Collaborations Beyond Simple Message Flows
A Technical Report on Their Semantics And Execution

Anti Alman[1], Jonas Lindner[2], Giovanni Meroni[2], Andrey Rivkin[2], Mathias Spezia[2], and Mihály Tass[2]

[1] Free University of Bozen-Bolzano, Italy anti.alman@unibz.it
[2] Technical University of Denmark, Denmark
{jmali,giom,ariv,matsp,mihta}@dtu.dk

1 Background and notation

1.1 Workflow Nets

Definition 1. *A WF-net [1] is a tuple $N = (P, T, F, in, out)$. Here,*

- *P and T are disjoint sets of places and transitions;*
- *$T = T_\tau \cup T_v$ is the disjoint union of silent transitions and visible transitions.*
- *$F \subseteq (P \times T) \cup (T \times P)$ is the flow relation;*
- *$in \in P$ (resp., $out \in P$) is the unique source (resp., sink) place s.t. $(t, i) \notin F$ (resp., $(o, t) \notin F$), for all $t \in T$;*
- *in the Petri net graph, every node in $P \cup T$ lies on some path from in to out.*

A marking is a function $m : P \to \mathbb{N}$. The initial and final markings m_0, m_f are the ones with a single token in *in* and *out*, respectively. Assuming the standard enabledness condition and firing rule [1], we write $m \xrightarrow{t \in T} m'$ for the transition firing relation. As customary, transition firing can be extended to sequences of transitions $\rho = t_1 \cdots t_k \in T^*$, to which we refer as *runs*. If m' can be reached from m by ρ, then we say m' is *reachable* from m, denoted as $m \xrightarrow{\rho} m'$. We denote with $\mathcal{R}(N)$ all markings of N that are reachable from m_0: $\mathcal{R}(N) = \{\, m \mid m_0 \xrightarrow{\sigma} m \text{ for some } \sigma \in T^* \,\}$.

A WF-net is 1-safe if every reachable marking assigns at most one token per place. In this paper, we restrict our focus to such nets.

Proposition 1 (Finiteness of markings). *Let N be a 1-safe WF-net. The set of reachable markings is finite. In particular, $|\mathcal{R}(N)| \leq 2^{|P|}$.*

The *language* of N is $\mathcal{L}(N) := \{\, \rho \mid m_0 \xrightarrow{\rho} m, m(out) = 1 \,\}$.

1.2 $\mathsf{LTL_f}$

Given a set Σ of proposition letters, a formula ϕ of $\mathsf{LTL_f}$ is defined as follows [5]:

$$\phi := p \mid \neg p \mid \phi \vee \phi \mid \phi \wedge \phi \qquad \text{Boolean connectives} \tag{1}$$

$$\mid \mathsf{X}\phi \mid \mathsf{wX}\phi \mid \phi \mathsf{U} \phi \mid \phi \mathsf{R} \phi \qquad \text{future modalities} \tag{2}$$

where $p \in \Sigma$. Note that the definition of the syntax of $\mathsf{LTL_f}$ generates formulas in *negated normal form* (NNF), that is with negations appearing only in front of proposition letters. The future temporal operators X, wX (also written $\tilde{\mathbf{X}}$), U, and R are called *tomorrow*, *weak tomorrow*, *until*, and *release*, respectively. We use the standard shortcuts for $\top \coloneqq p \vee \neg p$, $\bot \coloneqq p \wedge \neg p$ (for some $p \in \Sigma$) and for other temporal operators: $\mathsf{F}\phi \coloneqq \top \mathsf{U} \phi$ (called *eventually*), $\mathsf{G}\phi \coloneqq \bot \mathsf{R} \phi$ (called *globally*), $\phi_1 \mathsf{W} \phi_2 \coloneqq \phi_1 \mathsf{U} \phi_2 \vee \mathsf{G}\phi_1$ (called *weak until*).

For any formula ϕ, we denote with $|\phi|$ the *size* of ϕ, i.e. its number of symbols.

Formulas of $\mathsf{LTL_f}$ over the alphabet Σ are interpreted over *simple finite traces* (or state sequences, or words), i.e. sequences in the set Σ^+. In the following, we will write *general finite traces semantics* to denote the interpretation under this structures. Let $\sigma = \langle \sigma_0, \ldots, \sigma_{n-1} \rangle \in \Sigma^+$ be a finite trace. We define the *length* of σ as $|\sigma| = n$. With $\sigma_{[i,j]}$ (for some $0 \leq i \leq j < |\sigma|$) we denote the subinterval $\langle \sigma_i, \ldots, \sigma_j \rangle$ of σ. The *satisfaction* of an $\mathsf{LTL_f}$ formula ϕ by σ at time $0 \leq i < |\sigma|$, denoted by $\sigma, i \models \phi$, is defined as follows:

- $\sigma, i \models p$ iff $p = \sigma_i$;
- $\sigma, i \models \neg p$ iff $p \neq \sigma_i$;
- $\sigma, i \models \phi_1 || \phi_2$ iff $\sigma, i \models \phi_1$ or $\sigma, i \models \phi_2$;
- $\sigma, i \models \phi_1 \wedge \phi_2$ iff $\sigma, i \models \phi_1$ and $\sigma, i \models \phi_2$;
- $\sigma, i \models \mathsf{X}\phi$ iff $i + 1 < |\sigma|$ and $\sigma, i + 1 \models \phi$;
- $\sigma, i \models \mathsf{wX}\phi$ iff either $i + 1 = |\sigma|$ or $\sigma, i + 1 \models \phi$;
- $\sigma, i \models \phi_1 \mathsf{U} \phi_2$ iff there exists $i \leq j < |\sigma|$ such that $\sigma, j \models \phi_2$, and $\sigma, k \models \phi_1$ for all k, with $i \leq k < j$;
- $\sigma, i \models \phi_1 \mathsf{R} \phi_2$ iff either $\sigma, j \models \phi_2$ for all $i \leq j < |\sigma|$, or there exists $i \leq k < |\sigma|$ such that $\sigma, k \models \phi_1$ and $\sigma, j \models \phi_2$ for all $i \leq j \leq k$;

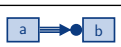
We say that σ is a *model* of ϕ (written as $\sigma \models \phi$) iff $\sigma, 0 \models \phi$. The *language* (of finite words) of ϕ, denoted by $\mathcal{L}(\phi)$, is the set of traces $\sigma \in \Sigma^+$ such that $\sigma \models \phi$. We say that two formulas $\phi, \psi \in \mathsf{LTL_f}$ are *equivalent* iff $\mathcal{L}(\phi) = \mathcal{L}(\psi)$.

