# From Sound Workflow Nets
# to LTL$_f$ Declarative Specifications
# by Casting Three Spells

Luca Barbaro[1], Giovanni Varricchione[2], Marco Montali[3], Claudio Di Ciccio[2]

[1]*Sapienza University of Rome, Italy, luca.barbaro@uniroma1.it*
[2]*Utrecht University, Netherlands, g.varricchione@uu.nl, c.diciccio@uu.nl*
[3]*Free University of Bozen-Bolzano, Italy, montali@inf.unibz.it*

**Abstract**

In process management, effective behavior modeling is essential for understanding execution dynamics and identifying potential issues. Two complementary paradigms have emerged in the pursuit of this objective: the imperative approach, representing all allowed runs of a system in a graph-based model, and the declarative one, specifying the rules that a run must not violate in a constraint-based specification. Extensive studies have been conducted on the synergy and comparisons of the two paradigms. To date, though, whether a declarative specification could be systematically derived from an imperative model such that the original behavior was fully preserved (and if so, how) remained an unanswered question. In this paper, we propose a three-fold contribution. (1) We introduce a systematic approach to synthesize declarative process specifications from safe and sound Workflow nets. (2) We prove behavioral equivalence of the input net with the output specification, alongside related guarantees. (3) We experimentally demonstrate the scalability and compactness of our encoding through tests conducted with synthetic and real-world testbeds.

**Keywords:** Process modeling, Petri nets, Linear-time Temporal Logic on finite traces, DECLARE

## 1  Introduction

The act of modeling a process is a key element in a multitude of domains, including business process management [20], and is specifically tailored to meet the specific requirements and objectives of the individual application scenarios. Two fundamental, complementary paradigms cover the spectrum

1

of modeling: the imperative (e.g., Worflow nets [3] and their derived industrial standard BPMN [20]) and the declarative (e.g., DECLARE [26] and DCR Graphs [21]). Generally, the former class offers the opportunity to explicitly capture the set of actions available at each reachable state of the process, from start to end. However, such models often show limitations when it comes to capture flexibility in execution, since the possible runs highly vary and their graph-based structure gets cluttered. To compactly represent that variability, declarative specifications depict the rules that govern the behavior of every instance, leaving the allowed sequences implicit as long as none of those rules is violated.

Research has acknowledged that none of the available representations would be superior in all cases, as imperative and declarative approaches are apt to different comprehension tasks [27]. The ability to translate one representation to the other while preserving behavioral equivalence would allow the comparison and selection of the most suitable one. The first work in this direction is [29], where a systematic procedure is proposed to turn a declarative specification into an imperative model. Other endeavors followed to close the circle by providing an approximate solution to the inverse path (i.e., from an imperative model to a declarative specification), resorting on re-discovery over simulations [30], state space exploration [31], or behavioral comparison [7].

The goal of this work is to close the existing gap of this procedural-to-declarative direction. To this end, we show how to encode a safe and sound Workflow net [3] into a behaviorally equivalent DECLARE specification [18]. The three spells mentioned in the title of this paper refer to the fact that we only employ three parametric constraint types (*templates*) in the DE-CLARE repertoire. Importantly, the encoding is obtained in one pass and modularly over the net, preserving runs and choice points without incurring the state space explosion caused by concurrency unfolding. A byproduct of the encoding is that a safe and sound Workflow net induces a star-free regular language when considering transitions of the former as the alphabet of the latter. This strengthens the well-known fact that languages induced by sound Workflow nets are regular. Then, we evaluate the scalability of our approach by experimentally testing our proof-of-concept implementation against synthetic and real-world testbeds. Also, we show a downstream reasoning task on process diagnostics with public benchmarks.

The remainder of the paper is organized as follows. Section 2 provides an overview of the background knowledge our research is built upon. We describe our algorithm and formally discuss its correctness and complexity in Section 3. Section 4 evaluates our implementation to demonstrate the

feasibility of our approach. Finally, we conclude by discussing related works in Section 5 and outlining future research directions in Section 6.

## 2 Background

In this section, we formally describe the foundational pillars our work is built upon.

### 2.1 Linear Temporal Logic on finite traces

$\text{LTL}_f$ has the same syntax of LTL [28], but is interpreted on finite traces. Here, we consider the LTL dialect including past modalities [24] for declarative process specifications as in [9]. From now on, we fix a finite set $\Sigma$ representing an alphabet of propositional symbols. A (finite) *trace* $\pi = \langle a_1, \ldots, a_n \rangle \in \Sigma$ is a finite sequence of symbols of length $|\pi| = n$ (with $n \in \mathbb{N}$), where the occurrence of symbol $a_i$ at instant $i$ of the trace represents an *event* that witnesses $a_i$ at instant $i$ —we write $\pi(i) = a_i$. Notice that *at each instant we assume that one and only one symbol occurs*. Using standard notation from regular expressions, $\Sigma^*$ denotes the overall set of finite traces derived from events belonging to $\Sigma$.

**Definition 1 ($\text{LTL}_f$).** *(Syntax) Well-formed $\text{LTL}_f$ formulae are built from a finite non-empty alphabet of symbols $\Sigma \ni$ a, the unary temporal operators $\mathbf{X}$ ("next") and $\mathbf{Y}$ ("yesterday"), and the binary temporal operators $\mathbf{U}$ ("until") and $\mathbf{S}$ ("since") as follows:*

$$\varphi ::= a \mid (\neg\varphi) \mid (\varphi_1 \wedge \varphi_2) \mid (\mathbf{X} \; \varphi) \mid (\varphi_1 \; \mathbf{U} \; \varphi_2) \mid (\mathbf{Y} \; \varphi) \mid (\varphi_1 \; \mathbf{S} \; \varphi_2).$$

*(Semantics) An $\text{LTL}_f$ formula $\varphi$ is inductively satisfied in some instant $i$ (with $1 \leq i \leq n$) of a trace $\pi$ of length $n \in \mathbb{N}$, written $\pi, i \vDash \varphi$, if the following holds:*

*$\pi, i \vDash a$ iff $\pi(i)$ is assigned with a; $\pi, i \vDash \neg\varphi$ iff $\pi, i \nvDash \varphi$; $\pi, i \vDash \varphi_1 \wedge \varphi_2$ iff $\pi, i \vDash \varphi_1$ and $\pi, i \vDash \varphi_2$;*

*$\pi, i \vDash \mathbf{X} \; \varphi$ iff $i < n$ and $\pi, i+1 \vDash \varphi$; $\pi, i \vDash \mathbf{Y} \; \varphi$ iff $i > 1$ and $\pi, i-1 \vDash \varphi$;*

*$\pi, i \vDash \varphi_1 \; \mathbf{U} \; \varphi_2$ iff there exists $i \leq j \leq n$ such that $\pi, j \vDash \varphi_2$, and $\pi, k \vDash \varphi_1$ for all $k$ s.t. $i \leq k < j$;*

*$\pi, i \vDash \varphi_1 \; \mathbf{S} \; \varphi_2$ iff there exists $1 \leq j \leq i$ such that $\pi, j \vDash \varphi_2$, and $\pi, k \vDash \varphi_1$ for all $k$ s.t. $j < k \leq i$.*

*A formula $\varphi$ is satisfied by a trace $\pi$, written $\pi \vDash \varphi$, iff $\pi, 1 \vDash \varphi$.*  ◁
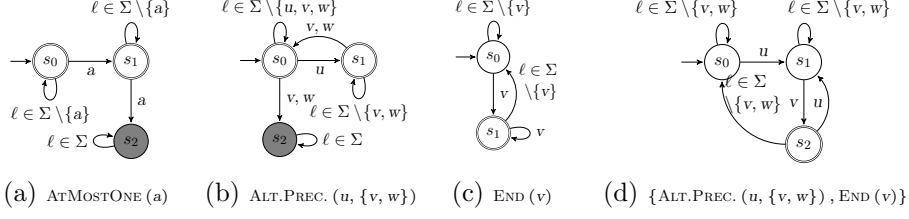
Figure 1: Example FSAs of DECLARE constraints. The FSA in Fig. 1(d) is trimmed.

From the basic operators above, the following can be derived: Classical boolean abbreviations **true**, **false**, $\lor$, $\rightarrow$; **F** $\varphi \equiv$ **true** **U** $\varphi$ indicating that $\varphi$ eventually holds true in the trace ("*eventually*"); **G** $\varphi \equiv \neg$ **F** $\neg\varphi$ indicating that $\varphi$ holds true from the current on ("*always*").

