constraints defined over such activities. The orderings of activities are implicitly specified by these constraints, and anything that does not violate them is possible during execution.

The case study in Fig. 2 describes a municipal process beginning with the filling of a form (INIT (f)). Other necessary passages are the payment ($) or the stop (x) of the procedure. We express this aspect with a so-called branched constraint [15]:

Table 1: DECLARE constraints involved in the running example of Fig. 2

Notation	Constraint	Explanation
Init x	INIT(x)	x is the *first* to occur in a trace
x — y	RESPONDEDEXISTENCE(x,y)	If x occurs in the trace, then y occurs as well
x → y	RESPONSE(x,y)	If x occurs, then y eventually occurs after x in the trace
x → y	PRECEDENCE(x,y)	y occurs only if preceded by x in the trace
x → y	SUCCESSION(x,y)	x occurs if and only if it is followed by y
x ‖ y	NOTCOEXISTENCE(x,y)	x and y cannot both occur in the same trace

Pre-print copy of the manuscript published by Springer (available at link.springer.com) identified by DOI: 10.1007/978-3-032-37777-7_10

RespondedExistence(f,{$,x}), which means that if f occurs, $ *or* x should occur in the same trace. The NotCoExistence($,x) constraint additionally imposes that f and x are mutually exclusive, though. To take place, they require the sending of a payment request (Precedence(r,$), Precedence(r,x)) and, in turn, the payment request necessitates that administrative expenses are previously determined (Precedence(d,r)). If the payment is processed, then payment information should be filled in afterwards (Succession($,i)), and then a signed transaction excerpt produced (Succession(i,e)). Finally, the signed excerpt should later be archived (Response(e,a)) and sent via mail (Succession(e,s)).

Schunselaar et al. [37] presented an approach to automatically derive those models out of configurable specifications, in order to apply those models, upon adaptation, in different contexts (municipalities). Later research contributions showed that we can verify that those rules are not contradictory or redundant [16,11,12]. However, how can domain experts verify whether the rules adhere to the expected behaviour, even though they are satisfiable? Next, we describe the background information that our approach leverages to generate possible scenarios depicting the allowed unfoldings.

2.1 Representing declarative constraints with Declare

Declare constraints are template instantiations that define parameterized classes of properties. For example, the Response(a,b) template specifies that when a occurs, then b should eventually occur after a in a process execution.

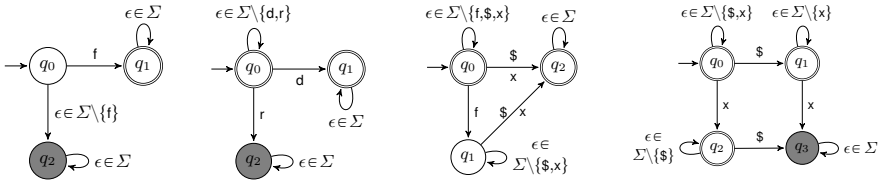
Every Declare constraint has a formal semantics based on Linear Temporal Logic over finite traces (LTL_f) [13], unambiguously characterising the notion of conforming process execution to a Declare specification. Since LTL_f is interpreted on finite execution traces, every LTL_f formula (hence every Declare constraint) can be captured as a *Deterministic Finite State Automaton* (DFA) recognizing all and only its conforming traces [13]. Formally, a DFA is a finite-state machine that accepts or rejects a given trace, by running it through a state sequence uniquely determined by the sequence of activities included in the trace [3]. Deterministic refers to the uniqueness of the computation run. In the following, we call *constraint automaton* a DFA used to encode a Declare constraint.

Definition 1 (Constraint Automaton). *A constraint automaton is a DFA* $\mathcal{A} = (\Sigma, Q^{\mathcal{A}}, q_0^{\mathcal{A}}, \delta^{\mathcal{A}}, F^{\mathcal{A}})$*, where:*

- Σ *is the* input alphabet*;*
- $Q^{\mathcal{A}}$ *is the (non-empty) finite set of* automaton states*;*
- $q_0^{\mathcal{A}} \in Q^{\mathcal{A}}$ *is the* initial state*;*
- $\delta^{\mathcal{A}} : Q^{\mathcal{A}} \times \Sigma \to Q^{\mathcal{A}}$ *is the* transition function*;*
- $F^{\mathcal{A}} \subseteq Q^{\mathcal{A}}$ *is the set of* final states.

Figure 3 depicts four DFAs for the encoding of Init (f), Precedence (d,r), RespondedExistence (f,{$,x}) and NotCoExistence ($,x), which are part of the Declare specification introduced in the running example. In the remainder of the paper, in DFA diagrams, we adopt the convention that double circles denote final states, while gray circles represent dead-end states. A DFA $\mathcal{A}$ reads in input sequences of symbols ("*string*") of its input alphabet. It starts in its initial state $q_0^{\mathcal{A}}$ and updates the state after having read each symbol via the state transition function $\delta^{\mathcal{A}}$. We say that a DFA $\mathcal{A}$ *accepts* a string π ($\pi \models \mathcal{A}$)

Pre-print copy of the manuscript published by Springer (available at link.springer.com) identified by DOI: 10.1007/978-3-032-37777-7_10



(a) Init(f) (b) Prec.(d,r) (c) Res.Ex(f,{$,x}) (d) NotCoEx.($,x)

Fig. 3: Example DFAs of Declare constraints.

if after reading it is in one of its accepting states (i.e., a state in $F^{\mathcal{A}}$), and otherwise we say that it *rejects* that string. The set of strings accepted by $\mathcal{A}$ is called the *language* of $\mathcal{A}$, i.e., $\mathcal{L}(\mathcal{A})$.

2.2 Automated Planning

Automated Planning (AP) is a research field focused on developing model-based autonomous solutions in Artificial Intelligence (AI) [24]. Planning systems (i.e., *planners*) are problem-solving algorithms that operate on explicit representations of states and actions. PDDL [22] is the standard Planning Domain Definition Language; it allows us to formulate a *planning model* with the description of the initial state of the world, the desired goal condition, and the planning domain.

A *planning domain* is built from a set of *propositions* describing the state of the world (i.e., the set of propositions that are true) and a set of *actions* Ω that can be executed. An *action schema* $a \in \Omega$ defines the list of *input parameters* for a, the *preconditions* under which a can be executed, and the *effects* of a on the state of the world. Both preconditions and effects are stated in terms of propositions in the planning domain, represented as *boolean predicates* and *numeric fluents*. While predicates represent facts about *domain objects* (i.e., things in the world that interest us) that can be true or false, fluents are number-valued functions that specify how the action increases the total cost of the plan being computed. The execution of actions can change the values of propositions in the planning domain, thereby transitioning the planning domain to a new state.