$\mathsf{LTL_f}$ is used as the formal counterpart of for the well-known template-based DECLARE language. The complete table of the DECLARE templates as well as their characterisation is presented in Table 1.

Table 1: Supported DECLARE templates

Name	**Notation**	$\mathsf{LTL_f}$	**Description**
existence(a)		$\mathbf{F}(a)$	an event matching a must occur at least once
absence2(a)		$\neg\mathbf{F}(a \wedge \mathbf{XF}a)$	at most one event matching a can occur
choice(a,b)		$\mathbf{F}(a \vee b)$	at least one event matching a or b must occur
ex-choice(a,b)		$\mathbf{F}(a \vee b) \wedge$ $\neg(\mathbf{F}a \wedge \mathbf{F}b)$	at least one event matching either a or b must occur

Table 1: Supported DECLARE templates

Name	Notation	$\mathsf{LTL_f}$	Description
resp-existence(a,b)		$\mathbf{F}a \rightarrow \mathbf{F}b$	if an event matching a occurs, then an event matching b must also occur (either before or later)
coexistence(a,b)		$(\mathbf{F}a \rightarrow \mathbf{F}b) \wedge$ $(\mathbf{F}b \rightarrow \mathbf{F}a)$	either events matching a and b occur, or none of them occurs
response(a,b)		$\mathbf{G}(a \rightarrow \mathbf{F}b)$	whenever an event matching a occurs, then an event matching b must occur later
precedence(a,b)		$\neg b \mathbf{W} a$	an event matching b can only occur if an event matching a has already occurred
alt-response(a,b)		$\mathbf{G}(a \rightarrow \mathbf{X}(\neg(a \mathbf{U} b))$	every event matching a must be followed by an event matching b, without any occurrence of other events matching a in between
alt-precedence(a,b)		$\neg b \mathbf{W} a \wedge$ $\mathbf{G}(b \rightarrow \tilde{\mathbf{X}}(\neg b \mathbf{W} a))$	every event matching b must be preceded by an event matching a, without any other occurrence of events matching b in between
chain-response(a,b)		$\mathbf{G}(a \rightarrow \mathbf{X}b)$	every event matching a must be immediately followed by an event matching b
chain-precedence(a,b)		$\mathbf{G}(\mathbf{X}b \rightarrow a)$	events matching b can only occur immediately after events matching a occur
not-coexistence(a,b)		$\neg(\mathbf{F}a \wedge \mathbf{F}b)$	two events respectively matching a and b cannot both occur
neg-succession(a,b)		$\mathbf{G}(a \rightarrow \neg \mathbf{F}b)$	two events respectively matching a and b cannot occur one after the other
neg-chain-succ.(a,b)		$\mathbf{G}(a \rightarrow \tilde{\mathbf{X}} \neg b) \wedge$ $\mathbf{G}(b \rightarrow \tilde{\mathbf{X}} \neg a)$	two events respectively matching a and b cannot occur next to each other

1.3 Automata on Finite Words

We recall standard notions on finite–word automata, focusing on ε-NFAs, which naturally arise when representing the visible behaviour of workflow nets with silent transitions. Given a finite alphabet Σ, a *word* is a finite sequence $\sigma \in \Sigma^*$. All automata considered in this paper read finite words.

Definition 2 (ε-NFAs and DFAs). *An* ε-NFA *over alphabet* Σ *is a tuple* $\mathcal{A} = \langle \Sigma, Q, I, \delta, Acc \rangle$ *where: (i)* Q *is a finite set of states; (ii)* $I \subseteq Q$ *is the set of initial states; (iii)* $Acc \subseteq Q$ *is the set of accepting states; (iv)* $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$ *is the transition function.*

A *run* of $\mathcal{A}$ on a word $\sigma = a_1 \cdots a_n \in \Sigma^*$ is a sequence of states $\pi = q_0, q_1, \ldots, q_m$ such that $q_0 \in I$, and there exist indices $0 = i_0 \leq i_1 \leq \cdots \leq i_n = m$ such that for each $k \in \{1, \ldots, n\}$: for every j with $i_{k-1} \leq j < i_k$ we have $q_{j+1} \in \delta(q_j, \varepsilon)$ (any number of ε-moves); and $q_{i_k} \in \delta(q_{i_k - 1}, a_k)$.

The run π is *accepting* if $q_m \in Acc$. The language of $\mathcal{A}$, written $\mathcal{L}(\mathcal{A})$, is the set of all words admitting an accepting run.

A *deterministic* finite automaton (DFA) is an ε-NFA with $|I| = 1$ and $|\delta(q, a)| = 1$ for all q in Q and a in $\Sigma \cup \{\varepsilon\}$.

Every ε-NFA can be converted into a language–equivalent DFA by the standard ε-removal followed by the subset construction [8].