As an example, let $\pi = \langle a, b, c, e, f, g, u, v \rangle$ be a trace and $\varphi_1$, $\varphi_2$ and $\varphi_3$ three LTL$_f$ formulae defined as follows: $\varphi_1 \doteq f$; $\varphi_2 \doteq$ **Y** $c$; $\varphi_3 \doteq$ **G** $(f \rightarrow$ **Y** $e)$. We have that $\pi, 1 \nvDash \varphi_1$ whereas $\pi, 5 \vDash \varphi_1$; $\pi, 1 \nvDash \varphi_2$ while $\pi, 4 \vDash \varphi_2$; $\pi, 1 \vDash \varphi_3$ (hence, $\pi \vDash \varphi_3$); in fact, $\pi, i \vDash \varphi_3$ for any instant $1 \leq i \leq |\pi|$.

## 2.2 Finite State Automata

Every LTL$_f$ formula can be encoded into a *deterministic finite state automaton* [11].

**Definition 2 (Finite State Automaton).** *A (deterministic) finite state automaton (FSA) is a tuple $\mathcal{A} = (S, s_0, s_F, \Sigma, \delta)$, where $S$ is a finite set of states, $s_0 \in S$ is the initial state, $s_F \subseteq S$ is the set of accepting states, $\Sigma$ is the input alphabet of the automaton, and $\delta : S \times \Sigma \rightarrow S$ is the state transition function.* ◁

Figure 1 depicts three finite state automata (FSAs). An FSA reads in input sequences of symbols ("*string*") of its input alphabet. It starts in its initial state $s_0$ and updates the state after having read each symbol via the state transition function $\delta$. We say that a FSA *accepts* a string if after reading it is in one of its accepting states (i.e., a state in $s_F$), and otherwise we say that it *rejects* that string. The set of strings accepted by an FSA $\mathcal{A}$ is called the *language* of $\mathcal{A}$.

**Definition 3 (Bisimilarity).** *Two FSAs $\mathcal{A} = (S, s_0, s_F, \Sigma, \delta)$ and $\mathcal{A}' = (S', s_0', s_F', \Sigma, \delta')$ are* bisimilar *if and only if there exists a relation $\sim \subset S \times S'$ such that the following hold: $(s_0, s_0') \in \sim$; if $(s, s') \in \sim$, then*

4

Table 1: Semantics of some DECLARE constraint templates

| Template | LTL$_f$ expression [12, 9] | Description |
|---|---|---|
| ATMOSTONE $(x)$ | $\mathbf{G}\ (x \rightarrow \neg\ \mathbf{X}\ \mathbf{F}\ x)$ | $x$ occurs at most once in the trace |
| END $(x)$ | $\mathbf{G}\ \mathbf{F}\ x$ | The last event of any trace is $x$ |
| ALTERNATEPRECEDENCE $(y,x)$ | $\mathbf{G}\ (x \rightarrow \mathbf{Y}\ (\neg x\ \mathbf{S}\ y))$ | Every occurrence of $x$ requires that $y$ occurred before, with no recurrence of $x$ in between |

$(\delta(s, \ell), \delta'(s', \ell)) \in \sim$ *for any* $\ell \in \Sigma$; *if* $(s, s') \in \sim$, *then* $s \in s_F$ *if and only if* $s' \in s_F'$. ◁

**Observation 1.** *In the case of FSAs, bisimilarity coincides with* language equivalence*, i.e., two FSAs are bisimilar if and only if the sets of strings that they accept are equal [22].* ◁

A direct approach that builds a non-deterministic FSA $\mathcal{A}_\varphi$ accepting all and only the traces that satisfy a given LTL$_f$ formula $\varphi$ is presented in [11]. We make two further observations from [22]: *(i)* the so-obtained FSAs can be determinized, minimized, and trimmed using standard techniques without modifying the accepted language, and *(ii)* given any two FSAs $\mathcal{A}_\varphi$ and $\mathcal{A}_{\varphi'}$, their *product* $\mathcal{A}_\varphi \times \mathcal{A}_{\varphi'}$ recognizes all and only the traces of $\varphi \wedge \varphi'$.

## 2.3 LTL$_f$-based declarative specifications

The semantics of a DECLARE template is given as an LTL$_f$ formula. Given the free variables ("*parameters*") $x$ and $y$, e.g., ALTERNATEPRECEDENCE $(y, x)$ corresponds to $\mathbf{G}\ (x \rightarrow \mathbf{Y}\ (\neg x\ \mathbf{S}\ y))$, witnessing that for every instant in which $x$ is verified, then a previous instant must verify $y$ without any occurrences of $x$ in between. Hitherto, we will occasionally use an abbreviation for the template name — ALT.PREC. $(y, x)$. Table 1 shows the LTL$_f$ formulae of some templates of the DECLARE repertoire. Standard DECLARE imposes that template parameters be interpreted as single symbols of $\Sigma$ to build *constraints*. For example, ALT.PREC. $(\mathsf{b}, \mathsf{c})$ interprets $x$ as $c$ and $y$ as $b$. Branched DECLARE [26] comprises the same set of templates of standard DECLARE, yet allowing the interpretation of parameters as elements of a join-semilattice $(\Sigma, \vee)$, i.e., an idempotent commutative semigroup, where $\vee$ is the join-operation [15]. We shall use a clausal set-notation whenever a parameter is interpreted as a disjunction of literals. For example, ALT.PREC. $(\{a, w\}, b)$ interprets $x$ as $b$ and $y$ as $a \vee w$: for every $b$ occurring in a trace, a previous instant must have verified $a \vee w$, without $b$ recurring between that instant and the following occurrence of $b$. The conjunction of a finite set of constraints forms a DECLARE specification. In the following, we formalize the above notions.

**Definition 4.** *A* DECLARE *specification* $\mathcal{DS} = (\text{REP}, \Sigma, K)$ *is a tuple wherein:*

REP *is a finite non-empty set of* templates, *or "repertoire", where each template* $\text{K}(x_1, \ldots, x_m)$ *is an* $\text{LTL}_f$ *formula parameterized on free variables* $x_1, \ldots, x_m$;

$\Sigma \ni \mathbf{a}_i$ *is a finite non-empty alphabet of* symbols $a_i$ *with* $1 \le i \le |\Sigma|$, $|\Sigma| \in \mathbb{N}$;

$K$ *is a finite set of* constraints, *namely pairs* $(\text{K}(x_1, \ldots, x_m), \kappa)$ *where* $\text{K}(x_1, \ldots, x_m)$ *is a template from* REP, *and* $\kappa : \{x_1, \ldots, x_m\} \to 2^\Sigma \setminus \{\}$ *is a mapping from every variable* $x_i$ *to a non-empty, finite set of symbols* $A_i = \{a_{i,1}, \ldots, a_{i,v_i}\} \subseteq \Sigma$, *with* $1 \le i \le m$ *and* $1 \le v_i \le |\Sigma|$; *we denote such a constraint with* $\text{K}(A_1, \ldots, A_m)$ *or equivalently* $\text{K}(\{a_{1,1}, \ldots, a_{1,v_1}\}, \ldots, \{a_{m,1}, \ldots, a_{m,v_m}\})$, *omitting curly brackets from the latter form whenever a variable is mapped to a singleton.* ◁

As an example, consider the following specification: REP $=$ $\{\text{ATMOSTONE}(x), \text{ END}(x), \text{ ALT.PREC.}(y, x)\}$, $\Sigma = \{a, b, c, d, e, f, g, u, v, w\}$, and $K = \{\text{ATMOSTONE}(a), \text{ END}(v), \text{ALT.PREC.}(e, f), \text{ ALT.PREC.}(\{a, w\}, b), \text{ ALT.PREC.}(u, \{v, w\})\}$, where, e.g., $\text{ALT.PREC.}(\{a, w\}, b)\}$ is derived from the template $\text{ALT.PREC.}(y, x)$ by mapping $y \mapsto_\kappa \{a, w\}$ and $x \mapsto_\kappa b$.

**Definition 5 (Constraint formula, satisfying trace).** *Let* $\text{K}(A_1, \ldots, A_m)$ *be a constraint, whereby* $A_i = \{a_{i,1}, \ldots, a_{i,v_i}\}$ *for each* $1 \le i \le m$. *Its* constraint formula, *written* $\varphi_{\text{K}(A_1,\ldots,A_m)}$, *is the* $\text{LTL}_f$ *formula obtained from the template* $\text{K}(x_1, \ldots, x_m)$ *by interpreting* $x_i$ *as* $(a_{i,1} \vee \cdots \vee a_{i,v_i})$ *for each* $1 \le i \le m$. *A trace* $\pi$ *satisfies* $\text{K}(A_1, \ldots, A_m)$ *iff* $\pi \models \varphi_{\text{K}(A_1,\ldots,A_m)}$; *otherwise, we say that* $\pi$ *violates* $\text{K}(A_1, \ldots, A_m)$. ◁

Considering Tab. 1 and the above example specification, we have that $\varphi_{\text{ALT.PREC.}(\{a,w\},b)} = \mathbf{G}\ (b \to \mathbf{Y}\ (\neg b\ \mathbf{S}\ (a \vee w)))$, and $\varphi_{\text{END}(v)} = \mathbf{G}\ \mathbf{F}\ v$. Traces $\langle a, b, c\rangle$, $\langle a, b, c, f, u, w, b\rangle$, and $\langle a, b, c, e, f, g, u, v\rangle$ satisfy $\text{ALT.PREC.}(\{a, w\}, b)$, while only the third one satisfies $\text{END}(v)$.