There exist several forms of AP in the literature (e.g., to capture durative and nondeterministic actions, etc.). In this paper, we focus on *classical planning* techniques characterized by *fully observable* and *deterministic* domains. To encode a classical planning problem, we use PDDL 2.1 [22], which is considered the most significant evolution over the original version of the language (called STRIPS) and is supported by the majority of state-of-the-art planners.[1] A solution to a planning problem is a sequence of operators—a *plan*—whose execution transforms the initial state into a state satisfying the goal by optimizing a pre-specified metric, generally related to the minimization of the plan's total cost.

From a computational perspective, classical planning is a PSPACE-complete problem [29]: finding a plan requires polynomial memory, although the computation time may still require exponential time in the worst case. Nonetheless, recent years have seen the emergence of classical planners with efficient heuristics that scale to large and challenging

[1] icaps-conference.org/index.php/main/competitions. Accessed: 1/6/2026.

Pre-print copy of the manuscript published by Springer (available at link.springer.com) identified by DOI: 10.1007/978-3-032-37777-7_10

problems across several Computer Science domains [35]. In this paper, we generate multiple candidate execution scenarios for a given declarative behaviour using three classical planners that produce distinct plans under different criteria: Top-K [33], which finds the k best plans by cost (without excluding lower-cost alternatives); Top-Quality [32], which returns all plans within a cost threshold; and Diverse [31], which generates maximally distinct plans (in terms of reordering of the plan's activities).

3 Planning-based Approach

Our approach targets to automatically generate execution scenarios ensuring compliance with (i) a declarative process specification and (ii) a set of user-defined requirements. We express these requirements as *preference constraints* on activity occurrence and their partial ordering, allowing users to customize the behavioural characteristics of the generated scenarios.

In this section, we first formalize the scenario generation problem and discuss how it can be solved using a DFA-based manipulation technique (Sect. 3.1). Then, we demonstrate how this technique reduces to a planning problem (Sect. 3.2).

3.1 Scenario generation problem

Let π be an *execution scenario*, i.e., a finite sequence of activities $\pi = \langle e_1,...,e_n \rangle$ over an alphabet Σ. We consider the following bipartite input for it:

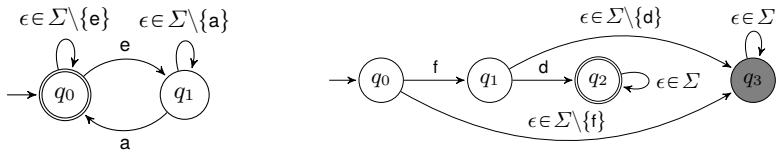
– A declarative specification $D = \{d_1,...,d_k\}$, consisting of a set of k Declare constraints, each specifying a behavioural requirement over the activities in Σ (e.g., Response, Precedence, Init, etc.).

– A set $P = \{p_1,...,p_m\}$ of m *preference constraints*, each expressing a user-defined requirement to constrain process execution. Specifically, building on the definition of relevant sequence patterns by Gusfield in [25], we consider five types of preference constraints: *prefix*, requiring the trace to start with a given sub-sequence of activities; *suffix*, requiring the trace to end with a given sub-sequence; *pattern*, requiring the trace to contain a given sub-sequence of activities; *pattern-gap*, requiring the trace to contain a given sub-sequence of activities not necessarily as consecutive elements, i.e., with possible activities between them; additionally, *min-len*, indicating the minimum trace length.

Notice that pattern (or pattern-gap) constraints with sub-sequences of length 1 (e.g., Pattern(e)) can be used to instruct the generator to include a given activity in the scenarios and, as a consequence, indicate that related constraints (e.g., Response(e,a)) must be triggered rather than only satisfied in a so-called *vacuous* manner [17] (that is, simply because e does not occur).

Each constraint in $D \cup P$ is translated into a corresponding DFA. Specifically, for each Declare constraint $d_i \in D$, we derive a *constraint automaton* $\mathcal{D}_i = (\Sigma, Q^{\mathcal{D}_i}, q_0^{\mathcal{D}_i}, \delta^{\mathcal{D}_i}, F^{\mathcal{D}_i})$ following standard DFA constructions for Declare semantics, as shown in Sect. 2. Similarly, for each preference constraint $p_j \in P$, we construct a *preference automaton* $\mathcal{P}_j = (\Sigma, Q^{\mathcal{P}_j}, q_0^{\mathcal{P}_j}, \delta^{\mathcal{P}_j}, F^{\mathcal{P}_j})$, defined as a DFA that characterizes a specific regular language by restricting the set of accepted traces according to conditions on their prefix, suffix, inclusion of a given sub-sequence, or minimum length.

Pre-print copy of the manuscript published by Springer (available at link.springer.com) identified by DOI: 10.1007/978-3-032-37777-7_10



(a) RESPONSE(e,a) (b) PREFIX(f,d)

Fig. 4: DFA encoding of RESPONSE(e,a) and PREFIX(f,d)

We illustrate the DFA construction with two examples built over our case study in Sect. 2. Figure 4(a) shows the DFA encoding the DECLARE template RESPONSE(e,a), which requires that every occurrence of activity e (Produce signed excerpt) is eventually followed by a (Archive). The automaton starts in the accepting state q_0 (the constraint is always satisfied if e never occurs); upon executing e, it moves to the non-accepting state q_1, from which it returns to q_0 only upon executing a. Any other activity leaves the automaton in its current state. Figure 4(b) shows the DFA encoding the PREFIX(f,d) preference constraint, which requires that the trace starts with the sub-sequence $\langle$f,d$\rangle$. The automaton advances through states q_0, q_1, and q_2 upon executing f (Fill in e-form) and then d (Determine administrative expenses) in this order; any deviation from this prefix leads to a non-accepting sink state q_3 from which no recovery is possible.

We refer to the *process frame* $\mathcal{F}$ as the collection of DFAs representing the DECLARE and preference constraints. We formally enunciate the scenario generation problem as follows.

Definition 2 (Scenario Generation). *Let $\mathcal{F} = \{\mathcal{D}_1,...,\mathcal{D}_k,\mathcal{P}_1,...,\mathcal{P}_m\}$ be a process frame that includes k constraint automata $\mathcal{D}_1,...,\mathcal{D}_k$ and m preference automata $\mathcal{P}_1,...,\mathcal{P}_m$. The generation of a scenario π over $\mathcal{F}$ consists of computing a sequence $\rho = \langle e_1 \cdots e_n \rangle \in \Sigma^*$ such that, for each $z = 0,...,n-1$, there exist:*