Proposition 2 (Determinization of ε-NFAs [8]). *For every ε-NFA $\mathcal{A}$ there exists a DFA $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$.*

Definition 3 (Product of DFAs [8]). *Given DFAs $\mathcal{A}_1 = \langle \Sigma, Q_1, q_1^0, \delta_1, Acc_1 \rangle$ and $\mathcal{A}_2 = \langle \Sigma, Q_2, q_2^0, \delta_2, Acc_2 \rangle$, their* product *is the DFA*

$$\mathcal{A}_1 \times \mathcal{A}_2 = \langle \Sigma, Q_1 \times Q_2, (q_1^0, q_2^0), \delta, Acc_1 \times Acc_2 \rangle,$$

where $\delta((q_1, q_2), a) = (\delta_1(q_1, a),\ \delta_2(q_2, a))$.

Definition 4 (ε-NFA of a workflow net). *Given a 1-safe workflow net N, its associated ε-NFA is $\mathcal{A}_N = \langle T_v, Q, I, \delta, Acc \rangle$, where: (i) $Q = \mathcal{R}(N)$ is the finite set of markings reachable from m_0; (ii) $I = \{m_0\}$ is the singleton initial set; (iii) $Acc = \{m \in \mathcal{R}(m_0) \mid m(out) = 1\}$ is the singleton accepting set; (iv) the transition function δ is defined by:*

$$m' \in \delta(m, a) \quad \Longleftrightarrow \quad \exists t \in T.\ m \xrightarrow{t} m' \text{ and } \begin{cases} a = \varepsilon & \text{if } t \in T_\tau, \\ a = t & \text{if } t \in T_v. \end{cases}$$

Intuitively, each marking of N is a state of the automaton; visible transitions become letter-labelled moves, and silent transitions become ε-moves.

We denote by $\rho|_{T_v}$ the sequence ρ after removing all elements that are not in T_v, i.e. removing all the silent transitions. We call this the *visible run*.

Let $\rho = t_1 \cdots t_k \in T^*$ and let $\rho|_{T_v} = a_1 \cdots a_n \in T_v^*$. Then there exist unique indices $1 \le j_1 < \cdots < j_n \le k$ such that: *(i)* $t_{j_\ell} = a_\ell \in T_v$ for all $\ell = 1, \ldots, n$; *(ii)* $t_i \in T_\tau$ for all $i \in \{1, \ldots, k\} \setminus \{j_1, \ldots, j_n\}$.

Proposition 3 (Language of $\mathcal{A}_N$). *The language of the ε-NFA $\mathcal{A}_N$ coincides with the visible language of WF-net N, that is,*

$$\mathcal{L}(\mathcal{A}_N) = \left\{ \rho|_{T_v} \mid \rho \in \mathcal{L}(N) \right\}.$$

Proof. ($\subseteq$) Let $\sigma \in \mathcal{L}(\mathcal{A}_N)$. Then there is an accepting run $m_0, m_1, \ldots, m_m$ of $\mathcal{A}_N$ on σ, starting in the initial marking and ending in a marking with $m(out) = 1$. By definition of δ in 4, each step of the run corresponds to firing either a silent transition (for ε-moves) or a visible transition labelled by the next symbol of σ. Collecting these transitions yields a firing sequence $\rho \in T^*$ such that $m_0 \xrightarrow{\rho} m_m$. By construction, the visible transitions in ρ appear exactly in the order of σ, i.e. $\rho|_{T_v} = \sigma$. Hence $\sigma = \rho|_{T_v}$ for some $\rho \in \mathcal{L}(N)$.

($\supseteq$) Let $\rho = t_1 \cdots t_k \in \mathcal{L}(N)$, with run $m_0 \xrightarrow{\rho} m_k$ and $m_k(out) = 1$. Let $\sigma = \rho|_{T_v}$ and let $a_1 \cdots a_n$ be its letters. Let the visible transitions occur at positions $j_1 < \cdots < j_n$. Now consider the marking sequence $m_0, \ldots, m_k$. Between consecutive visible steps, the transitions are silent, hence interpreted as ε-moves; at each j_ℓ the visible transition a_ℓ is interpreted as reading symbol a_ℓ. Thus $m_0, \ldots, m_k$ is a valid accepting run of $\mathcal{A}_N$ on σ. Hence $\sigma \in \mathcal{L}(\mathcal{A}_N)$.

Both inclusions hold, so the languages coincide.

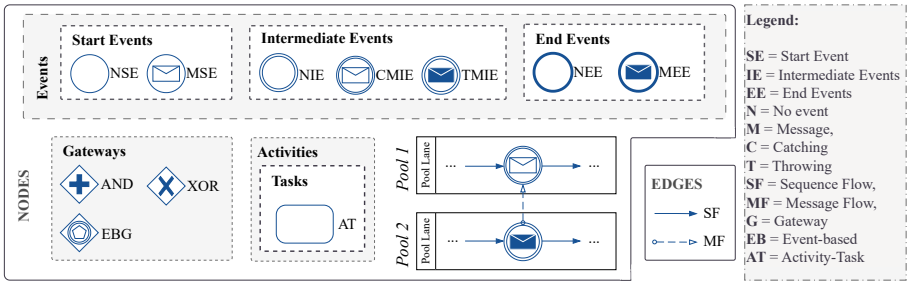


Fig. 1. A subset of BPMN considered in this paper

Proposition 4 ([5]). *For any formula* $\phi \in \mathsf{LTL}_f$*, there exists a* DFA $\mathcal{A}$ *such that* $\mathcal{L}(\phi) = \mathcal{L}(\mathcal{A})$*.*

2 Multi-model processes

To address the limitations of imperative process models, we embrace the Multi-model paradigm [2]. In particular, we extend BPMN collaboration diagrams [9] to support the definition of inter-process declarative constraints, by integrating it with the well-known DECLARE formalism [10].

The resulting multi-model collaboration (*MM-collaboration* for short)[3] diagrams present several advantages.

2.1 On the Marriage of BPMN and DECLARE

In this section, we present the main components of our *MM-notation*. We then explain how these components are combined to form the final notation.