**Definition 6 (Specification formula, model trace).** *A given* DECLARE *specification* $\mathcal{DS} = (\text{REP}, \Sigma, K)$ *is logically represented by conjoining its constraint formulae* $\varphi_{\mathcal{DS}} \doteq \bigwedge_{\text{K}(A_1,\ldots,A_m) \in K} (\varphi_{\text{K}(A_1,\ldots,A_m)})$. *A trace is a* model trace *for the specification,* $\pi \models \mathcal{DS}$, *iff* $\pi \models \varphi_{\mathcal{DS}}$, *i.e., it satisfies the conjunction of all the constraint formulae,* $\pi \models \varphi_{\text{K}(A_1,\ldots,A_m)}$ *for each* $\text{K}(A_1, \ldots, A_m) \in K$. ◁
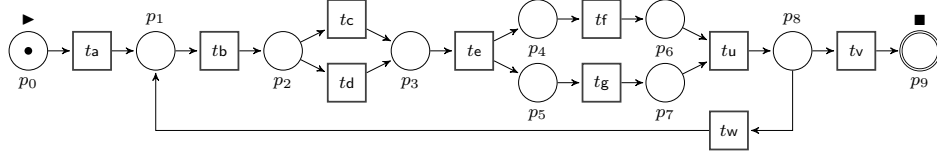
Figure 2: A Workflow net

The specification formula of the above example is

$$( \ \mathbf{G} \ (a \to \neg \ \mathbf{X} \ \ \mathbf{F} \ a)) \wedge$$
$$( \ \mathbf{G} \ \ \mathbf{F} \ v) \wedge$$
$$( \ \mathbf{G} \ (f \to \ \mathbf{Y} \ (\neg f \ \mathbf{S} \ e))) \wedge ( \ \mathbf{G} \ (b \to \ \mathbf{Y} \ (\neg b \ \mathbf{S} \ (a \vee w)))) \wedge$$
$$( \ \mathbf{G} \ ((v \vee w) \to \ \mathbf{Y} \ (\neg (v \vee w) \ \mathbf{S} \ u)))$$

$\langle a, b, c, e, f, g, u, v \rangle$ is a model trace for it, unlike $\langle a, b, c \rangle$ or $\langle a, b, c, f, u, w, b \rangle$.

Leveraging the techniques mentioned at the end of Sect. 2.2 and the above definition, we can create an FSA that accepts all and only the traces of a single DECLARE formula $\varphi$ and of a whole specification $\mathcal{DS}$.

**Definition 7 (Constraint and specification FSA).** *Let $\varphi_1, \ldots, \varphi_{|K|}$ be the constraint formulae of a process specification $\mathcal{DS}$. A* constraint automaton *$\mathcal{A}_{\varphi_i}$ is an FSA that accepts all and only those traces that satisfy $\varphi_i$ [16] with $1 \leq i \leq |K|$. The product automaton $\mathcal{A}_{\varphi_1} \times \cdots \times \mathcal{A}_{\varphi_{|K|}}$ is the* specification FSA*, recognizing all and only the traces satisfying $\mathcal{DS}$.* ◁

Figures 1(a) to 1(c) show the automata of constraints ATMOSTONE $(a)$, ALT.PREC. $(u, \{v, w\})$, and END $(v)$, respectively. Figure 1(d) depicts the FSA of a specification consisting of END $(v)$ and ALT.PREC. $(u, \{v, w\})$. Notice that the accepting state cannot be reached from $s_2$ in Figs. 1(a) and 1(b). Instead, the FSA in Fig. 1(d) has no such trap states due to trimming.

Aside from keeping the FSA's language unchanged, trimming caters for structural compatibility with the state space representation of Workflow nets, which we discuss next.

## 2.4 Workflow nets

A Workflow net (see, e.g., Fig. 2) is a renowned subclass of Petri nets suitable for the formal representation of imperative process models [3].

**Definition 8 (Petri net).** *A place/transition net [14] (henceforth,* Petri net*) is a bipartite graph $(P, T, F)$, where $P$ (the finite set of "*places*") and*

7

$T$ (the finite set of "transitions") constitute the nodes ($P \cap T = \emptyset$), and the flow relation $F \subseteq (P \times T \uplus T \times P)$ defines the edges. ◁

Given a place $p \in P$, we shall denote the sets $\{t \mid (t,p) \in F\}$ and $\{t \mid (p,t) \in F\}$ with $\succ p$ ("*preset*") and $p \succ$ ("*postset*"), respectively. For example, in Fig. 2, $\succ p_3 = \{t\mathsf{c}\}$ and $p_3 \succ = \{t\mathsf{e}, t\mathsf{f}\}$.

**Definition 9 (Workflow net).** *A Workflow net $\mathcal{WN} = (P, T, F)$ is a Petri net such that:*
1. *There is a unique place ("initial place", $\blacktriangleright \in P$) such that its preset is empty;*
2. *There is a unique place ("output place", $\blacksquare \in P$) such that its postset is empty;*
3. *Every place $p \in P$ and transition $t \in T$ is on a path of the underlying graph from $\blacktriangleright$ to $\blacksquare$.* ◁

We remark that we operate with full knowledge of the imperative model's structure, treated as a white box. Therefore, we directly focus on transitions rather than on their labels here.

In Petri and Workflow nets, places can be *marked* with tokens, intuitively representing resources that are processed by the transitions succeeding them in the net. In Fig. 2, a token is graphically depicted as a solid circle (see $p_0$ in the figure). The state of a net is defined by the distribution of tokens over places. This is formalized with the notion of *marking*, a function mapping each place to the number of tokens in it. The net's state changes with the consumption and production of tokens caused by the execution ("*firing*") of transitions.

**Definition 10 (Marking and firing).** *Let $\mathcal{WN} = (P, T, F)$ be a Workflow net. A* marking *is a function $M : P \to \mathbb{N} \cup \{0\}$. The* initial marking $M_0$ *of $\mathcal{WN}$ maps $\blacktriangleright$ to $1$ and any other $p \in P \setminus \{\blacktriangleright\}$ to $0$. A marking $M$ of $\mathcal{WN}$ is* final *if $M(\blacksquare) > 0$. A marking* enables *a transition $t \in T$ iff $M(p) > 0$ for all places $p$ such that $t \in p\succ$. An enabled transition can* fire, *i.e., turn a marking $M$ into $M'$ (in symbols, $M[t\rangle M'$), according to the following rule: For each place $p \in P$, $M'(p) = M(p) + 1$ if $t \in \succ p$; $M'(p) = M(p) - 1$ if $t \in \succ p$; otherwise, $M'(p) = M(p)$.* ◁

In Fig. 2, e.g., the initial marking enables $t\mathsf{a}$. Denoting markings with a multi-set notation, $\{p_0\} [t\mathsf{a}\rangle \{p_1\}$. Subsequently, $t\mathsf{b}$ gets enabled. After firing $t\mathsf{b}$, and $t\mathsf{c}$ get enabled. With Petri and Workflow nets, interleaving semantics are adopted, thus only one transition can fire per timestep, thus the firing of $t\mathsf{b}$ and $t\mathsf{c}$ are mutually exclusive in that state.
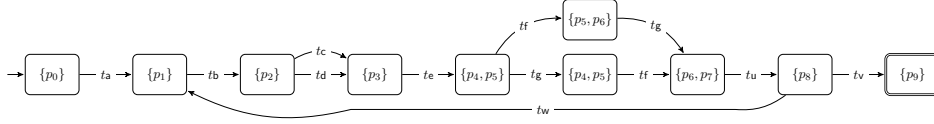
Figure 3: Reachability graph derived from the Workflow net in Fig. 2.

**Definition 11 (Firing sequence and run).** *Given a Workflow net $\mathcal{WN} = (P, T, F)$, a (finite) firing sequence $\sigma$ is $\langle \rangle$ or a sequence of transitions $\langle t_1, \ldots, t_n \rangle$ such that, for any index $1 \leq i \leq n$ with $n \in \mathbb{N}$, $t_i \in T$: (a) The i-th marking enables the i-th transition; (b) The $i+1$-th marking $M_{i+1}$ is such that $M_i[t_i\rangle M_{i+1}$. A marking $M'$ is* reachable *in $\mathcal{WN}$ if there exists a firing sequence $\sigma$ leading from the initial marking $M_0$ to $M'$ (in symbols, $M_0[\sigma\rangle M'$). A firing sequence leading from $M_0$ to a final marking is a* run.◁

Given a workflow net $\mathcal{WN}$, we will use $\mathcal{M}_{\mathcal{WN}}$ to denote the set of markings that can be reached from its initial marking $M_0$.