- *(in each $\mathcal{D}_i$, $1 \leqslant i \leqslant k$) a transition $(q_z^{\mathcal{D}_i}, e_{z+1}, q_{z+1}^{\mathcal{D}_i}) \in \delta^{\mathcal{D}_i}$, denoted as $q_z^{\mathcal{D}_i} \xrightarrow{e_{z+1}} q_{z+1}^{\mathcal{D}_i}$, such that $q_0^{\mathcal{D}_i} \xrightarrow{e_1} q_1^{\mathcal{D}_i} \cdots q_{n-1}^{\mathcal{D}_i} \xrightarrow{e_n} q_n^{\mathcal{D}_i}$, where $q_0^{\mathcal{D}_i}$ is the initial state of $\mathcal{D}_i$ and $q_n^{\mathcal{D}_i} \in F^{\mathcal{D}_i}$.*
- *(in each $\mathcal{P}_j$, $1 \leqslant j \leqslant m$) a transition $(q_z^{\mathcal{P}_j}, e_{z+1}, q_{z+1}^{\mathcal{P}_j}) \in \delta^{\mathcal{P}_j}$, denoted as $q_z^{\mathcal{P}_j} \xrightarrow{e_{z+1}} q_{z+1}^{\mathcal{P}_j}$, such that $q_0^{\mathcal{P}_j} \xrightarrow{e_1} q_1^{\mathcal{P}_j} \cdots q_{n-1}^{\mathcal{P}_j} \xrightarrow{e_n} q_n^{\mathcal{P}_j}$, where $q_0^{\mathcal{P}_j}$ is the initial state of $\mathcal{P}_j$ and $q_n^{\mathcal{P}_j} \in F^{\mathcal{P}_j}$.*

We say that $\mathcal{F}$ accepts *a scenario π ($\pi \models \mathcal{F}$) if there exists an execution ρ on the declarative automata $\mathcal{D}_1,...,\mathcal{D}_k$ and on the preference automata $\mathcal{P}_1,...,\mathcal{P}_m$ such that the last state of all these automata is final at the end of the execution.*

Therefore, a scenario π satisfies all declarative and preference constraints defined in a process frame $\mathcal{F}$ if and only if it is *accepted* by every DFA in $\mathcal{F}$; i.e., $\pi \in \mathcal{L}(\mathcal{D}_i)$ and $\pi \in \mathcal{L}(\mathcal{P}_j)$ for each $\mathcal{D}_i$ and $\mathcal{P}_j$ in $\mathcal{F}$. For example, considering the process frame specified in Fig. 4, any scenario that begins with the subsequence $\langle$f,d$\rangle$ and in which activity a occurs after the execution of e can be considered a valid scenario with respect to the frame.

3.2 Scenario generation as planning

In this section, based on the formalization presented above, we show how we can reduce the scenario generation problem to a deterministic planning problem. The idea behind

Pre-print copy of the manuscript published by Springer (available at link.springer.com) identified by DOI: 10.1007/978-3-032-37777-7_10

the reduction is to model the activities from the alphabet Σ as planning actions whose execution triggers state changes for each $\mathcal{D}_i$ and $\mathcal{P}_j$ included in a process frame $\mathcal{F}$, according to their respective transition function.

Planning actions are executable based on the current state of the DFAs: an action e can be executed only if the current states of each $\mathcal{D}_i$ and $\mathcal{P}_j$ have an outgoing transition labelled with the related process activity e. The problem then consists in finding a deterministic plan that takes such DFAs from their initial state to a final state, where actions corresponding to process activities are assigned a cost depending on a predefined cost function $\mathcal{C}$.

Inspired by common practice in conformance checking [6], we associate costs with activities. Here, costs represent penalties incurred when less preferred activities are included during scenario generation. For example, considering a process frame that includes the DFAs derived from the DECLARE specification in Fig. 2 and the Prefix constraint in Fig. 4(b), some valid scenarios may include the x (Stop) activity, which corresponds to stopping the payment procedure (e.g., $\pi_1 = \langle \mathsf{f,d,r,x} \rangle$), while others follow the standard payment flow (e.g., $\pi_2 = \langle \mathsf{f,d,r,e,\$,i,e,a,s} \rangle$). If interrupting the procedure is considered an undesirable yet still valid outcome, assigning a significantly higher cost to x than to the other activities (which may retain a standard unit cost) leads a cost-aware planning technique such as TOP-K [33] to prioritize the generation of scenarios that do not include the Stop activity (such as π_1) over those that do (such as π_2). We note that designing custom cost functions to weight individual activities is beyond the scope of this work, as it is inherently domain-specific and requires case-by-case customization. In contrast, our approach is intentionally domain-agnostic, focusing on a rigorous formulation of the scenario generation problem with broad applicability across diverse contexts. Interested readers can consult the work [5] for an in-depth exploration of this topic.

The obtained plan is a representation of an *execution scenario* that solves the generation problem. We provide the details of this reduction below.

A *deterministic planning domain with action costs* over a set of propositions Φ is a tuple $\mathcal{N} = (S, \Omega, cost, \tau)$, where:

- $S \subseteq 2^{\Phi}$ is the finite set of *domain states*, with each state $s = \{\phi_1, ..., \phi_n\}$ representing the propositional interpretation that assigns $true$ to a proposition ϕ if and only if $\phi \in s$;
- Ω is the finite set of *domain actions*;
- $cost : \Omega \mapsto \mathbb{N}_0$ is a *cost function*, assigning a non-negative cost to each planning action;
- $\tau : S \times \Omega \mapsto S$ is the domain *transition function*.

A *planning problem* is a tuple $\mathcal{R} = (\mathcal{N}, s_0, G)$, where $s_0 \in S$ is the *initial state* of the problem and G is a propositional formula over Φ representing the *goal* condition. A *plan* for $\mathcal{R}$ is a finite sequence of actions $\pi = e_1 \cdots e_n \in \Omega^*$ whose execution from s_0 leads to a state s_n such that $s_n \models G$. A solution π of $\mathcal{R}$ is said to be *optimal* if, for all other solutions π', it is the case that $cost(\pi) \leqslant cost(\pi')$.

We encode the scenario generation problem as a planning problem as follows. Let $\mathcal{F} = \{\mathcal{D}_1, ..., \mathcal{D}_k, \mathcal{P}_1, ..., \mathcal{P}_m\}$ be a process frame that includes k constraint automata $\mathcal{D}_1, ..., \mathcal{D}_k$ and m preference automata $\mathcal{P}_1, ..., \mathcal{P}_m$. The current state of each DFA is modelled through suitable propositions. We define the planning domain $\mathcal{N} = (S, \Omega, cost, \tau)$ over the set of propositions $\Phi = Q^{\mathcal{D}_1} \cup \cdots \cup Q^{\mathcal{D}_k} \cup Q^{\mathcal{P}_1} \cup \cdots \cup Q^{\mathcal{P}_m}$ (with all state sets assumed pairwise disjoint), where:

Pre-print copy of the manuscript published by Springer (available at link.springer.com) identified by DOI: 10.1007/978-3-032-37777-7_10