BPMN. We adopt BPMN as the language to represent the imperative components of our MM-notation. Every process P is constructed using the elements from Figure 1. For simplicity, we assume that each process is well-formed, following the well-formedness defined in [7]. Moreover, we assume that each individual process of the collaboration is enclosed in a distinct BPMN pool.

By $\texttt{act}(P)$ we define the set of all unique activity identifiers used in P, and assume its natural extension to sets such that $\texttt{act}(\{P_1, \ldots, P_n\}) = \texttt{act}(P_1) \cup \ldots \cup \texttt{act}(P_n)$. Similarly, by $\texttt{ev}(P)$ we define the set of all events used in P, and by $\texttt{ev}(\{P_1, \ldots, P_n\}) = \texttt{ev}(P_1) \cup \ldots \cup \texttt{ev}(P_n)$ the set of all events used in n different process models. To correctly identify the type of a node x in a given BPMN graph, we use the function $\texttt{type}$. All the considered node types are shown in Figure 1. We use ✉ and ✉ to respectively denote intermediate message throw and catch event types. Multi-instance markers of BPMN [9] are not considered.

[3] For brevity, we shorten "multi-model" to "MM" in the remainder of this paper.

Declarative Collaboration Constraints. To represent the declarative collaboration constraints, we opt for the well-known template-based DECLARE formalism [10,6]. A DECLARE specification is a finite set of behavioral constraints that must hold throughout the system (or process) execution. Each constraint is based on a parametric unary $\varphi(x)$ or binary $\varphi(x, y)$ template, which is instantiated with elements from the finite set of activity names A.

DECLARE templates are formalised using Linear Temporal Logic over finite traces ($\mathsf{LTL_f}$) [4,6]. We use $\mathcal{C}(A)$ to denote the set of all unary and binary DECLARE constraints defined over A.

Given a finite sequence (a *trace*) $\sigma \in A^+$, a DECLARE specification $S \subseteq \mathcal{C}(A)$ is satisfied over σ, written as $\sigma \models S$, iff each constraint from S is satisfied on σ following the $\mathsf{LTL_f}$ semantics, i.e., $\sigma \models_{\mathsf{LTL_f}} \bigwedge_{c \in S} c$. Similarly, S is violated on σ, written as $\sigma \not\models S$, iff $\sigma \not\models_{\mathsf{LTL_f}} c$ for some $c \in S$. Since each constraint in S is an $\mathsf{LTL_f}$ formula, it can be represented as a finite automaton by Proposition 4, allowing for any form of automata-based analysis, including runtime monitoring [4,3]. In this setting, a constraint c is satisfied on σ if and only if the corresponding automata reaches an accepting state on σ.

MM-Collaboration Notation. Towards the goal of extending BPMN collaboration diagrams with more advanced interaction patterns defined using DECLARE, we propose the following formalization:

Definition 5. *An* MM-collaboration *is a tuple* $\mathcal{H} = (\mathcal{P}, F_{\mathtt{msg}}, C)$*, where:*

- $\mathcal{P} = \{P_1, \ldots, P_n\}$ *is a set of BPMN processes.*
- $F_{\mathtt{msg}} \subseteq \mathtt{ev}(\mathcal{P}) \times \mathtt{ev}(\mathcal{P})$ *is the message flow relation s.t.* $(e_1, e_2) \in F_{\mathtt{msg}}$ *iff* $e_1 \in \mathtt{ev}(P_i) \wedge \mathtt{type}(e_1) =$ ✉ *and* $e_2 \in \mathtt{ev}(P_j) \wedge \mathtt{type}(e_2) =$ ✉*, for* $i \neq j$*;*
- $C \subseteq \mathcal{C}\big(\mathtt{act}(\mathcal{P})\big)$ *is the set of* DECLARE *patterns over the activities of* $\mathcal{P}$ *s.t. for each binary pattern* $\varphi(t_1, t_2) \in C$ *it holds that* $t_1 \in \mathtt{act}(P_i)$ *and* $t_2 \in \mathtt{act}(P_j)$*, for* $i \neq j$*.*

2.2 Execution semantics

For the notation proposed in Definition 5, we introduce a few execution semantics that largely rely on the standard Petri-net-based one proposed in [7], and briefly reported in Figure 2 and Figure 3. More specifically, for each BPMN process model (without message flows) constructed using the elements introduced in Section 2.1, we can create a behaviorally equivalent Petri net. Without loss of generality, we assume that such net is a workflow net (WF-net) [1].

Like that, given an MM-collaboration $\mathcal{H} = (\mathcal{P}, F_{\mathtt{msg}}, C)$, it is possible to represent $(\mathcal{P}, F_{\mathtt{msg}})$ with a WF-net $N_{\mathcal{H}} = (P, T, F, in, out)$.

Notice that $N_{\mathcal{H}}$ captures unique activity identifiers (instead of their names) and thus is not labeled. The method from [7] can produce nets with silent transitions, for which we assume $T_\tau \subset T$. Moreover, as per Definition 5, it is easy to see that all non-silent transitions in T are those from $\mathtt{act}(\mathcal{P})$ (i.e., $\mathtt{act}(\mathcal{P}) \subseteq T$). We can thus obtain $N_{\mathcal{H}}$ as a union of WF-nets N_i produced for every $P_i \in \mathcal{P}$ and refined through additional places connected to some transitions from $\bigcup_{i=1}^{|\mathcal{P}|} T_i$ to

