Monitoring Hybrid Process Specifications with Conflict Management (Extended Version)

Anti Alman*, Fabrizio Maria Maggi†, Marco Montali†, Fabio Patrizi‡ and Andrey Rivkin†
*University of Tartu, Tartu, Estonia
Email: anti.alman@ut.ee
†Free University of Bozen-Bolzano, Bolzano, Italy
Email: {maggi,montali,andrey}@inf.unibz.it
‡Sapienza University of Rome, Rome, Italy
Email: patrizi@diag.uniroma1.it

***Abstract*—Business process monitoring approaches have thus far mainly focused on monitoring the execution of a process with respect to a single process model. However, in some cases it is necessary to consider multiple process specifications simultaneously. In addition, these specifications can be procedural, declarative, or a combination of both. For example, in the medical domain, a clinical guideline describing the treatment of a specific disease cannot account for all possible co-factors that can coexist for a specific patient and therefore additional constraints may need to be considered. In some cases, these constraints may be incompatible with the clinical guideline, therefore requiring the violation of either the guideline or the constraints. In this paper, we propose a solution for monitoring the interplay of hybrid process specifications that can be expressed as a combination of (data-aware) Petri nets and temporal logic rules. During the process execution, if these specifications are in conflict with each other, it is possible to violate some of them. The monitoring system is equipped with a violation cost model according to which the system provides recommendations about the next course of actions in a way that would either avoid possible violations or minimize the total cost of violations.**

***Index Terms*—Business Process Monitoring, Data Petri Net, Declare, Automaton, Model Interplay, Hybrid Process**

I. Introduction

A key functionality of any process-aware information system is *monitoring* [9]. Monitoring concerns the ability to verify at runtime whether an ongoing process execution conforms to the corresponding process model. This runtime form of conformance checking allows to detect, and therefore handle, deviations appearing in ongoing process instances. However, in several scenarios, different process specifications must be valid during the process execution and the monitoring system should take into consideration all of them and their interplay.

One such scenario would be the treatment of a patient having co-morbid conditions. In this case, the standard treatment procedures for each condition can be specified using procedural models, while additional knowledge, such as harmful drug interactions, can be specified using declarative constraints [2]. Note that, the interplay of process specifications can generate conflicts during the process execution [2], [18]. For example, by making a decision that, based on a procedural model, will lead to administering a drug that the patient is allergic to. To be able to take informed decisions in these situations, experts responsible for the execution of such process(es) need to be promptly alerted about the presence of conflicts.

In this paper, we present a monitoring approach with respect to multiple process specifications, each of which may also include conditions on the data perspective. In particular, we use data Petri nets (DPNs) [13] for procedural models and Linear Temporal Logic over finite traces (LTL_f) [7] for declarative models (additionally supporting the LTL_f based modeling language MP-Declare [17], [3]). This allows us to capture sophisticated forms of scoping and interaction among the different process specifications (that is, the different elicited DPNs and declarative constraints), going beyond what is captured so far in the literature, and providing a full logic-based characterization of the so-resulting hybrid processes [2], [18], [19]. These models differ from loosely coupled hybrid models [1], in that the different process specifications all interact with each other at the same level of abstraction.

Of particular importance, in our monitoring approach, is the early detection of conflicts among process specifications, arising when the process is in a state where at least one specification will eventually be violated. This aspect has been considered before with a purely declarative approach [11], [12], [14] but never applied to a hybrid setting.

In our context, there are three main novel challenges that need to be addressed. First and foremost, we need to tackle the infinity induced by the presence of data, which in general leads to undecidability of monitoring. In our specific setting, we show that we can recast data abstraction techniques studied for verification of DPNs [4], [5] so as to produce finitely representable monitors based on finite-state automata. Second, we need to homogeneously construct monitors for constraints and DPNs, and define how to combine them into a unique, global monitor for conflict detection; we do so by recasting the standard notion of automata product, producing a global monitor that conceptually captures a hybrid model where DPNs and constraints are all simultaneously applied (i.e. all DPNs are executed concurrently, while checking the validity of constraints). Third, we need to handle situations where the global monitor returns a permanent violation (due to the explicit violation of a process specification, or the presence of a conflict), but distinguishing among different continuations

is still relevant as they may lead to violate different process specifications. Assuming a violation cost is given for each specification, we show how to augment our monitors with the ability of returning the best-possible next events, that is, those keeping the overall violation cost at the minimum possible.