Runs of the Workflow net in Fig. 2 include $\langle t\mathsf{a}, t\mathsf{b}, t\mathsf{c}, t\mathsf{e}, t\mathsf{f}, t\mathsf{g}, t\mathsf{u}, t\mathsf{v} \rangle$ and $\langle t\mathsf{a}, t\mathsf{b}, t\mathsf{d}, t\mathsf{e}, t\mathsf{f}, t\mathsf{g}, t\mathsf{u}, t\mathsf{w}, t\mathsf{c}, t\mathsf{e}, t\mathsf{f}, t\mathsf{g}, t\mathsf{u}, t\mathsf{v} \rangle$. Any prefix of the first run of length 7 or less is a firing sequence from the initial marking $\{p_0\}$ but not a run.

In this paper, we assume that Workflow nets enjoy the following properties.

**Definition 12 (Soundness and safety of a Workflow net).** *Let $\mathcal{WN} = (P, T, F)$ be a Workflow net. $\mathcal{WN}$ is $k$-bounded if the number of tokens assigned by any reachable marking $M'$ to any place $p \in P$ is such that $M'(p) \leq k$. A 1-bounded Workflow net is* safe*. $\mathcal{WN}$ is* sound *iff it enjoys the following properties:* **Option to complete:** *from any marking $M$ it is possible to reach the final marking;* **Proper completion:** *if a reachable marking $M$ is such that $M(\blacksquare) > 0$, then $M$ is the final marking;* **No dead transitions:** *for any transition $t \in T$, there exists a reachable marking $M$ such that $t$ is enabled by $M$.*

Safe and sound Workflow nets (like the one depicted in Fig. 2) are a superclass of sound S-coverable nets, which in turn subsume safe and sound free-choice and well-structured nets [1]. These structural characteristics are widely recognized as recommendable in process management [3] and underpin well-formed business process diagrams [25]. Notice that, given a *safe* net, all markings that are reachable from the initial one are such that each place can be marked with at most one token. Also, the final marking of a sound workflow net is $\{\blacksquare\}$.
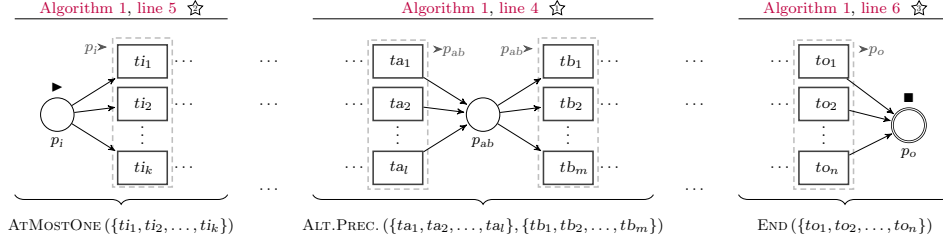
9

Figure 4: A graphical sketch of the execution of Alg. 1

The state space of $k$-bounded Petri nets can be represented in the form of a deterministic labeled transition system that go under the name of *reachability graph* [14]. Safe and sound Workflow nets have a given initial marking and one final marking. We can thus endow the state representation with these characteristics and reinterpret the known concept of reachability graph as a finite state automaton.

**Definition 13 (Reachability FSA).** *Given a sound and safe Workflow net* $\mathcal{WN}$*, the reachability FSA* $\mathcal{A}^{\mathcal{WN}} = \left(S^{\mathcal{WN}}, s_0^{\mathcal{WN}}, s_F^{\mathcal{WN}}, \Sigma^{\mathcal{WN}}, \delta^{\mathcal{WN}}\right)$ *is a finite state automaton where:*
$S^{\mathcal{WN}} = \mathcal{M}_{\mathcal{WN}}$ *, i.e., the set of states is the set of reachable markings in* $\mathcal{WN}$*;*
$s_0^{\mathcal{WN}} = \{\blacktriangleright\}$ *, i.e., the initial state is the initial marking of* $\mathcal{WN}$*;*
$s_{\mathbf{F}}^{\mathcal{WN}} = \{\{\blacksquare\}\}$ *, i.e., the accepting state set is a singleton containing the final marking of* $\mathcal{WN}$*;*
$\Sigma^{\mathcal{WN}} = T$ *, i.e., the alphabet is the set of transitions of* $\mathcal{WN}$*;*
$\delta^{\mathcal{WN}}$ *is s.t.* $\delta(M, t) = M'$ *iff* $M[t\rangle M'$ *for every transition* $t$ *and reachable* $M, M'$ *in* $\mathcal{WN}$*.* ◁

Figure 3 depicts the reachability FSA of the Workflow net in Fig. 2.

When dealing with accepting traces and languages, working with trimmed or non-trimmed FSAs is equivalent. This is not the case when we structurally relate the FSA of a DECLARE specification with the reachability FSA of a Workflow net. That FSA is indeed constructed, state-by-state, considering only enabled transitions, which globally yields that it is trimmed by design. Therefore we operate with trimmed FSAs for this comparison.

## 3   Synthesis of LTL$_f$ specifications from Workflow nets

In this section, we outline the algorithm (including the three spells to cast: ☆, ☆, and ③⟨) to synthesize a DECLARE specification $\mathcal{DS}$ from a given input

**Algorithm 1:** Wizard's guide to synthesize DECLARE specifications from Workflow nets

**Input:** $\mathcal{WN} = (P, T, F)$, a workflow net;
**Output:** $\mathcal{DS} = (\text{REP}, \text{ACT}, K)$, a declarative process specification

1  $\Sigma \leftarrow T;\quad K \leftarrow \{\};$   # Assign the alphabet with the transition set, and initialize the constraint set
2  $\mathcal{DS} \leftarrow (\{\text{ATMOSTONE}\,(x)\,, \text{END}\,(x)\,, \text{ALT.PREC.}\,(x, y)\}, \Sigma, K)$   # Initialize $\mathcal{DS}$ including templates
3  **foreach** $p \in P$ **do**   # Visit all places in $\mathcal{WN}$
4   **if** $\blacktriangleright p \neq \emptyset$ *and* $p \blacktriangleright \neq \emptyset$ **then** $K \leftarrow K \cup \{\text{ALT.PREC.}\,(\blacktriangleright p, p \blacktriangleright)\}$   # Add the ALT.PREC. $(\blacktriangleright p, p \blacktriangleright)$ constraint ☆
5   **else if** $\blacktriangleright p = \emptyset$ **then** $K \leftarrow K \cup \{\text{ATMOSTONE}\,(p \blacktriangleright)\}$   # Add the ATMOSTONE $(p \blacktriangleright)$ constraint ☆
6   **else if** $p \blacktriangleright = \emptyset$ **then** $K \leftarrow K \cup \{\text{END}\,(\blacktriangleright p)\}$   # Add the END $(\blacktriangleright p)$ constraint ☆

Table 2: DECLARE specification generated from the Workflow net in Fig. 2

| | | | | |
|---|---|---|---|---|
| ATMOSTONE $(t\mathsf{a})$ | END $(t\mathsf{v})$ | ALT.PREC. $(\{t\mathsf{a}, t\mathsf{w}\}, t\mathsf{b})$ | ALT.PREC. $(t\mathsf{b}, \{t\mathsf{d}, t\mathsf{c}\})$ | ALT.PREC. $(\{t\mathsf{d}, t\mathsf{c}\}, t\mathsf{e})$ |
| ALT.PREC. $(t\mathsf{e}, t\mathsf{f})$ | ALT.PREC. $(t\mathsf{e}, t\mathsf{g})$ | ALT.PREC. $(t\mathsf{f}, t\mathsf{u})$ | ALT.PREC. $(t\mathsf{g}, t\mathsf{u})$ | ALT.PREC. $(t\mathsf{u}, \{t\mathsf{v}, t\mathsf{w}\})$ |

safe and sound Workflow net $\mathcal{WN}$ ensuring behavioral equivalence between them. Algorithm 1 illustrates the transformation process. The algorithm initializes $\mathcal{DS}$ by assigning its alphabet with the transition set of $\mathcal{WN}$ (Alg. 1, ln. 1). Given the Workflow net in Fig. 2, e.g., $\Sigma$ gets $\{t\mathsf{a}, \ldots, t\mathsf{g}, t\mathsf{u}, t\mathsf{v}, t\mathsf{w}\}$. Then, it sets the three (necessary) templates that will be used (ln. 2): ATMOSTONE $(x)\,, \text{END}\,(x)\,,$ and ALT.PREC. $(y, x)$. A cycle begins to visit all places in $\mathcal{WN}$ and update $\mathcal{DS}$ by including a new constraint per place. Figure 4 graphically sketches this passage, which casts the three spells as follows: ☆ If $p$ is the output place as $p \blacktriangleright$ is empty, END $(\blacktriangleright p)$ is included (ln. 6); ☆ If $p$ is the input place as $\blacktriangleright p$ is empty, ATMOSTONE $(p \blacktriangleright)$ is added (ln. 5); ☆ Otherwise, ALT.PREC. $(\blacktriangleright p, p \blacktriangleright)$ becomes one of the constraints in $\mathcal{DS}$ (ln. 4). Intuitively, the rationale is that: ☆ Every time a transition in the postset of $p$ fires, it is necessary that at least one of the transitions in the preset of $p$ fired before and that no transition in the postset of $p$ has fired since then; ☆ Any of the transitions in the postset of ▶ will start the run and will not repeat afterwards (because no firing can assign ▶ with a token again by definition); ☆ Every run must terminate with one of the transitions in the preset of ■.