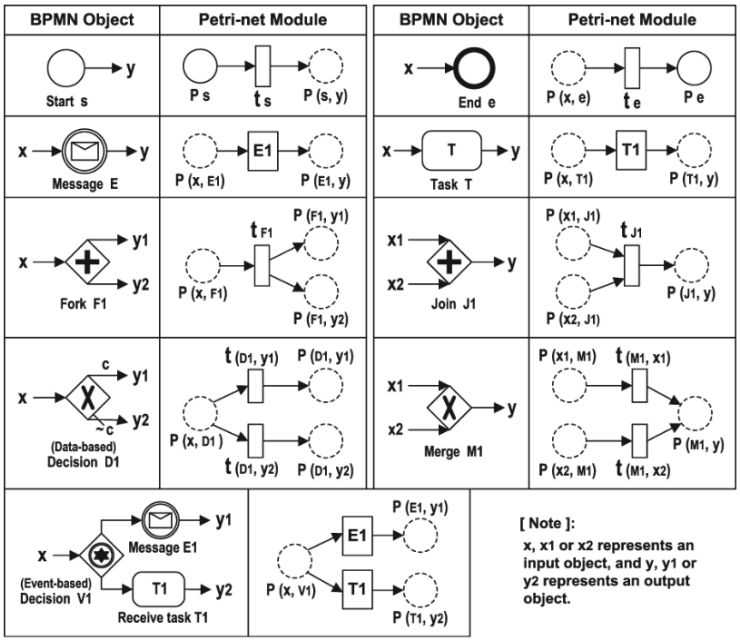


Fig. 2. Mapping task, events, and gateways onto Petri-net modules from [7]

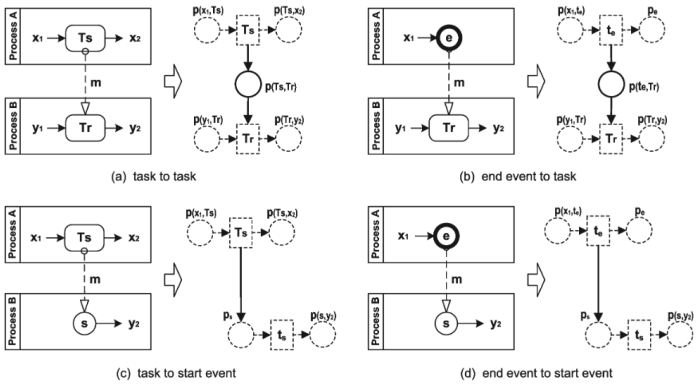


Fig. 3. Mappings from [7] for the message flows. Note that in this technical report, the message flow is defined between message events only, but the existing mapping can be seamlessly adapted to capture it.

capture $F_{\mathtt{msg}}$ following the message flow mapping described in [7] (reported in Figure 3). Moreover, multiple source (resp., sink) places of individual WF-nets are handled by simply adding a transition t to T_τ s.t. $(in, t) \in F$ and $(t, in_i) \in F$ (resp., $(t, out) \in F$ and $(out_i, t) \in F$) for every $P_i \in \mathcal{P}$.

We can now proceed with defining the operational semantics of MM-collaborations $(\mathcal{P}, F_{\mathtt{msg}}, C)$. As mentioned in Section 1.2, each $c \in C$ has an $\mathsf{LTL_f}$ formula as formal counterpart. By Proposition 4, c can be represented by the automaton $\mathcal{A}_c = (Q_c, \mathtt{act}(\mathcal{P}), \delta_c, q_c^0, Acc)$. We use this to define the states of MM-collaborations.

Definition 6. *Let $\mathcal{H} = (\mathcal{P}, F_{\mathtt{msg}}, \{c_1, \ldots, c_k\})$ be an MM-collaboration, $N_{\mathcal{H}}$ be the WF-net for $(\mathcal{P}, F_{\mathtt{msg}})$ and $\{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ are DFAs for $\{c_1, \ldots, c_k\}$. A* configuration *of $\mathcal{H}$ is a pair $s = (m, \mathbf{q})$, where m is a marking of $N_{\mathcal{H}}$ and $\mathbf{q} = \langle q_1, \ldots, q_k \rangle \in Q_1 \times \cdots \times Q_k$ is a combination of states of the constraint, where Q_i is the set of states of $\mathcal{A}_i$.*

Definition 7. *Let $\mathcal{H} = (\mathcal{P}, F_{\mathtt{msg}}, \{c_1, \ldots, c_k\})$ be an MM-collaboration with configuration $s = (m, \mathbf{q})$. Let t be a transition from the WF-net $N_{\mathcal{H}}$ for $(\mathcal{P}, F_{\mathtt{msg}})$. We say that t is* enabled in *s iff t is enabled in m on $N_{\mathcal{H}}$.*
If t is enabled, we can perform an MM-step *resulting in $s' = (m', \mathbf{q}')$, written as $(m, \mathbf{q}) \xrightarrow{t}_{\mathcal{H}} (m', \mathbf{q}')$, iff 1) $m \xrightarrow{t} m'$, and 2) $\mathbf{q}' = \mathbf{q}$, if $t \in T_\tau$, or $\mathbf{q}' = \langle q_1', \ldots, q_k' \rangle$ with $q_j' = \delta_j(q_j, t)$ for each $1 \leq j \leq k$, if $t \in \mathtt{act}(\mathcal{P})$.*

From the above notion of the MM-run, we can obtain a *(visible) process run* for $\mathcal{H}$ by removing from ρ all $t \in T_\tau$ while preserving the order. We denote it by $\rho|_{\mathtt{act}(\mathcal{P})}$. We say that an MM-run is *complete* if the final marking is covered (i.e., $m_n(out) = 1$); a run is *accepting* if it is a complete run, and all the constraints are satisfied (i.e., $\mathbf{q}_n \in Acc_1 \times \cdots \times Acc_k$).

Following the above definition of the operational semantics for MM-collaborations, we can observe that each constraint automaton $\mathcal{A}_j$ evolves synchronously with the visible transitions. Hence, we have $q_j^n = \delta_j^*(q_j^0, \rho|_{\mathtt{act}(\mathcal{P})})$, for all $1 \leq j \leq k$. It is easy then to see that any process run $\rho|_{\mathtt{act}(\mathcal{P})}$ over $N_{\mathcal{H}}$ coincides with a computation performed by $\{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ given $\rho|_{\mathtt{act}(\mathcal{P})}$ as input.