1. $S \subseteq \{\{\bigcup_{i=1,\ldots,k} q^{\mathcal{D}_i}\} \cup \{\bigcup_{j=1,\ldots,m} q^{\mathcal{P}_j}\} \mid q^{\mathcal{D}_i} \in Q^{\mathcal{D}_i}, q^{\mathcal{P}_j} \in Q^{\mathcal{P}_j}\}$ is the set of domain states, each containing exactly one state per DFA in $\mathcal{F}$;
2. $\Omega = \Sigma$, i.e., each activity $e \in \Sigma$ corresponds to a planning action, modeling the synchronous advancement of all DFAs upon executing e;
3. For all $e \in \Sigma$, $cost(e) = \mathcal{C}(e)$, i.e., it depends on the specific cost function $\mathcal{C}$;
4. For $e \in \Omega$, $q_a^{\mathcal{D}_i}, q_b^{\mathcal{D}_i} \in Q^{\mathcal{D}_i}$, $q_a^{\mathcal{P}_j}, q_b^{\mathcal{P}_j} \in Q^{\mathcal{P}_j}$, we define the domain transition function as $\tau(\{\bigcup_{i=1,\ldots,k} q_a^{\mathcal{D}_i}\} \cup \{\bigcup_{j=1,\ldots,m} q_a^{\mathcal{P}_j}\}, e) = \{\bigcup_{i=1,\ldots,k} q_b^{\mathcal{D}_i}\} \cup \{\bigcup_{j=1,\ldots,m} q_b^{\mathcal{P}_j}\}$ if:
   (a) $(q_a^{\mathcal{D}_i}, e, q_b^{\mathcal{D}_i}) \in \delta^{\mathcal{D}_i}$, for each $\mathcal{D}_i \in \mathcal{F}$;
   (b) $(q_a^{\mathcal{P}_j}, e, q_b^{\mathcal{P}_j}) \in \delta^{\mathcal{P}_j}$, for each $\mathcal{P}_j \in \mathcal{F}$.

We call $\mathcal{N}$ the *scenario generation planning domain* of $\mathcal{F}$. Since all automata are DFAs, the transition function τ is deterministic. Moreover, since $\Omega = \Sigma$, every plan for $\mathcal{N}$ directly corresponds to a scenario executed over Σ.

We then define the planning problem $\mathcal{R} = (\mathcal{N}, s_0, G)$, where:

- $s_0 = \{q_0^{\mathcal{D}_1}, \ldots, q_0^{\mathcal{D}_m}, \ldots, q_0^{\mathcal{P}_1}, \ldots, q_0^{\mathcal{P}_k}\}$ is the initial state, corresponding to the initial state of each DFA in $\mathcal{F}$;
- $G = (\bigvee_{q^i \in F^{\mathcal{D}_i}} q^i) \wedge (\bigvee_{q^j \in F^{\mathcal{P}_j}} q^j)$ is the goal condition, requiring every DFA to be simultaneously in an accepting state.

We call $\mathcal{R}$ the *scenario generation planning problem* of $\mathcal{F}$. In planning terms, finding a *plan* $\pi = e_1 \cdots e_n \in \Omega^*$ for $\mathcal{R}$ amounts to navigating the synchronous product of all DFAs from their joint initial state to a joint accepting state.

One of the major advantages of our planning-based formulation is that it naturally supports the generation of multiple and diverse scenarios by adopting different planning strategies. Depending on the specific generation objectives, various planners and parameter settings can be employed to control the scenario generation process, as discussed in the next section.

4 Evaluation

We have developed a planning-based tool in Python that implements the technique described in Sect. 3. The tool is available for reproducibility alongside the testbeds at: https://doi.org/10.5281/zenodo.20842315. Based on a set of DECLARE and preference constraints in input, and following the translation approach of Sect. 3.2, it converts such constraints into DFAs and generates the planning tasks in PDDL. To generate multiple scenarios, our tool leverages the FORBID-ITERATIVE classical planner,[2] which can produce multiple plans that vary depending on the underlying planning algorithm. In particular, we employ three well-known planning algorithms: TOP-K [33], DIVERSE [31], and TOP-QUALITY [32].

4.1 Experiments with Synthetic Specifications

To assess the scalability of our technique, we conducted two experiments with synthetic input specifications of increasing complexity. For both experiments, we used a unit-cost

[2] https://github.com/IBM/forbiditerative. Accessed: 1/6/2026.

Pre-print copy of the manuscript published by Springer (available at link.springer.com) identified by DOI: 10.1007/978-3-032-37777-7_10

Table 2: First-experiment results on synthetic specifications

	10 activities				20 activities			
	3 constraints	5 constraints	7 constraints	9 constraints	3 constraints	5 constraints	7 constraints	9 constraints
Grounding time	0.0017 s	0.0017 s	0.0028 s	0.0183 s	0.0021 s	0.0022 s	0.0028 s	0.0227 s
Planner	Average planning time [s]							
diverse-10	0.25	0.27	0.29	0.4	0.22	0.3	0.31	0.4
diverse-25	0.18	0.31	0.3	0.4	0.19	0.28	0.31	0.34
diverse-50	0.17	0.3	0.32	0.45	0.16	0.31	0.31	0.33
topk-10	0.2	0.21	$4.52 \cdot 10^{-2}$	0.41	0.19	0.24	$6.56 \cdot 10^{-2}$	0.45
topk-25	0.17	0.31	0.17	0.44	0.17	0.29	0.26	0.51
topk-50	0.15	0.32	0.25	0.61	0.16	0.31	0.27	0.71
topq-10	0.28	0.37	0.45	0.42	0.32	0.34	0.48	0.5
topq-15	0.32	0.42	0.48	0.49	0.35	0.39	0.53	0.6
topq-20	0.27	0.37	0.55	0.57	0.33	0.47	0.64	0.74

function for all planning actions, as varying action costs are unlikely to significantly affect performance. Conversely, differentiated costs could improve efficiency by reducing equivalent-cost alternatives and adding structure to the planner's search space. Under this assumption, cost-optimal plans coincide with shortest plans. Tests were run on a 12$^{\text{th}}$-gen. Intel Core i7 with 8 GB RAM.

In the first experiment, we evaluated the performance of our technique against scenario inputs of increasing size (see Sect. 3.1), namely specification and preference constraints. Different types of selected constraints were systematically added to assess their impact on the overall performance of the scenario generation task. In the second experiment, we evaluated the scalability of our technique by using a One-Factor-at-a-Time (OFAT) design. Specifically, we varied one input parameter at a time (i.e., the size of the activity alphabet, the number of constraints, or the number of generated scenarios) while keeping the other two fixed. This controlled setup allowed us to assess each parameter's impact independently.