Table 2 shows the constraints that are generated by our algorithm if the Workflow net in Fig. 2 is fed in input. It is noteworthy to analyze in particular the non-trivial behavior entailed by constraints that stem from the parsing of places that begin or end cycles like $p_8$ and $p_1$ in Fig. 2. From the former we derive ALT.PREC. $(t\mathsf{u}, \{t\mathsf{v}, t\mathsf{w}\})$. It states that before $t\mathsf{v}$ or $t\mathsf{w}$,

$t$u must occur. Also, *neither* $t$v nor $t$w car recur until $t$u is repeated. As a consequence, an *exclusive* choice between $t$v and $t$w is enforced cyclically for each recurrence of $t$u. Dually, with ALT.PREC. ($\{t\mathsf{a}, t\mathsf{w}\}, t\mathsf{b}$) (generated by fetching the pre- and post-sets of $p_1$) we demand that *each* occurrence of $t$b follows $t$a or $t$w. From Tab. 2 we notice that $t$a can occur only once (ATMOSTONE ($t$a)), thus the subsequent recurrences of $t$b are bound to $t$w.

Given the construction in Alg. 1, it is clear that the semantics of the resulting DECLARE specification $\mathcal{DS}$ is an $\text{LTL}_f$ formula, traces of which are finite sequences of transitions of the input Workflow net $\mathcal{WN}$. Notice that the mapping of the transitions of $\mathcal{WN}$ to labels (as usual in a process modeling context) can be treated as a post-hoc refinement of $\mathcal{WN}$ and equivalently of $\mathcal{DS}$: Assuming that $t_1$ maps to label z, e.g., the occurrence of transition $t_1$ will emit z regardless of the underlying behavioral representation.

As established in the beginning of this section, our goal is to now show the behavioral equivalence between the DECLARE specification given in output by Alg. 1 and the input safe and sound Workflow net. To this end, we use the following notion of bisimilarity.

**Definition 14 (Bisimilarity of Workflow nets and Declare specifications).**
*A safe and sound Workflow net $\mathcal{WN}$ is bisimilar to a DECLARE specification $\mathcal{DS}$ if and only if the reachability FSA of $\mathcal{WN}$ (as per Def. 13) is bisimilar to the specification FSA of $\mathcal{DS}$ (as per Def. 7).* ◁

Given *(i)* this notion of bisimilarity, and *(ii)* Observation 1, it suffices to show that the two automata accept the same language to prove our claim. This, in turn, means that the DECLARE specification returned by Alg. 1 accepts all and only the runs of the input safe and sound Workflow net. We now proceed to formally express our claim.

**Theorem 1.** *Given a safe and sound Workflow net $\mathcal{WN}$, Alg. 1 returns a DECLARE specification $\mathcal{DS}$ such that: (i) any run of $\mathcal{WN}$ satisfies $\mathcal{DS}$, and (ii) any trace satisfying $\mathcal{DS}$ is a run of $\mathcal{WN}$.* ◁

*Proof.* We prove that $\mathcal{DS}$ and $\mathcal{WN}$ satisfy the two conditions stated in the claim.

*(i)* Let $\sigma$ be a run of $\mathcal{WN}$. We show that $\sigma \models \varphi_{\mathcal{DS}}$. As $\mathcal{WN}$ is a Workflow net, it has a unique input place and a unique output place. Let $p_i$ be ▶ and $p_o$ be ■. In $\varphi_{\mathcal{DS}}$, we have only one constraint for the templates END $(x)$ and ATMOSTONE $(x)$, namely END $(\blacktriangleright p_o)$ and ATMOSTONE $(p_i \blacktriangleright)$. Let $\{to_1, \dots to_n\}$ be the preset of $p_o$ (with $n \in \mathbb{N}$). Any run of $\mathcal{WN}$ must satisfy **G F** $(to_1 \vee \dots \vee to_n)$, i.e., $\varphi_{\text{END}(\blacktriangleright p_o)}$, as one of the transitions in

the preset of $p_o$ must fire last. Let $\{ti_1, \ldots ti_k\}$ be the postset of $p_i$ (with $k \in \mathbb{N}$). No other place $p' \neq p_i$ can be such that $ti \in p'\blacktriangleright$ for any $ti \in p_i\blacktriangleright$, otherwise $ti$ would be a dead transition (thus contradicting soundness): The initial marking assigns no token to $p'$, and no marking except the initial one assigns a token to $p_i$. As a consequence, any run of $\mathcal{WN}$ must satisfy $\mathbf{G}\ ((ti_1 \vee \ldots \vee ti_k) \rightarrow \neg\ \mathbf{X}\ \mathbf{F}\ (ti_1 \vee \ldots \vee ti_k))$, i.e., $\varphi_{\mathrm{ATMOSTONE}(p_i\blacktriangleright)}$.

It remains to show that $\sigma \models \varphi_{\mathrm{ALT.PREC.}(\blacktriangleright p, p\blacktriangleright)}$ for any arbitrary place $p \in P \setminus \{p_i, p_o\}$. Assume by contradiction that $\sigma \not\models \mathbf{G}\ (p\blacktriangleright \rightarrow \mathbf{Y}\ (\neg p\blacktriangleright\ \mathbf{S}\ \blacktriangleright p))$. Then, there must be a timestep $\ell < |\sigma|$ such that $\sigma, \ell \models p\blacktriangleright$, i.e., a transition in $p\blacktriangleright$ was fired, but $\sigma, \ell \not\models \mathbf{Y}\ (\neg p\blacktriangleright\ \mathbf{S}\ \blacktriangleright p)$. Notice that, as a transition in $p\blacktriangleright$ was fired, it means that a transition in $\blacktriangleright p$ was fired at a timestep $\ell' < \ell$, as otherwise there would be no token assigned to $p$ at timestep $\ell$. Then, for $\sigma, \ell \not\models \mathbf{Y}\ (\neg p\blacktriangleright\ \mathbf{S}\ \blacktriangleright p)$ to be true, it must be the case that a transition in $p\blacktriangleright$ was fired at some timestep $\ell''$ such that $\ell' < \ell'' < \ell$, and no transition in $\blacktriangleright p$ has been fired between timesteps $\ell''$ and $\ell$. However, this, in conjunction with the fact the Workflow net is safe, implies that it would not have been possible to fire a transition in $p\blacktriangleright$ at timestep $\ell$: $p$ has no token assigned at timestep $\ell$ as it was consumed to fire a transition in $p\blacktriangleright$ at timestep $\ell''$. Therefore, $\sigma \models \varphi_{\mathrm{ALT.PREC.}(\blacktriangleright p, p\blacktriangleright)}$ for any arbitrary place $p \in P \setminus \{p_i, p_o\}$, thus implying that $\sigma \models \varphi_{\mathcal{DS}}$.

  *(ii)* We now show that if a trace $\sigma$ is such that $\sigma \models \varphi_{\mathcal{DS}}$ then $\sigma$ is a run of $\mathcal{WN}$. Let $p_i$ be $\blacktriangleright$ and $p_o$ be $\blacksquare$ again. Since $\sigma \models \varphi_{\mathcal{DS}}$, we have that the trace correctly ends with a transition in the preset of $p_o$, because $\sigma \models \varphi_{\mathrm{END}(\blacktriangleright p_o)}$. Also, in a run of $\mathcal{WN}$, the transitions in $p_i\blacktriangleright$ can only be fired once, otherwise $\blacktriangleright p_i$ would be non-empty against the definition of $\blacktriangleright$. This holds true in $\sigma$, as $\sigma \models \varphi_{\mathrm{ATMOSTONE}(p_i\blacktriangleright)}$. Notice that, unlike all other transitions in $\mathcal{WN}$, only those in $p_i\blacktriangleright$ do *not* map to $x$ for $\mathrm{ALT.PREC.}(y, x)$ in $\mathcal{DS}$ by design of Alg. 1. Therefore, every trace will begin with the occurrence of one of the transitions in $p_i\blacktriangleright$ as it happens with the runs of $\mathcal{WN}$.