The proposed approach is implemented as a standalone application that takes as input the process specifications, violation costs, and an event log. After each event in the event log, the application finds the next events and attribute values leading to the minimum total cost of violations, effectively functioning as a recommendation engine. Additionally, the state of the process is monitored both at the level individual models and at the level of overall satisfiability.

The remainder of this paper is structured as follows. Section II provides an example monitoring scenario. Section III and Section IV introduce the necessary preliminaries and the monitoring approach respectively. Section V presents an evaluation using a prototype implementation. Finally, Section VI concludes the paper.

II. Example Scenario

Consider the following real-life scenario, where a patient with co-morbidities is simultaneously treated with different guidelines: a guideline for peptic ulcer (PU) and a guideline for venous thromboembolism (VT). More specifically, we are considering two tiny, yet relevant fragments of the guideline models presented in [18]. The two fragments are represented in Fig. 1 using DPNs (recalled in Section III).

When PU starts, the helicobacter pylori test is executed. Based on the test result, different therapies are chosen: amoxicillin administration in case of positive test, gastric acidity reduction otherwise. Afterwards, the peptic ulcer is evaluated to estimate the effects of the therapy.

VT requires an immediate intervention, chosen among three different possibilities based on the situation of the specific patient at hand. Mechanical intervention uses devices that prevent the proximal propagation or embolization of the thrombus into the pulmonary circulation, or involves the removal of the thrombus. The other two possibilities are an anticoagulant therapy based on warfarin, or a thrombolytic therapy.

The interaction between amoxicillin therapy (in the PU procedure) and warfarin therapy (in the VT procedure) is usually avoided in medical practice, since amoxicillin increases the anticoagulant effect of warfarin, raising the risk of bleedings. Therefore, in cases where the PU and VT procedures are performed simultaneously, the medical practice suggests specifying that *amoxicillin therapy* and *warfarin therapy* cannot coexist (declarative constraint C). Such a constraint C is an example of background medical knowledge rule [2].

Based on these specifications, if helicobacter pylori is tested positive and anticoaugulant is chosen to deal with venous thromboembolism, then there is a conflict between C and the two guidelines. In this outlier, but possible situation there would be three alternatives:

1) Violating PU (by skipping the amoxicillin therapy);
2) Violating VT (by using an alternative anticoagulant);
3) Violating C (giving priority to the two guidelines).

Informing medical experts about the presence of a conflict is crucial to help them in assessing the current situation, ponder the implications of one choice over the others, and finally make an informed decision. One can go beyond mere information, by also presenting the violation severity of the different alternatives. This can be done by assigning violation costs to the process specifications (in our case, PU, VT, and C). In this case, we can assume that skipping the amoxicillin therapy is rather costly, given the lack of viable alternatives for treating peptic ulcer in case of helicobacter pylori. Instead, violating the VT procedure has a lower cost, given the existence of other anticoagulants (e.g., heparin) that may be less effective but do not interact strongly with amoxicillin. Additionally, constraint C comes with the highest violation cost as complications such as serious bleeding should definitely be avoided.

In our approach, we can also deal with more sophisticated (meta-)constraints [6] that impose conditions on the process execution depending on the truth value of other constraints. Within our example scenario, we can for example specify a meta-constraint dictacting that if constraint C gets violated, then at least we expect that *warfarin therapy* is executed *after amoxicillin therapy*, to reduce the risk of a harmful interaction of warfarin and amoxicillin.

III. Process Components

In this section, we define the models used to specify declarative and procedural data-aware process components, by relying on Multi Perspective-Declare (MP-Declare) [3], [14], [10]) and data Petri nets (DPNs [13], [4]).