Lemma 1. *Let $\rho \in T^+$ be over $N_{\mathcal{H}}$, and $(m_0, \mathbf{q}^0)$ and $(m_n, \mathbf{q}^n)$ are two configurations. If $(m_0, \mathbf{q}^0) \xrightarrow{\rho}_{\mathcal{H}} (m_n, \mathbf{q}^n)$, then $\mathbf{q}^n = \langle \delta_1^*(q_1^0, \rho|_{\mathtt{act}(\mathcal{P})}), \ldots, \delta_k^*(q_k^0, \rho|_{\mathtt{act}(\mathcal{P})}) \rangle$.*

The proof is trivial derived directly from the definition of process run and Definition 7. The lemma guarantees that visible parts of traces produced by $(\mathcal{P}, F_{\mathtt{msg}})$ can be always correctly processed by the automata of the constraints from C. However, in this semantics we allow the collaboration to continue even if some of the constraints from C are violated (i.e., $\rho|_{\mathtt{act}(\mathcal{P})}$ brings some $\mathcal{A}_j$ to a state different from those in Acc_j). Any possible violations are not lost, but, instead, simply "recorded" in the constraint automata.Like that, one can view a BPMN collaboration as an event generator that runs in parallel with a set of DECLARE monitors, which change their states based on the received events. We call this semantics ***permissive***.

2.3 Anticipatory semantics

However, being permissive may not be entirely desired by the modeler. In that case, the collaboration constraints can be enforced, allowing to avoid violating executions by stopping the entire collaboration the moment a constraint becomes violated. To achieve this, we start by modifying the enabledness condition from Definition 7, allowing only for transitions whose firing in configuration s does not violate any of the constraints from C in all possible future configurations.

Definition 8. *Let $\mathcal{H} = (\mathcal{P}, F_{\mathtt{msg}}, \{c_1, \ldots, c_k\})$ be an MM-collaboration, $s = (m, \mathbf{q})$ a configuration of $\mathcal{H}$, $\rho \in T^*$ be a MM-run, such that $(m_0, \mathbf{q}^0) \xrightarrow{\rho}_{\mathcal{H}} (m, \mathbf{q})$, and $t \in T$. We say that t* is a violating transition in *s if for all $c \in C$ there is no $\rho' \in T^*$ such that (1) $(m, \mathbf{q}) \xrightarrow{t \cdot \rho'}_{\mathcal{H}} (m', \mathbf{q}')$ and $m'(out) = 1$, and (2) $\delta^*_{\mathcal{A}_c}(q^0_{\mathcal{A}_c}, (\rho t \rho')|_{\mathtt{act}(\mathcal{P})}) \in Acc_{\mathcal{A}_c}$.*

Condition *(1)* above says that the final state of $N_{\mathcal{H}}$ is reachable starting from m, and the corresponding sequence of MM-steps must start with t, while condition *(2)* means that it is possible to reach an accepting state using $(\rho t \rho')|_{\mathtt{act}(\mathcal{P})}$.

Adding the above violation check to the Definition 7 (specifically, t must be enabled and non-violating to perform a MM-step with t) allows us to turn our permissive semantics into a ***anticipatory*** one. This semantics goes beyond simply monitoring the collaboration constraints and, instead, does not allow the collaboration to execute an activity that, if executed, would lead to a certain violation of a collaboration constraint.

The variant of anticipatory semantics thus can be used in scenarios where the collaboration constraints are fully prescriptive and their violation requires an outside intervention. We denote by $(m, \mathbf{q}) \overset{t}{\Rightarrow}_{\mathcal{H}} (m', \mathbf{q}')$ MM-steps performed under the modified enabledness condition as specified above.

Proposition 5. *Let $\mathcal{H}$ be the MM-collaboration. The set of all complete MM-runs ρ of $\mathcal{H}$ under the anticipatory semantics coincides with the set of accepting runs of $\mathcal{H}$.*

Automata for process runs The externally observable behavior of the collaboration is completely determined by the *visible process run* generated by an underlying run of $N_{\mathcal{H}}$. Since constraint satisfaction depends only on process runs, it is natural to abstract away from markings and work at the level of such visible runs. This motivates the introduction of finite automata that capture exactly the set of visible process runs produced by $N_{\mathcal{H}}$ and the constraint automata, and which provide a finite representation of the future behavior relevant for the anticipatory semantics.

Definition 9. *Let $\mathcal{A}^D_{N_{\mathcal{H}}}$ be the DFA obtained from determinizing $\mathcal{A}_{N_{\mathcal{H}}}$. The (deterministic)* MM-automaton *is the product automaton*

$$\mathcal{A}_{\mathcal{H}} = \mathcal{A}^D_{N_{\mathcal{H}}} \times \mathcal{A}_1 \times \cdots \times \mathcal{A}_k.$$

Proposition 6. *The language of $\mathcal{A}_\mathcal{H}$ is exactly the set of complete process runs of $\mathcal{H}$ under anticipatory semantics. Formally,*

$$\mathcal{L}(\mathcal{A}_\mathcal{H}) = \left\{ \rho|_{\mathtt{act}(\mathcal{P})} \mid (m_0, \mathbf{q}^0) \overset{\rho}{\Rightarrow}_\mathcal{H} (m, \mathbf{q}), m(out) = 1 \right\}.$$

Theorem 1. *The set of accepting runs of $\mathcal{H}$ under the standard semantics is the same as the set of complete runs of $\mathcal{H}$ under the anticipatory semantics; their corresponding set of process runs is the same as the language of $\mathcal{A}_\mathcal{H}$. Formally,*

$$S = \left\{ \rho \mid (m_0, \mathbf{q}^0) \xrightarrow{\rho} (m, \boldsymbol{q}) \textit{ is accepting} \right\} = \left\{ \rho \mid (m_0, \mathbf{q}^0) \overset{\rho}{\Rightarrow} (m, \boldsymbol{q}) \textit{ is complete} \right\},$$

$$\textit{and } \left\{ \rho|_{\mathtt{act}(\mathcal{P})} \mid \rho \in S \right\} = \mathcal{L}(\mathcal{A}_\mathcal{H}).$$

Proof. Direct consequence of Proposition 5 and Proposition 6.