It remains to show that every transition in the trace $\sigma$ was fired in $\mathcal{WN}$ following the preceding sequence of transitions in the trace. Suppose by contradiction that this is not the case, i.e., that there is some transition $t$ fired at a timestep $\ell$ which could not have been fired in $\mathcal{WN}$ given the prefix of $\sigma$ from 1 to $\ell - 1$. Then, this implies that at least one of the places $p$ such that $t \in p\blacktriangleright$ does not have a token at timestep $\ell - 1$, i.e., $M_{\ell-1}(p) = 0$. Two conditions can entail this situation: Either no transition in $\blacktriangleright p$ was fired before, or a transition in $p\blacktriangleright$ was fired since the last timestep in which a transition in $\blacktriangleright p$ was fired, consuming the only token assigned to $p$. Both cases contradict the fact that $\sigma \models \varphi_{\mathrm{ALT.PREC.}(\blacktriangleright p, p\blacktriangleright)}$, thus proving that $\sigma$ is a valid sequence of transitions with respect to $\mathcal{WN}$. Thus, $\sigma$ is a run of $\mathcal{WN}$. $\dashv$

Given Observation 1, we immediately obtain the following corollary.

**Corollary 1.** *The* DECLARE *specification* $\mathcal{DS}$ *given in output by Alg. 1 is bisimilar to the input safe and sound Workflow net* $\mathcal{WN}$. ◁

This result has a profound implication that transcends DECLARE and $\text{LTL}_f$ but pertains to the languages recognized by safe and sound Workflow nets.

**Theorem 2.** *Languages of safe and sound Workflow nets are star-free regular expressions.* ◁

*Proof.* The claim follows from Theorem 1, recalling that DECLARE patterns are expressed in $\text{LTL}_f$, which is expressively equivalent to star-free regular expressions [12]. ⊣

**Space and time complexity.** Algorithm 1 outputs a DECLARE specification $\mathcal{DS} = (\text{REP}, \Sigma, K)$ which contains, for each place in the input Workflow net $\mathcal{WN} = (P, T, F)$, a constraint with the pre- and post-sets as its actual parameters. Each transition that is in relation with a place $p$ in the flow relation $F$ appears exactly once in the constraint stemming from $p$; therefore, the space complexity class of Alg. 1 is $\mathcal{O}(|F|)$. As for the time complexity, we can assume that a pre-processing step is conducted to represent $F$ in the form of a sequence of pairs, associating every place to its pre-set and post-set. The cost of this operation is $\Theta(F)$ and $\mathcal{O}(|P| \times |T|)$. For each place, the algorithm performs up to three if-checks and then a new constraint is created in constant time, hence $\mathcal{O}(|P|)$. Therefore, the time complexity of the algorithm is bounded by the update of $K$, necessitating up to $\mathcal{O}(|P| \times |T|)$ time.

Next, we experimentally validate and put the above theoretical results to the test.

## 4  Implementation and evaluation

We implemented Alg. 1 in the form of a proof-of-concept prototype encoded in Python. The tool, testbeds, and experimental results are available for public access.[1] In the following, we report on tests conducted with our algorithm's implementation to empirically confirm its soundness, assess memory efficiency, and gauge runtime performance. Finally, we showcase a process diagnostic application as a downstream task for our approach.
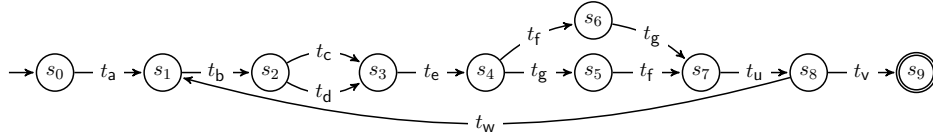
Figure 5: FSA of the specification in Tab. 2

## 4.1 Automata bisimulation

To experimentally validate the correctness of the implementation of Alg. 1, we performed a preliminary comparison of the reachability FSA (Def. 13) of known Workflow nets and the specification FSA (Def. 7) consisting of the DECLARE constraints returned by our tool. Figure 5 illustrates the FSA of the specification derived from the Workflow net in Fig. 2, computed with a dedicated module presented in [16]. Also by visual inspection, we can conclude that the two FSAs are bisimilar, as expected. Owing to space constraints, we cannot portray the entire range of automata derived from the Workflow nets in our experiments. The interested reader can find the full collection collection (including non-free choice nets such as that of [3, Fig. 24]) in our public codebase.[1]

## 4.2 Performance analysis

Here, we report on the quantitative assessment of our solution in terms of scalability given an increasing workload, and against real-world testbeds. For the former, we observe the time and space performance of our implemented prototype fed in input with Workflow nets of increasing size. We control the expansion process in two directions, so as to obtain the following separate effects: *(i)* more constraints are generated, while each is exerted on up to three literals; *(ii)* the amount of generated constraints remains fixed, while the literals mapped to their parameters increase. For the real-world testbed, we take as input processes discovered by a well-known imperative process mining algorithm from a collection of openly available event logs. We conducted the performance tests on an AMD Ryzen 9 8945HS CPU at 4.00 GHz with 32 GB RAM running Ubuntu 24.04.1. For the sake of reliability, we ran three iterations for every test configuration and averaged the outputs to derive the final result.

**Increasing constraint-set cardinality.** To examine the effectiveness of the Alg. 1 in handling an incremental number of constraints, we examine memory utilization and execution time through the progressive rise in the complexity
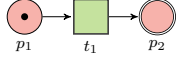
---

[1]https://github.com/l2brb/Sp3llsWizard

15

Figure 6: Base net used to initiate the expansion mechanism in Fig. 8
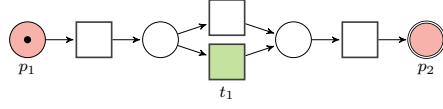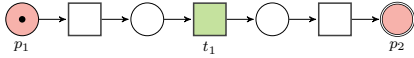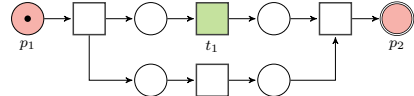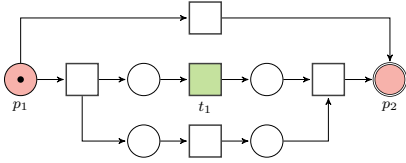


Figure 7: Conditional expansion of the net in Fig. 8(a)
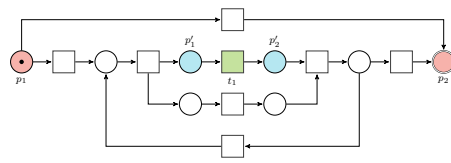


(a) Sequential expansion of the net in Fig. 6



(b) Parallel expansion of the net in Fig. 8(a)



(c) Conditional expansion of the net in Fig. 8(b)



(d) Loop expansion of the net in Fig. 8(c)

Figure 8: Transformation rules used to iteratively expand a safe and sound Workflow net.

of the input Workflow net. Our evaluation method relies on an expansion mechanism that iteratively applies a structured pattern of four soundness-preserving transformation rules from [2] to progressively increase the number of nodes and their configuration. This leads to a gradual increase in the number of constraints our algorithm needs to initiate. Starting from the Workflow net in Fig. 6, we designate transition $t_1$ as a fixed 'pivot', retaining the initial and final places ($p_1$ and $p_2$), and iteratively apply the expansion mechanism illustrated in Fig. 8. We apply known workflow patterns in the following order: (Fig. 8(a)) We add a transition before and after $t_1$; (Fig. 8(b)) We introduce a parallel execution path; (Fig. 8(c)) We insert an exclusive branch; (Fig. 8(d)) Finally, we incorporate a loop structure. Upon completion of the expansion process, we execute the algorithm, record the results, and initiate a new iteration, maintaining $t_1$ unchanged while reassigning $p_1$ and $p_2$ with the places that have $t_1$ in the preset and postset (see the places colored in blue and labeled with $p'_1$ and $p'_2$ in Fig. 8(d)). We reiterated the procedure 1000 times.