A. Events and Conditions

We start by fixing some preliminary notions related to events and traces. An *event signature* is a tuple $\langle n, A\rangle$, where: n is the *activity* name and $A = \{a_1, \ldots, a_\ell\}$ is the set of event *attribute (names)*. We assume a finite set $\mathcal{E}$ of event signatures, each having a distinct name (thus we can simply refer to an event signature $\langle n, A\rangle$ using its name n). By $\mathcal{N}_\mathcal{E} = \bigcup_{\langle n,A\rangle\in\mathcal{E}} n$ we denote the set of all event names from $\mathcal{E}$ and by $\mathcal{A}_\mathcal{E} = \bigcup_{\langle n,A\rangle\in\mathcal{E}} A$ the set of all attribute names occurring in $\mathcal{E}$.

Each event comes with a name matching one of the names in $\mathcal{N}_\mathcal{E}$, and provides actual values for the attributes of the corresponding signature. In the context of this paper, attributes range over reals equipped with comparison predicates (simpler types such as strings with equality and booleans can be seamlessly encoded).

Definition 1 (Event): An *event* of event signature $\langle n, A\rangle$ is a pair $e = \langle n, \nu\rangle$ where $\nu : A \mapsto \mathbb{R}$ is a total function assigning a real value to each attribute in A.

As usual, sequences of events form (process) traces.

Definition 2: A *trace* over a set $\mathcal{E}$ of event signatures is a finite sequence $\sigma = e_1 \cdots e_\ell$, where each e_i is an event of some signature in $\mathcal{E}$.

By $|\sigma|$ we denote the *length* of σ. For $1 \leq i \leq |\sigma|$, we define $\sigma(i) \doteq e_i$.

B. Multi-Perspective Declare with Local Conditions

To represent declarative process components, we resort to a multi-perspective variant of the well-known Declare language [17]. A Declare model describes *constraints* that must be satisfied throughout the process execution. Constraints, in turn, are based on *templates*. Templates are patterns that define parameterized classes of properties, and constraints are their concrete instantiations. The template semantics is formalized using Linear Temporal Logic over finite traces (LTL_f) [15].

In this work, we consider temporal constraints enriched with boolean combinations of attribute-to-constant comparisons. The resulting language closely resembles that of variable-to-constant conditions in [4], thus providing a good basis for combining declarative constraints with procedural models expressed with DPNs.

Definition 3: A condition φ over a set $\mathcal{E}$ of event signatures is an expression of the form:

$$\varphi := x \mid a \odot c \mid \neg\varphi \mid \varphi \wedge \varphi,$$

where: $x \in \mathcal{N}_\mathcal{E}$; $a \in \mathcal{A}_\mathcal{E}$; $\odot \in \{<, =, >\}$; $c \in \mathbb{R}$.
Conditions of the form $a \odot c$ and x are called *atomic*. We define the usual abbreviations: $\varphi_1 \vee \varphi_2 \doteq \neg(\neg\varphi_1 \wedge \neg\varphi_2)$; $\varphi_1 \rightarrow \varphi_2 \doteq \neg\varphi_1 \vee \varphi_2$; $a \leq c \doteq \neg(a > c)$; $a \geq c \doteq \neg(a < c)$; and $a \neq c \doteq \neg(a = c)$. In addition, we denote by $\mathcal{L}_\mathcal{E}$ the language of conditions over $\mathcal{E}$.

Conditions of $\mathcal{L}_\mathcal{E}$ are interpreted over events as follows.

Definition 4: We inductively define when a condition φ is *satisfied* by an event $e = \langle n, \nu \rangle$, written $e \models \varphi$, as follows:

- $e \models x$ iff $x = n$;
- $e \models a \odot c$ iff $\nu(a)$ is defined and $\nu(a) \odot c$;
- $e \models \neg\varphi$ iff $e \not\models \varphi$;
- $e \models \varphi_1 \wedge \varphi_2$ iff $e \models \varphi_1$ and $e \models \varphi_2$.

We are now ready to define LMP-Declare constraints, that is, MP-DECLARE constraints with local conditions. Their syntactic and semantic definition basically corresponds to that of LTL_f formulae with conditions as atomic formulae, interpreted over traces of the form given in Definition 2.