This theorem gives us an overview over the different semantics and the language of the hybrid automaton.

3 Implementing the Operational Semantics

3.1 Overview and 5-valued coloring

Intuitively, anticipatory semantics induces a binary judgment on visible prefixes: a run $\sigma \in T^+$ is ruled out as soon as no accepting process run can exist with $\sigma|_{\mathtt{act}(\mathcal{P})}$ as its prefix, and is allowed otherwise. Then, to check if there exists a complete process run having a prefix whose corresponding process run is σ, it is enough to check whether a process run σ can be extended to reach a final state in $\mathcal{A}_\mathcal{H}$ (this follows from Definition 8). While this is sufficient for early rejection, the space of possible completions of σ often carries more information.

To make this explicit, consider all complete MM-runs whose process run extends σ. We can thus use an approach similar to the one adopted in monitoring DECLARE constraints [6] with which $\mathcal{A}_\mathcal{H}$ is "colored" using the following schema and perceived under the permissive execution semantics so that, for example, violations are not explicitly removed from the automaton but can be anticipated. This additional information is provided according to the following classification:

- σ is `Satisfied` if every such complete run satisfies all constraints;
- σ is `Violated` if none of them does;
- σ is `Inconclusive` if both satisfying and violating completions remain possible and σ is not itself the process run obtained from some complete run;
- σ is `Temporarily Satisfied` if σ already corresponds to the process run obtained from some accepting run while it can still be extended to a process run obtained from a complete run and violating some of the costraints;
- σ is `Temporarily Violated` if the symmetrical condition to temporary satisfaction holds.

The above yields a five-way coloring (or refinement) in which the classical "violating" prefixes form one class and the remaining four distinguish the different ways in which a prefix can still lead to satisfaction.

This refined classification can be computed directly on the MM-automaton $\mathcal{A}_{\mathcal{H}}$.

Let p be a state of the MM-automaton $\mathcal{A}_{\mathcal{H}}$ reached after process run prefix σ, and write

$$\mathsf{Comp}(p) := \{\sigma \in T_v^* \mid \delta^*_{\mathcal{H}}(p,\sigma) \in Acc_{\mathcal{A}^D_{N_{\mathcal{H}}}} \times Q_1 \times \cdots \times Q_k\}$$

for the set of prefixes that would form a complete process run from p (that is, regardless of the constraints). A completion $\sigma \in \mathsf{Comp}(p)$ is *satisfying* if $\delta^*_{\mathcal{H}}(p,\sigma)$ satisfies all constraints; otherwise it is *violating*. We set

$$\mathsf{Sat}(p) = \{\sigma \mid \delta^*_{\mathcal{H}}(p,\sigma) \in Acc_{\mathcal{A}_{\mathcal{H}}}\}, \qquad \mathsf{Viol}(p) = \mathsf{Comp}(p) \setminus \mathsf{Sat}(p).$$

Notice that $\mathsf{Sat}(p) \subseteq \mathsf{Comp}(p)$. This implies, in particular, that if $\varepsilon \in \mathsf{Comp}(p)$. then either $\epsilon \in \mathsf{Sat}(p)$ or $\epsilon \in \mathsf{Viol}(p)$. Every state falls into exactly one of the following categories:

- `Satisfied`: $\mathsf{Sat}(p) \neq \varnothing$ and $\mathsf{Viol}(p) = \varnothing$.
- `Violated`: $\mathsf{Sat}(p) = \varnothing$.
- `Inconclusive`: $\mathsf{Sat}(p) \neq \varnothing$, $\mathsf{Viol}(p) \neq \varnothing$, and $\varepsilon \notin \mathsf{Comp}(p)$.
- `Temporarily Satisfied`: $\varepsilon \in \mathsf{Sat}(p)$ and $\mathsf{Viol}(p) \neq \varnothing$.
- `Temporarily Violated`: $\varepsilon \in \mathsf{Viol}(p)$ and $\mathsf{Sat}(p) \neq \varnothing$.

The classification can be visualized in Figure 4.

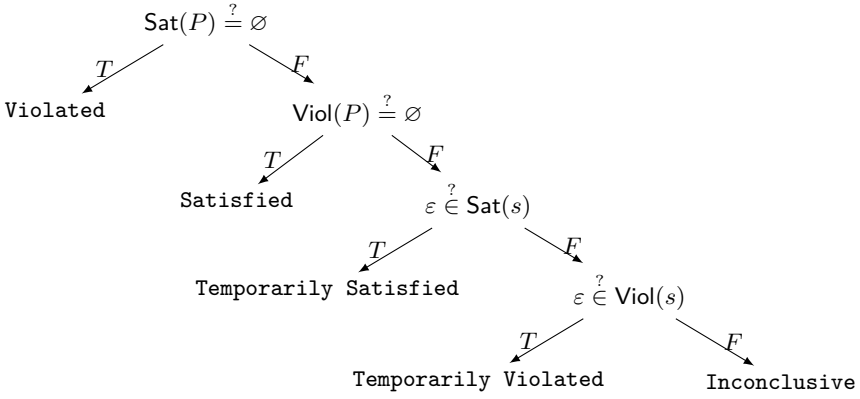


Fig. 4. Visual representation of the classification of the

3.2 Algorithms

Preliminaries. Recall that the MM-automaton $\mathcal{A}_{\mathcal{H}} = \mathcal{A}^D_{N_{\mathcal{H}}} \times \mathcal{A}_1 \times \cdots \times \mathcal{A}_k$ is a DFA over the alphabet T_v. Let $Q_{\mathcal{H}}$ be its set of states and $\delta_{\mathcal{H}} : Q_{\mathcal{H}} \times T_v \to Q_{\mathcal{H}}$ its transition function. We denote by $Acc_{\mathcal{A}^D_{N_{\mathcal{H}}}}$ the accepting states of the deterministic process automaton $\mathcal{A}^D_{N_{\mathcal{H}}}$, and by $Acc_{\mathcal{A}_{\mathcal{H}}}$ the accepting states of the full MM-automaton, i.e. those in which the process has finished and all constraints are satisfied.