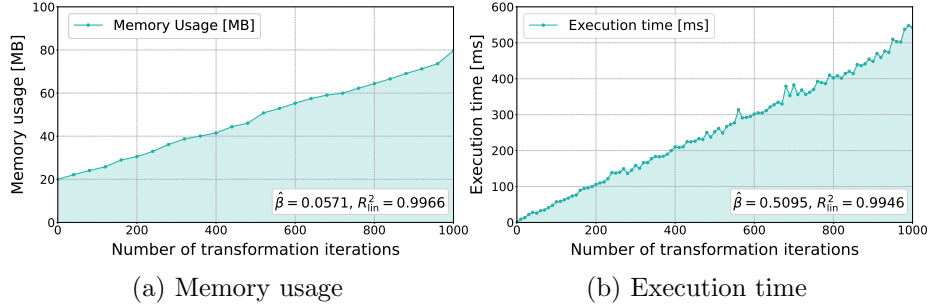Figure 9 displays the registered memory usage and execution time. To

(a) Memory usage          (b) Execution time

Figure 9: Test results for the incremental number of constraints setup
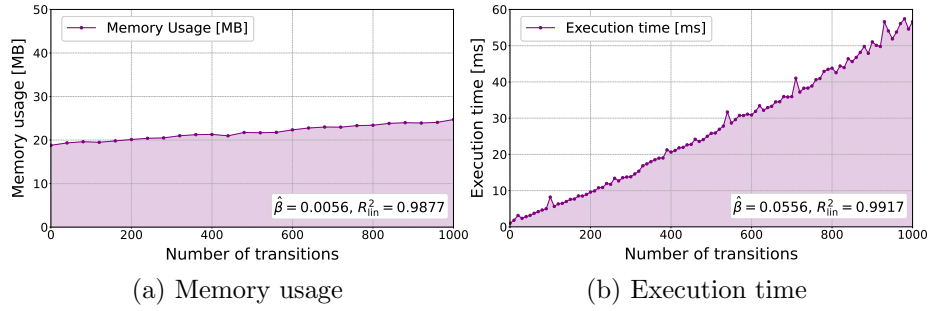


(a) Memory usage          (b) Execution time

Figure 10: Test results for the incremental constraints dimension setup

interpret the performance trends, we employ two well-established measures: the coefficient of determination $R^2_{\text{lin}}$, which assesses the goodness-of-fit of the data to a linear trend, and the $\hat{\beta}$ rate, which serves as a meter for the line's slope. As depicted in Fig. 9(a), memory consumption increases linearly with the number of iterations of the expansion mechanism confirmed by $R^2_{\text{lin}} = 0.9966$ and a low slope increase ($\hat{\beta} = 0.0571$). Figure 9(b) displays the execution time plot, with $R^2_{\text{lin}} = 0.9946$ and $\hat{\beta} = 0.5095$, thus indicating a linear trend with a slope exhibiting a moderate incline. We remark that these results are in line with the theoretical analysis of the space and time complexity in Sect. 3.

**Increasing constraint formula size.** Here, we configure the test on memory usage and execution time to investigate the algorithm's performance while handling an expanding constraints' formula size (i.e., with an increasing number of disjuncts). To this end, we progressively broaden the Workflow net by applying the soundness-preserving conditional expansion rule from [2] depicted in Fig. 7 to transition $t_1$ in the net of Fig. 8(a). We reiterate the process 1000 times. Figure 10 displays the results we registered. Observing

17

Table 3: Performance comparison with real-world process models

| Event log | Trans. | Places | Nodes | Mem.usage [MB] | Exec.time [ms] |
|---|---|---|---|---|---|
| BPIC 12 | 78 | 54 | 174 | 19.97 | 5.11 |
| BPIC $13_{cp}$ | 19 | 54 | 44 | 19.76 | 1.70 |
| BPIC $13_{inc}$ | 23 | 17 | 50 | 19.89 | 2.03 |
| BPIC $14_f$ | 46 | 35 | 102 | 19.90 | 3.31 |
| BPIC $15_{1f}$ | 135 | 89 | 286 | 20.44 | 8.39 |
| BPIC $15_{2f}$ | 200 | 123 | 422 | 20.91 | 12.30 |
| BPIC $15_{3f}$ | 178 | 122 | 396 | 20.77 | 11.49 |
| BPIC $15_{4f}$ | 168 | 115 | 368 | 20.55 | 11.38 |
| BPIC $15_{5f}$ | 150 | 99 | 320 | 20.43 | 9.16 |
| BPIC 17 | 87 | 55 | 184 | 19.91 | 5.67 |
| RTFMP | 34 | 29 | 82 | 19.81 | 3.47 |
| Sepsis | 50 | 39 | 116 | 19.75 | 3.65 |

Fig. 10(a), we can assert that the memory utilization increases linearly ($R^2_{lin} = 0.9877$) with a minimal rate ($\hat{\beta} = 0.0056$). The execution time plotted in Fig. 10(b) also exhibits a linear increase ($R^2_{lin} = 0.9917$), with a moderate slope inclination ($\hat{\beta} = 0.0556$). Once more, the results align with the theoretical complexity analysis in Sect. 3.

**Real-world process model testing.** To evaluate the performance of our algorithm in application on real process models, we conduct the same memory usage and execution time tests employing Workflow nets directly derived from a collection of real-life event logs available at *4TU.ResearchData.*[2] To this end, we employ the Inductive Miner algorithm version proposed in in [23], which filters out infrequent behavior while still discovering well-structured, sound models [5]. Thus, we first run the Inductive Miner on the event logs considered in [5] to generate the Workflow nets. We then apply Alg. 1 to derive the corresponding DECLARE specification. We report the aggregate test result in Tab. 3, detailing the memory usage, the execution time, and all the features of the mined Workflow nets. We find that the overall differences in resource usage are negligible. These real-world test outcomes again follow the complexity assumptions outlined in Sect. 3.

---

[2]The event logs used in our experiments are publicly available at https://data.4tu.nl/
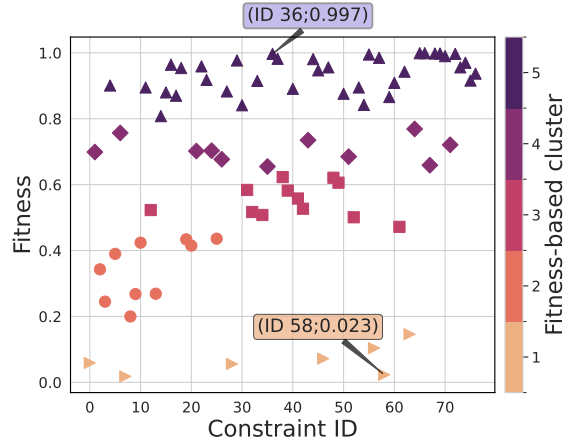
Figure 11: Fitness-based clusters of the constraints in the descriptive model of BPIC 15$_{5f}$

## 4.3 A downstream task: Using constraints as determinants for process diagnosis

Algorithm 1 enables the transition from an overarching imperative model to a constraint-based specification, enclosing parts of behavior into separate constraints. Herewith, we aim to demonstrate how we can single out the violated rules constituting the process model behavior, thereby spotlighting points of non-compliance with processes. In other words, we aim to use constraints as determinants for a process diagnosis. For this purpose, we created a dedicated module extending a declarative specification miner for constraint checking via the replay of runs on semi-symbolic automata like those in Figs. 1(a) to 1(c), following [17]. Without loss of generality, we build the runs from data pertaining to building permit applications in Dutch municipalities from BPIC 15$_{5f}$ [19] and apply the preprocessing technique mentioned in [5], resulting in 975 traces. We then process the log with the $\alpha$-algorithm [4] and provide the returned net as input to our implementation of Alg. 1.

We observe that the specification consists of 129 constraints. Of those, our tool detected violations by at least a trace for 77 of those. Figure 11 illustrates the percentage of satisfying traces (henceforth, *fitness* for brevity) of the 77 constraints, which we clustered into five distinct groups to ease inspection. We first focus on violated constraints exhibiting high fitness (the blue upward triangles at the top of Fig. 11). Let us take, e.g., the constraint identified by ID

19

36 in the figure: ALT.PREC. ($\{t01\_HOOFD\_490\_1, t13\_CRD\_010\}, t01\_HOOFD\_490\_1a$), which exhibits a fitness of 0.997. This constraint imposes that when "*Set Decision Status*" ($t01\_HOOFD\_490\_1a$) occurs, it should be preceded by either "*Create Environmental Permit Decision*" ($t01\_HOOFD\_490\_1$) or "*Coordination of Application*" ($t13\_CRD\_010$). In the three traces violating the constraint (11369696, 9613229, 12135936), though, "*Set Decision Status*" is preceded by neither of the two. By further inspection, we observe "*No Permit Needed or Only Notification Needed*" ($t14\_VRIJ\_010$) in the trace prefix instead, suggesting that the process bypasses the standard decision-making steps defined by the reference model in favor of an alternative where a permit decision is unnecessary. On the other side of the spectrum, let us look at constraints with low trace fitness values (depicted by rightward orange triangles in Fig. 11). These constraints likely suffer from systematic defects rather than spurious alterations in the process behavior. Constraint ID 58, e.g., belongs to this group: ALT.PREC. ($t1\_HOOFD\_510\_2, \{t01\_HOOFD\_510\_3, t01\_HOOFD\_520, tEND\}$) (depicted in the lower section of Fig. 11). Other constraints in the same group have in common the presence of $tEND$ in the activator's set. An explanation is that the BPIC 15 log allows a multitude of possible conclusions. The $\alpha$-algorithm, though, disregards the occurrence frequency of individual transitions during model construction, resulting in a non-selective inclusion of all events. Consequently, this affects the fitness of those constraints. Our tool specifically pinpoints and isolates the effect of this tendency from the remainder of the net. Thoroughly assessing the suitability of our approach for process diagnostics transcends the scope of this paper but paves the path for future work.