Scalability w.r.t. scenario input size: Setup. We considered two activity alphabets, with sizes 10 and 20, respectively. For each alphabet, we incrementally built specifications with 3, 5, 7, and 9 constraints: the 3-constraint specification contains only Declare constraints; the 5-, 7-, and 9-constraint specifications contain 4, 5, and 6 Declare constraints and 1, 2, and 3 preference constraints, respectively. Constraints were added in this order: AtLeastOne (t0), Response (t1,t2), NotCoExistence (t3,t4), ChainPrecedence (t5,t4), Choice(t5,t6), and NotSuccession (t7,t8) as Declare constraints, and Pattern-Gap(t4,t6), Pattern(t7,t8), and Min-Len(8) as preference constraints. For Top-K and Diverse, we generated 10, 25, and 50 plans (i.e., scenarios) per test. Note that Top-K returns cost-optimal plans, while Diverse returns mutually distinct plans. For Top-Quality, we generated 20 plans per test with cost bounds of 10, 15, and 20.

Scalability Experiment w.r.t. Scenario Input Size: Results. Table 2 reports the results of the first experiment. For each test, we measured the *grounding time*, i.e., the preprocessing time needed to instantiate the PDDL specification into an explicit planning model that can be processed by the planner, and the average *planning time* required by the planner to explore that model and generate a scenario. The results suggest that for all three planning algorithms, increasing the number of constraints leads to a moderate increase in both grounding and planning time. However, this increase remains manageable

Pre-print copy of the manuscript published by Springer (available at link.springer.com) identified by DOI: 10.1007/978-3-032-37777-7_10

Table 3: OFAT experiment results on synthetic specifications

	15 constraints - 15 activities					15 constraints - 15 plans					15 activities - 15 plans				
	10 plans	**15 plans**	**20 plans**	**25 plans**	**30 plans**	**10 act.**	**15 act.**	**20 act.**	**25 act.**	**30 act.**	**10 const.**	**15 const.**	**20 const.**	**25 const.**	**30 const.**
Grounding time			0.0412 s			0.0699 s	0.0733 s	0.0822 s	0.0931 s	0.1067 s	0.0234 s	0.0567 s	0.1685 s	0.5865 s	1.9058 s
Planner	Average planning time [s]														
Diverse	0.31	0.31	0.32	0.32	0.32	0.25	0.26	0.28	0.3	0.3	0.3	0.39	1.01	1.73	3.97
Top-k	0.41	0.42	0.43	0.44	0.45	0.52	0.53	0.59	0.6	0.68	0.42	0.54	3.98	5.85	16.31
Top-q	0.37	0.38	0.39	0.4	0.42	0.3	0.31	0.32	0.32	0.32	0.39	0.5	10.54	15.38	68.21

across all configurations, indicating that the proposed approach scales well with respect to the number of constraints and can potentially support compliance specifications of considerable size. Moreover, the results also indicate that performance degradation depends on the number of constraints rather than their type, with no significant differences observed across the specific DECLARE and preference constraints considered.

A further observation concerns the number of plans/scenarios to be generated. The results show that this parameter does not significantly affect performance: the average time per plan remains largely constant across configurations, with no clear correlation between the number of requested plans and the time required to generate them. For instance, examining the results in Table 2 related to the configurations with an alphabet size of 20 activities, for both TOP-K and DIVERSE the per-plan generation time for 10, 25, and 50 plans does not vary substantially, confirming that the computational cost scales linearly with the number of plans rather than increasing exponentially. A similar behaviour is observed for TOP-QUALITY: the average time per plan remains largely stable as the cost bound increases from 10 to 15 and 20, suggesting that relaxing the quality threshold does not introduce significant computational overhead.

Finally, increasing the size of the alphabet also affects both grounding and planning time. However, the approach scales well along this dimension, with the observed increase in execution time remaining modest. Notably, the number of constraints has a stronger impact on performance than the size of the activity alphabet, suggesting that the complexity of the compliance specification is a more influential factor than the number of activities.

OFAT experiment: Setup. We adopted a controlled setup in which two parameters of the input configuration (e.g., number of constraints and activity alphabet size) were fixed at 15 (e.g., 15 constraints and 15 activities), while the third (e.g., number of generated plans) was varied over the values 10, 15, 20, 25, and 30. This yielded three experimental configurations, each isolating the effect of one parameter of interest. This allowed us to independently assess the impact of each parameter, namely (i) the number of generated plans, (ii) the size of the activity alphabet, and (iii) the number of constraints, on the performance of the three tested planning algorithms, while keeping the remaining parameters fixed.

OFAT experiment: Results. Table 3 shows the results of the second experiments. For each configuration, we report the grounding time and the average planning time per plan. Each experiment was repeated five times, and the reported time is the average across runs. Note that, in the first configuration, grounding time remains constant across different numbers of plans, as it depends only on the fixed number of activities and constraints. In the other two configurations, it increases with the number of activities or constraints, reflecting the growing complexity of the planning task. Regarding the average planning time, the results in Fig. 5 support the following observations. (I) In the first configuration (15 constraints, 15

Pre-print copy of the manuscript published by Springer (available at link.springer.com)
identified by DOI: 10.1007/978-3-032-37777-7_10

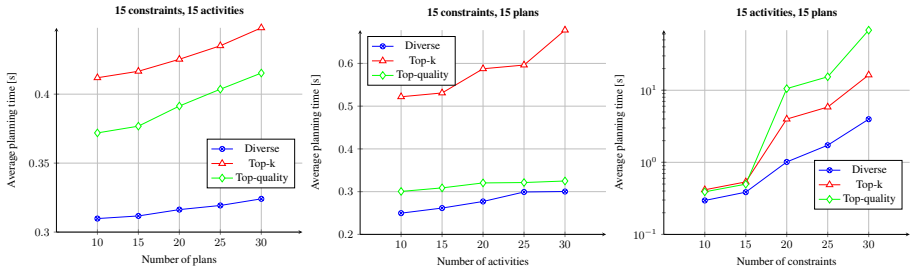


Fig. 5: Average planning time of the OFAT experiment

activities), the average planning time increases almost linearly with the number of generated plans, with DIVERSE being faster than TOP-K and TOP-QUALITY, which exhibit similar trends. (II) In the second configuration (15 constraints, 15 plans), planning times grow moderately when the number of activities increases, with DIVERSE emerging as the fastest planner. These results can be explained by the fact that a larger alphabet directly increases the number of transitions in the DFAs encoding the constraints, thereby expanding the state space that the planner must explore. (III) The third configuration (15 activities, 15 plans) further confirms the findings of the first experiment, namely that the number of constraints is the most influential factor affecting planning time. While variations in the number of activities or plans result in only modest changes, increasing the number of constraints causes a substantial increase in computation time across all three planning algorithms.

It is worth noting that among the three planning algorithms, DIVERSE demonstrates the best overall performance. This can be attributed to the fact that, unlike TOP-K and TOP-QUALITY, which balance both plan diversity and quality, DIVERSE focuses solely on generating mutually distinct plans without optimizing for cost. As a result, its search is less constrained and thus more efficient.