We use these to identify, inside $Q_{\mathcal{H}}$, the states that represent completed process runs, distinguishing whether the constraints are satisfied or not:

$$\begin{aligned} \mathit{ProcFinal} &:= Acc_{\mathcal{A}^D_{N_{\mathcal{H}}}} \times Q_1 \times \cdots \times Q_k, \\ \mathit{SatFinal} &:= Acc_{\mathcal{A}_{\mathcal{H}}}(\subseteq \mathit{ProcFinal}), \\ \mathit{ViolFinal} &:= \mathit{ProcFinal} \setminus \mathit{SatFinal}. \end{aligned}$$

Intuitively, *SatFinal* contains all states reached by visible traces that correspond to complete runs satisfying all constraints, while *ViolFinal* contains those reached by complete runs that violate at least one constraint.

Backward reachability. To determine, for each state $p \in Q_{\mathcal{H}}$, whether it admits satisfying and/or violating completions, we perform two backward graph searches on the transition graph of $\mathcal{A}_{\mathcal{H}}$ (ignoring labels).

Let $E \subseteq Q_{\mathcal{H}} \times Q_{\mathcal{H}}$ be the edge relation induced by $\delta_{\mathcal{H}}$, i.e. $(p, p') \in E$ iff $\exists a \in T_v.\ \delta_{\mathcal{H}}(p, a) = p'$. We also consider the reverse edge relation $E^{-1} = \{(p', p) \mid (p, p') \in E\}$.

We compute two sets of states:

- SatReach: the set of states from which some state in *SatFinal* is reachable;
- ViolReach: the set of states from which some state in *ViolFinal* is reachable.

Both sets are computed by standard breadth-first search (BFS) on the reverse graph:

1. Initialisation:
   - SatReach := *SatFinal*;
   - ViolReach := *ViolFinal*.
2. For each set $X \in \{\mathsf{SatReach}, \mathsf{ViolReach}\}$: repeatedly add to X every predecessor p such that there exists $p' \in X$ with $(p, p') \in E$, until a fixpoint is reached.

By construction, a state p belongs to SatReach iff there is a path in $\mathcal{A}_{\mathcal{H}}$ from p to some state in *SatFinal*; analogously for ViolReach and *ViolFinal*.

Coloring algorithm. Once SatReach and ViolReach have been computed, the five-valued coloring is obtained by a simple pass over all states $p \in Q_{\mathcal{H}}$. We assign to each state p exactly one of the five labels as follows:

- `Violated` if $p \notin$ SatReach.

- `Satisfied` if $p \in \mathsf{SatReach}$ and $p \notin \mathsf{ViolReach}$.
- Otherwise $p \in \mathsf{SatReach} \cap \mathsf{ViolReach}$, so both satisfying and violating completions are possible. In this case:
  - if $p \in \mathit{SatFinal}$, then p is `Temporarily Satisfied`;
  - if $p \in \mathit{ViolFinal}$, then p is `Temporarily Violated`.
  - else p is `Inconclusive`;

By inspection of the cases, every state receives exactly one label, and the five coloring classes form a partition of $Q_{\mathcal{H}}$. The overall computational cost is linear in the size of the MM-automaton: each backward search runs in time $\mathcal{O}(|Q_{\mathcal{H}}| + |E|)$, and the final classification pass is $\mathcal{O}(|Q_{\mathcal{H}}|)$. The dominant cost in practice typically lies in constructing $\mathcal{A}^{D}_{N_{\mathcal{H}}}$ and $\mathcal{A}_{\mathcal{H}}$ themselves rather than in the coloring algorithm.

References

1. van der Aalst, W.M.P.: Workflow verification: Finding control-flow errors using petri-net-based techniques. In: Business Process Management. Lecture Notes in Computer Science, vol. 1806, pp. 161–183. Springer (2000)
2. Alman, A., Maggi, F.M., Rinderle-Ma, S., Rivkin, A., Winter, K.: Towards a multi-model paradigm for business process management. In: CAiSE. Lecture Notes in Computer Science, vol. 14663, pp. 178–194. Springer (2024)
3. De Giacomo, G., De Masellis, R., Maria Maggi, F., Montali, M.: Monitoring constraints and metaconstraints with temporal logics on finite traces. ACM Trans. Softw. Eng. Methodol. **31**(4), 68:1–68:44 (2022)
4. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: Rossi, F. (ed.) Proc. of IJCAI. pp. 854–860. IJCAI/AAAI (2013)
5. De Giacomo, G., Vardi, M.Y., et al.: Linear temporal logic and linear dynamic logic on finite traces. In: Ijcai. vol. 13, pp. 854–860 (2013)
6. Di Ciccio, C., Montali, M.: Declarative process specifications: Reasoning, discovery, monitoring. In: Process Mining Handbook, Lecture Notes in Business Information Processing, vol. 448, pp. 108–152. Springer (2022)
7. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Inf. Softw. Technol. **50**(12), 1281–1294 (2008)
8. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc., USA (2006)
9. OMG: Business process model and notation (BPMN). Technical report, Object Management Group (2014), `https://www.omg.org/spec/BPMN/2.0.2/PDF`
10. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: DECLARE: full support for loosely-structured processes. In: EDOC. pp. 287–300. IEEE CS (2007)