# 5 Related work

The relationship between imperative and declarative modeling approaches has been extensively explored in the existing literature, with a prevailing focus directed toward the development of analytics tools that effectively compare and integrate the strengths of both paradigms. Building on previous contributions aimed at establishing a formal connection between these paradigms [29, 10], our research focuses on providing a systematic approach for translating safe and sound Workflow nets into their declarative counterparts. A growing research stream configures this transformation aiming to leverage the support provided by the declarative specifications for conformance checking and anomaly detection. Notably, integrating declarative constraints into event log analysis facilitates more comprehensive diagnostics than those provided

by trace replaying techniques. In this regard, Rocha et al. [31] propose an automated method for generating conformance diagnostics using declarative constraints derived from an input imperative model. Their method relies on a library of templates internally maintained in the tool. Eligible constraints are generated by verifying the instantiation of those templates against the model's state space. The ones that are behaviorally compatible are then subject to redundancy removal pruning. Finally, the retained constraints are checked for conformance against log traces. In [7], the authors present a tool that derives a set of eligible constraints directly extracting relations based on a selection of BPMN models' activity patterns. The work of Rebmann et al. [30] proposes a framework for extracting best-practice declarative constraints from a collection of imperative models aiming to discover potential violations and undesired behavior. Constraints are extracted akin to [7], then refined and validated via natural-language-processing techniques to measure their relevance for a given event log. Busch et al. [8] adopt a similar technique to check constraints characterizing process model repositories against event logs. All these techniques share our aim to derive declarative constraints from imperative models given as input. However, they do so via simulation or state space exploration, with limited guarantees of behavioral equivalence. In contrast, our work proposes an algorithm that is proven to establish a formal equivalence between the given imperative model and the derived declarative specification. Being based on the sole exploration of the net's structure, it is also lightweight in terms of computational demands.

# 6 Conclusion and future work

In this paper, we presented a systematic approach to translate safe and sound Workflow nets into bisimilar DECLARE specifications. The latter are based solely on three $LTL_f$ formula templates from the DECLARE repertoire with branching: ATMOSTONE, END, and ALTERNATEPRECEDENCE. We provide a proof-of-concept implementation, of which we evaluate scalability and showcase applications against synthetic and real-world testbeds.

We believe that the scope of this research may be expanded in a number of directions. A natural extension of our work is the inclusion of label-mappings of the Workflow net in the declarative specifications, which would turn the constraints' semi-symbolic automata into transducers that are advantageous in conformance checking contexts.Moreover, we seek to broaden the application of our solution to detect behavioral violations, extending support to a wider range of imperative input models. Also, we aim to

investigate the correspondence between specification inconsistencies and Workflow net unsafeness and/or unsoundness. Another promising application lies in hybrid representations combining imperative and declarative paradigms. In this regard, our approach could facilitate behavioral comparisons akin to [6] and enable the construction of hybrid representations tailored to diverse scenarios [13].

# References

[1] van der Aalst, W.M.P.: Structural characterizations of sound workflow nets. Tech. Rep. 9623, Technische Universiteit Eindhoven (1996)

[2] van der Aalst, W.M.P.: Verification of workflow nets. In: ICATPN. pp. 407–426 (1997)

[3] van der Aalst, W.M.P.: The application of petri nets to workflow management. J. Circuits Syst. Comput. **8**(1), 21–66 (1998)

[4] van der Aalst, W.M.P., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE Trans. Knowl. Data Eng. **16**(9), 1128–1142 (2004)

[5] Augusto, A., Conforti, R., Dumas, M., et al.: Automated discovery of process models from event logs: Review and benchmark. IEEE Trans. Knowl. Data Eng. **31**(4), 686–705 (2019)

[6] Baumann, M.: Comparing imperative and declarative process models with flow dependencies. In: IEEESOSE 2018. pp. 63–68. IEEE Computer Society (2018)

[7] Bergmann, A., Rebmann, A., Kampik, T.: BPMN2Constraints: Breaking down BPMN diagrams into declarative process query constraints. In: BPM Demos. vol. 3469, pp. 137–141 (2023)

[8] Busch, K., Kampik, T., Leopold, H.: xSemAD: Explainable semantic anomaly detection in event logs using sequence-to-sequence models. In: BPM. vol. 14940, pp. 309–327 (2024)

[9] Cecconi, A., Di Ciccio, C., De Giacomo, G., Mendling, J.: Interestingness of traces in declarative process mining: The Janus LTLpf approach. In: BPM. pp. 121–138 (Sep 2018)

[10] Cosma, V.P., Hildebrandt, T.T., Slaats, T.: Transforming dynamic condition response graphs to safe petri nets. In: PETRI NETS 2023. Lecture Notes in Computer Science, vol. 13929, pp. 417–439. Springer (2023)

[11] De Giacomo, G., De Masellis, R., Montali, M.: Reasoning on LTL on finite traces: Insensitivity to infiniteness. In: AAAI. pp. 1027–1033 (2014)

[12] De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: IJCAI. pp. 854–860 (2013)

[13] De Smedt, J., De Weerdt, J., Vanthienen, J., et al.: Mixed-paradigm process modeling with intertwined state spaces. Bus. Inf. Syst. Eng. **58**(1), 19–29 (2016)

[14] Desel, J., Reisig, W.: Place/transition Petri Nets, pp. 122–173 (1998)

[15] Di Ciccio, C., Maggi, F.M., Mendling, J.: Efficient discovery of Target-Branched Declare constraints. Information Systems **56**, 258–283 (Mar 2016)

[16] Di Ciccio, C., Maggi, F.M., Montali, M., et al.: Resolving inconsistencies and redundancies in declarative process models. Information Systems **64**, 425–446 (Mar 2017)

[17] Di Ciccio, C., Maggi, F.M., Montali, M., et al.: On the relevance of a business constraint to an event log. Information Systems **78**, 144–161 (Nov 2018)

[18] Di Ciccio, C., Montali, M.: Declarative Process Specifications: Reasoning, Discovery, Monitoring, pp. 108–152. Springer (Jun 2022), open access

[19] van Dongen, B.: Bpi challenge 2015 municipality 5 (2015)

[20] Dumas, M., Rosa, M.L., Mendling, J., et al.: Fundamentals of Business Process Management, Second Edition (2018)

[21] Hildebrandt, T.T., Mukkamala, R.R.: Declarative event-based workflow as distributed Dynamic Condition Response graphs. In: PLACES 2010. vol. 69, pp. 59–73 (2010)

[22] Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)

[23] Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs containing infrequent behaviour. In: BPM Workshops 2013. vol. 171, pp. 66–78 (2013)

[24] Lichtenstein, O., Pnueli, A., Zuck, L.D.: The glory of the past. In: Logics of Programs. pp. 196–218 (1985)

[25] Ouyang, C., Dumas, M., van der Aalst, W.M.P., et al.: From business process models to process-oriented software systems. ACM Trans. Softw. Eng. Methodol. **19**(1), 2:1–2:37 (2009)

[26] Pesic, M.: Constraint-based Workflow Management Systems: Shifting Control to Users. Ph.D. thesis, TU Eindhoven (10 2008)

[27] Pichler, P., Weber, B., Zugal, S., et al.: Imperative versus declarative process modeling languages: An empirical investigation. In: BPM Workshops 2011. vol. 99, pp. 383–394 (2011)

[28] Pnueli, A.: The temporal logic of programs. In: FOCS. pp. 46–57 (1977)

[29] Prescher, J., Di Ciccio, C., Mendling, J.: From declarative processes to imperative models. In: SIMPDA 2014. vol. 1293, pp. 162–173 (2014)

[30] Rebmann, A., Kampik, T., Corea, C., van der Aa, H.: Mining constraints from reference process models for detecting best-practice violations in event log. CoRR **abs/2407.02336** (2024)

[31] Rocha, E.G., van Zelst, S.J., van der Aalst, W.M.P.: Mining behavioral patterns for conformance diagnostics. In: BPM 2024. vol. 14940, pp. 291–308 (2024)