4.2 Experiments with real-world specifications

To further demonstrate the applicability of our approach, we tested it on four real-world declarative process specifications from the process science literature: *Municipality* [37], *Fracture Treatment* [14], *University Admission* [17], and *Surgery Process* [41]. Specifically, Municipality [37] is an extract of a process for requesting an excerpt from the civil registration. We considered three variants of this specification, namely *MunicipalityA*, *MunicipalityB*, and *MunicipalityC*, which differ in the number and type of constraints. Fracture Treatment [14] describes the clinical process involved in the management of bone fractures. University Admission [17] specifies the process of admitting students to a university program. Finally, Surgery Process [41] describes the pre-, intra-, and post-operative procedures for colorectal surgery.

For these real-world scenarios, we evaluated the scenario generation using TOP-K and TOP-QUALITY, as these cases call for scenarios that balance both diversity and quality. DIVERSE, while effective in terms of performance as shown in Sect. 4.1, does not account for plan costs and therefore may produce plans that, although distinct, do not reflect

Pre-print copy of the manuscript published by Springer (available at link.springer.com) identified by DOI: 10.1007/978-3-032-37777-7_10

Table 4: Experimental results for the real-world specifications

	Municipality A	Municipality B	Municipality C	Fracture treatment	University admission	Surgery process
Grounding time	0.0067 s	0.0236 s	0.0235 s	0.0040 s	0.1380 s	0.0142 s
Planner	Average planning time [s]					
topk-10	0.22	0.48	0.38	0.31	0.93	0.14
topk-25	0.18	0.41	0.19	0.31	0.58	$6.93 \cdot 10^{-2}$
topk-50	0.19	0.47	0.19	0.34	0.51	$4.94 \cdot 10^{-2}$
topq-10	0.64	0.48	0.44	0.29	0.78	0.24
topq-15	0.44	0.5	0.49	0.3	0.86	0.35
topq-20	0.43	0.57	0.54	0.32	0.87	0.42
Constraints	10	11	12	7	15	21

meaningful differences in process behavior. Moreover, in the considered real-world scenarios, the ordering of activities is a key dimension of diversity: plans that involve the same set of activities but differ in their execution order should be regarded as genuinely distinct. This is particularly evident in the Surgery Process case, where the sequencing of activities is clinically significant. For example, performing a medical treatment before a surgical intervention, or vice versa, can lead to substantially different clinical outcomes.

Table 4 summarizes our findings with the real-world use cases. The results confirm that our approach delivers scalable performance on real process specifications, with execution times in the order of a tenth of a second or less, despite the modest hardware infrastructure.

A comparative example. Beyond scalability, a key objective of the real-world evaluation is to demonstrate the practical utility of our approach in generating meaningful and realistic process scenarios. To this end, we compare our approach with MINERful [30], a well-known tool for declarative process discovery that also supports trace generation. MINERful allows the user to configure the minimum and maximum length of generated traces, without support for preference constraints or ranking criteria. Such a limitation can lead to the generation of traces that are semantically trivial or unrealistic, though formally compliant with the specified constraints. Next, we provide an exemplificative illustration of this statement, leaving a full-scale comparative evaluation of the effectiveness of planning-based scenario generations against randomized approaches for future work.

As an example, we take into account the municipality use case of Sect. 2. We make MINERful generate 100 traces (its default setting) of length ranging between 5 (the minimum number of activity occurrences required by the specification) and 20 (the uppermost value for the TOP-QUALITY parameter in our tests). The traces generated by MINERful contain repeated occurrences of the same activities, resulting in unrealistic scenarios. In contrast, our approach generates scenarios that are both compliant and avoid repetitions of activities that are vacuously satisfied by the DECLARE constraints. As an example, the first five scenarios produced by our tool have a significantly more contained length between 4 and 10 activities, 8.4 on average, compared to the 10 to 20 activities of MINERful, 16.2 on average. They simulate plausible runs as expected. As an example, one of the top-5 generated scenarios is the following: $\pi = \langle$Fill in e-form, Determine administrative expenses, Send payment request, Fill in payment information, Process payment, Produce signed excerpt, Archive, Send excerpt$\rangle$.

This scenario reflects a realistic execution of the municipality process, with activities occurring in a semantically coherent order and without unnecessary repetitions. In contrast,

Pre-print copy of the manuscript published by Springer (available at link.springer.com) identified by DOI: 10.1007/978-3-032-37777-7_10

the first two traces generated by our run of MINERful, with and without the knock-out occurrence of x respectively, are ⟨f, a, a, a, f, a, d, a, d, f, d, $, x⟩ and ⟨f, d, a, f, a, a, d, a, $, i, i, d, f, p, i, i, i, p, s, a⟩ using the abbreviations introduced in Sect. 2. The reader, irked by the need to filter out the unnecessary repetitions of tasks or navigating through so long sequences out of which few significant steps are considered (especially compared to the preceding scenario), can intuitively experience the feeling that motivated our research. To give a quantitative account of this aspect, we can consider the coverage of the generated runs for the constraints. Intuitively, coverage [8] measures the ratio of events triggering a constraint, like the occurrence of e for RESPONSE(e,a). Excluding INIT(f), which is trivially triggered by every initial event in a trace, the coverage of the constraints is 0.180 in the top-5 scenarios our tool generates (each constraint is triggered by 18 % of the events), whereas it halves to 0.092 with the 100 traces generated by MINERful (0.067 if we take the first five).

5 Related Work

Zugal et al. [42,43] empirically demonstrated the effectiveness that a test-driven approach has on the modeling and maintainability of DECLARE specifications, whereby test cases are exemplary traces against which a process specification is checked (akin to what we name scenarios in this paper). Slaats et al. [38] theorize the use of short, open-world-like test cases for the iterative modeling of DCR Graphs. These valuable contributions strengthen the argument motivating our work, as we aim to yield automated generators of such test cases.

Several approaches in the literature leverage computational logic and formal models to generate event logs from declarative process specifications. A seminal work in the area [14] simulates DECLARE models by translating constraints into regular expressions and DFAs, and generates logs compliant with a given model by navigating the product of automata derived from individual constraints. Building upon logic-based foundations, other authors have explored different paradigms. Chesani et al. [9] propose using abductive logic programming to generate both positive and negative traces (i.e., those violating the model) for DECLARE models. Similarly, Answer Set Programming (ASP) was adopted as a highly flexible technique for synthetic log generation [10]. Donadello et al. [19] explore the purpose-guided log generation via ASP, demonstrating high trace diversification, and Maggi et al. [34] propose a translation of DECLARE models augmented with data into Alloy programs.

While these declarative and procedural methods offer flexibility, they often face limitations when attempting to generate interpretable scenarios for model validation. State-of-the-art simulation engines typically adopt random approaches to trace generation. Consequently, they tend to produce traces that show common behavior while leaving out significant parts of the possible execution paths. On the other hand, solver-based approaches (i.e., ASP or SAT) heavily rely on internal search heuristics. Indeed, as noted by Donadello et al. [18], the problem of computing diverse solutions for combinatorial problems is intrinsically difficult: solvers often yield a relatively small number of solutions and require approximation techniques to operate within reasonable time bounds. Hence, these limitations might hinder the selection of a curated subset of runs exhibiting characteristic traits, which is central for the proper validation of complex declarative specifications. To mitigate these limitations, our technique formally encodes the scenario generation problem as an automated planning task. Unlike random simulation approaches or solver-dependent

Pre-print copy of the manuscript published by Springer (available at link.springer.com)
identified by DOI: 10.1007/978-3-032-37777-7_10

searches, our solution enables the customization and ranking of the generated scenarios. By navigating the synchronous product of DFAs through classical planning techniques encoded in PDDL 2.1, our approach allows users to tailor scenario generation to specific needs.

6 Discussion and Concluding Remarks

In this paper, we presented an AI planning-based approach to generate representative scenarios for declarative process specifications. To cater for succinct yet characterizing exemplary runs, we leverage three deterministic planning algorithms, namely TOP-K, TOP-QUALITY, and DIVERSE, where actions carry associated costs and valid plans must meet specific criteria. These algorithms facilitate the recommendation of alternative unfoldings of process behaviour within the conforming space defined by the process frame, guided by controlled criteria that shape both the quality and diversity of the generated plans.

A more general advantage of adopting planning for scenario generation is that it provides solutions that are correct, transparent, and scalable (three key requirements for trustworthy systems). Specifically, planning leverages mature reasoning mechanisms developed over decades of research to guarantee solution correctness [24]; it supports explainability through verification techniques such as Model Checking [4], enabling the analysis of relevant plan properties (for instance, explaining why a specific action was selected); despite its worst-case computational complexity (see Sect. 2.2), advances in domain-independent heuristics have made most practical problems, particularly in BPM and process mining, tractable and scalable in practice [35], as confirmed by our empirical evaluation. It is worth noting that our approach is applicable to the full DECLARE repertoire of templates and beyond (see, e.g., the preference constraints). In fact, our approach can handle any constraint expressible via DFAs, and DFAs have a strictly higher expressiveness than LTL_f [13], on which DECLARE is based. Therefore, its applicability includes and extends beyond DECLARE.

Our approach paves the path for future amelioration. First and foremost, its applicability in the field calls for an investigation with users into the most effective framings for scenario generation, in terms of preference automata and activity costs. To this end, the integration of a dedicated specification language for log creation is highly promising [18]. Also, we aim to design specific heuristics that make it applicable for process specifications discovered from event logs, which often consist of a larger number of constraints. In that regard, it becomes compelling to make use of process mining quality measures of the mined constraints, like confidence and support, to guide the generation of plans. Our reliance on classical planners is tailored to the generation of linear execution traces. While sound for scenario generation, this choice outlines a clear boundary if the execution context shifts to a runtime environment driven by uncontrollable exogenous events (e.g., unexpected customer cancellations). This would require coupling the approach with more expressive knowledge representation formalisms to natively manage non-deterministic control flows and exogenous events [7]. We also plan to study how the scenarios we generate can be effectively used for validation and refinement of process specifications, thus closing the cycle presented in Fig. 1. Inspired by the play-out approach of Harel and Marelly [28], we foresee as a highly interesting research avenue the inclusion of user interactions to guide the generation of scenarios. Although originally conceived for live sequence charts (LSCs), the authors advocate the adoption of their approach with "languages other than LSCs, such as appropriate variants of temporal

Pre-print copy of the manuscript published by Springer (available at link.springer.com) identified by DOI: 10.1007/978-3-032-37777-7_10

logic," on which DECLARE is indeed based. Finally, we aim to develop a comparative framework to empirically analyze the properties of scenarios generated by our planning-based approach against those produced by randomized simulation techniques, in order to better assess the utility of our approach with respect to existing techniques in the literature.

Acknowledgments. The work of A. Marrella has been supported by the Sapienza projects CAI4PPI and FOND-AIBPM, and the EU Horizon Europe Research and Innovation Programme under Grant 101299128 (euroFMX). A. Casciani conducted this work while enrolled in the National Doctorate in AI run by Sapienza (CUP B53C23003500006, Mission 4 NRRP Generic Research Scholarship, Component 1).

References

1. van der Aa, H., Balder, K.J., Maggi, F.M., Nolte, A.: Say it in your own words: Defining declarative process models using speech recognition. In: BPM (Forum). pp. 51–67 (2020)
2. van der Aa, H., Di Ciccio, C., Leopold, H., Reijers, H.A.: Extracting declarative process models from natural language. In: CAiSE. pp. 365–382 (2019)
3. Agostinelli, S., Chiariello, F., Maggi, F.M., Marrella, A., Patrizi, F.: Process mining meets model learning: Discovering deterministic finite state automata from event logs for business process analysis. Inf. Syst. **114** (2023)
4. Baier, C., Katoen, J.P.: Principles of model checking. MIT press (2008)
5. Boltenhagen, M., Chatain, T., Carmona, J.: A discounted cost function for fast alignments of business processes. In: BPM'21. pp. 252–269 (2021)
6. Carmona, J., van Dongen, B., Weidlich, M.: Conformance checking: foundations, milestones and challenges. In: Process mining handbook, pp. 155–190 (2022)
7. Casciani, A., Agostinelli, S., Lespérance, Y., Marrella, A., Sardiña, S.: Formal semantics for knowledge representation and automated reasoning in BPMN process models. Inf. Syst. **140** (2026)
8. Cecconi, A., De Giacomo, G., Di Ciccio, C., Maggi, F.M., Mendling, J.: Measuring the interestingness of temporal logic behavioral specifications in process mining. Inf. Syst. **107** (2022)
9. Chesani, F., Di Francescomarino, C., Ghidini, C., Loreti, D., Maggi, F.M., Mello, P., Montali, M., Skydanienko, V., Tessaris, S.: Towards the generation of the "perfect" log using abductive logic programming. In: CILC. pp. 179–192 (2019)
10. Chiariello, F., Maggi, F.M., Patrizi, F.: Asp-based declarative process mining. In: AAAI. pp. 5539–5547 (2022)
11. Corea, C., Delfmann, P.: Quasi-inconsistency in declarative process models. In: BPM Forum. pp. 20–35 (2019)
12. Corea, C., Kuhlmann, I., Thimm, M., Grant, J.: Paraconsistent reasoning for inconsistency measurement in declarative process specifications. Inf. Syst. **122** (2024)
13. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: IJCAI. pp. 854–860 (2013)
14. Di Ciccio, C., Bernardi, M.L., Cimitile, M., Maggi, F.M.: Generating event logs through the simulation of Declare models. In: EOMAS. pp. 20–36 (2015)
15. Di Ciccio, C., Maggi, F.M., Mendling, J.: Efficient discovery of target-branched declare constraints. Inf. Syst. **56**, 258–283 (2016)
16. Di Ciccio, C., Maggi, F.M., Montali, M., Mendling, J.: Resolving inconsistencies and redundancies in declarative process models. Inf. Syst. **64**, 425–446 (2017)
17. Di Ciccio, C., Montali, M.: Declarative process specifications: Reasoning, discovery, monitoring. In: Process Mining Handbook, pp. 108–152. Springer (2022)

Pre-print copy of the manuscript published by Springer (available at link.springer.com)
identified by DOI: 10.1007/978-3-032-37777-7_10

18. Donadello, I., Maggi, F.M., Patrizi, F., Tessaris, S., Zorzi, M.: Flexible event log generation using answer set programming. Inf. Syst. **139**, 102714 (2026)
19. Donadello, I., Maggi, F.M., Riva, F., Singh, M.: Asp-based log generation with purposes in declare4py. In: ICPM Doctoral Consortium / Demo (2023)
20. Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management, Second Edition. Springer (2018)
21. Figl, K., Di Ciccio, C., Reijers, H.A.: Do declarative process models help to reduce cognitive biases related to business rules? In: ER. pp. 119–133 (2020)
22. Fox, M., Long, D.: PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. JAIR **20** (2003)
23. Frederiks, P.J.M., van der Weide, T.P.: Information modeling: The process and the required competencies of its participants. Data Knowl. Eng. **58**(1), 4–20 (2006)
24. Geffner, H., Bonet, B.: A Concise Introduction to Models and Methods for Automated Planning. Synthesis Lectures on AI and Machine Learning **8**(1) (2013)
25. Gusfield, D.: Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology. Cambridge University Press (1997)
26. Haisjackl, C., Barba, I., Zugal, S., et al.: Understanding declare models: strategies, pitfalls, empirical results. Softw. Syst. Model. **15**(2), 325–352 (2016)
27. Harel, D., Marelly, R.: Come, Let's Play: Scenario-based Programming Using LSCs and the Play-Engine. Springer (2003)
28. Harel, D., Marelly, R.: Specifying and executing behavioral requirements: the play-in/play-out approach. Software and System Modeling **2**, 82–107 (2003)
29. Helmert, M.: New Complexity Results for Classical Planning Benchmarks. In: 16th Int. Conf. on Automated Planning and Scheduling (ICAPS'06). pp. 52–62 (2006)
30. Iacometta, C., Di Ciccio, C.: Declarative process mining with minerful, reloaded. In: BPM (Demos / Resources Forum). pp. 328–335 (2025)
31. Katz, M., Sohrabi, S.: Reshaping diverse planning. In: AAAI 2020 (2020)
32. Katz, M., Sohrabi, S., Udrea, O.: Top-quality planning: Finding practically useful sets of best plans. In: AAAI 2020. pp. 9900–9907 (2020)
33. Katz, M., Sohrabi, S., Udrea, O., Winterer, D.: A novel iterative approach to top-k planning. In: ICAPS 2018. pp. 132–140 (2018)
34. Maggi, F.M., Marrella, A., Patrizi, F., Skydanienko, V.: Data-aware declarative process mining with SAT. ACM Trans. Intell. Syst. Tech. **14**(4), 75:1–75:26 (2023)
35. Marrella, A.: Automated planning for business process management. J. Data Semant. **8**(2) (2019)
36. Park, G., Kofferath, J., Cho, M.: Extracting declarative constraints for process modeling from natural language descriptions with large language models. Decis. Support Syst. **203**, 114627 (2026)
37. Schunselaar, D.M.M., Maggi, F.M., Sidorova, N., van der Aalst, W.M.P.: Configurable declare: Designing customisable flexible process models. In: CoopIS (2012)
38. Slaats, T., Debois, S., Hildebrandt, T.T.: Open to change: A theory for iterative test-driven modelling. In: BPM. pp. 31–47 (2018)
39. Uchitel, S., Kramer, J., Magee, J.: Implied scenario detection in the presence of behaviour constraints. In: VISS@ETAPS. vol. 65, pp. 65–84 (2002)
40. Uchitel, S., Kramer, J., Magee, J.: Synthesis of behavioral models from scenarios. IEEE Trans. Software Eng. **29**(2), 99–115 (2003)
41. Vespa, M., Bellodi, E., Chesani, F., et al.: Probabilistic compliance in declarative process mining. In: PMAI@ECAI24. pp. 11–22 (2024)
42. Zugal, S., Pinggera, J., Weber, B.: Creating declarative process models using test driven modeling suite. In: CAiSE Forum (Selected papers). pp. 16–32 (2011)
43. Zugal, S., Pinggera, J., Weber, B.: The impact of testcases on the maintainability of declarative process models. In: BMMDS/EMMSAD. pp. 163–177 (2011)

Pre-print copy of the manuscript published by Springer (available at link.springer.com) identified by DOI: 10.1007/978-3-032-37777-7_10

This document is a pre-print copy of the manuscript
(Giovannetti et al. 2026)
published by Springer (available at link.springer.com).

The final version of the paper is identified by DOI: 10.1007/978-3-032-37777-7_10

References

Giovannetti, Alessandra, Giacomo Acitelli, Angelo Casciani, Claudio Di Ciccio, and Andrea Marrella (2026). "Customizable Declarative Process Specification Scenarios via Automated Planning". In: *BPM*. Ed. by Fabrizio Maria Maggi, Henrik Leopold, Flávia Maria Santoro, and Avigdor Gal. Vol. 16936. Lecture Notes in Computer Science. Springer, pp. 128–147. ISBN: 978-3-032-37776-0. DOI: 10.1007/978-3-032-37777-7_10.

BibTeX

```
@InProceedings{   Giovannetti.etal/BPM2026:DeclarativeProcessSpecificationScenariosviaPlanning,
  author        = {Giovannetti, Alessandra and Acitelli, Giacomo and
                   Casciani, Angelo and Di Ciccio, Claudio and Marrella,
                   Andrea},
  booktitle     = {BPM},
  title         = {Customizable Declarative Process Specification Scenarios
                   via Automated Planning},
  year          = {2026},
  pages         = {128--147},
  crossref      = {BPM2026},
  doi           = {10.1007/978-3-032-37777-7_10},
  keywords      = {Automated planning; Simulation; Process science; Declare}
}
@Proceedings{     BPM2026,
  title         = {Business Process Management - 24th International
                   Conference, {BPM} 2026, Toronto, ON, Canada, September
                   27-October 2, 2026, Proceedings},
  year          = {2027},
  editor        = {Fabrizio Maria Maggi and Henrik Leopold and Fl{\'{a}}via
                   Maria Santoro and Avigdor Gal},
  isbn          = {978-3-032-37776-0},
  publisher     = {Springer},
  series        = {Lecture Notes in Computer Science},
  volume        = {16936},
  doi           = {10.1007/978-3-032-37777-7}
}
```