Definition 5: An *LMP-Declare constraint* is an expression of the form:

$$\Phi := \top \mid \varphi \mid \mathbf{X}\,\Phi \mid \Phi_1 \mathbf{U}\,\Phi_2 \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2$$

where φ is a condition from $\mathcal{L}_\mathcal{E}$ (cf. Definition 3).
As in standard LTL_f, $\mathbf{X}$ denotes the *strong next* operator (which requires the existence of a next state where the inner formula holds), while $\mathbf{U}$ stands for *strong until* (which requires the right-hand formula to eventually hold, forcing the left-hand formula to hold in all intermediate states).

Definition 6: We inductively define when an LMP-Declare constraint Φ is *satisfied* by a trace σ at position $1 \leq i \leq |\sigma|$, written $\sigma, i \models \Phi$, as follows:

- $\sigma, i \models \top$;
- $\sigma, i \models \varphi$ iff $\sigma(i) \models \varphi$ according to Definition 4;
- $\sigma, i \models \Phi_1 \wedge \Phi_2$ iff $\sigma, i \models \Phi_1$ and $\sigma, i \models \Phi_2$;
- $\sigma, i \models \neg\Phi$ iff $\sigma, i \not\models \Phi_1$;
- $\sigma, i \models \mathbf{X}\,\Phi$ iff $i < |\sigma|$ and $\sigma, i+1 \models \Phi$;
- $\sigma, i \models \Phi_1 \mathbf{U} \Phi_2$ iff there exists j, $1 \leq j \leq |\sigma|$, s.t. $\sigma, j \models \Phi_2$ and for every k, $1 \leq k \leq j-1$, we have $\sigma, k \models \Phi_1$.

We define the usual abbreviations: $\Phi_1 \vee \Phi_2 \doteq \neg(\neg\Phi_1 \wedge \neg\Phi_2)$; $\Phi_1 \rightarrow \Phi_2 \doteq \neg\Phi_1 \vee \Phi_2$; $\mathbf{F}\,\Phi = \mathit{true}\mathbf{U}\,\Phi$ (*eventually*); and $\mathbf{G}\,\Phi = \neg\mathbf{F}\,\neg\Phi$ (*globally*).

Example 1: Consider two event signatures $\langle \mathtt{a}, \{x, y\} \rangle$ and $\langle \mathtt{b}, \{z\} \rangle$. The *negation response* LMP-Declare constraint

$$\mathbf{G}(\mathtt{a} \rightarrow \neg\mathbf{X}\mathbf{F}(\mathtt{b} \wedge z > 10))$$

captures that whenever event $\mathtt{a}$ occurs then $\mathtt{b}$ cannot later occur with its attribute z carrying a value greater than 10.

C. Data Petri nets

We define data Petri nets (DPNs) by adjusting [13], [4] to our needs. In particular, our definition needs to accommodate the fact that a monitored trace will be matched against multiple process components (which will be the focus of Section IV).

Let $\mathcal{E}$ be a finite set of event signatures. The language $\mathcal{G}_\mathcal{E}$ of *guards* γ over $\mathcal{E}$ is defined as follows:

$$\gamma := a \odot c \mid \neg\gamma \mid \gamma_1 \wedge \gamma_2.$$

Observe that $\mathcal{G}_\mathcal{E}$ is the sub-language of conditions over $\mathcal{E}$, i.e., $\mathcal{L}_\mathcal{E}$, with formulas γ not mentioning event names. We can then specialize the notion of satisfaction to guards, by considering only the assignment to the event attributes. Namely, given an assignment $\alpha : \mathcal{A}_\mathcal{E} \rightarrow \mathbb{R}$ and an atomic condition $a \odot c$, we have that $\alpha \models a \odot c$ iff $\alpha(a) \odot c$. Boolean combinations of atomic conditions are defined as usual. Given a condition γ, we denote by *Var*(γ) the set of attributes mentioned therein.

Definition 7: A *Petri net with data and variable-to-constant conditions* (DPN) over a set $\mathcal{E}$ of event signatures is a tuple $D = \langle P, T, F, l, V, r, w \rangle$, where:

- P and T are two finite disjoint sets of *places* and *transitions*, respectively;
- $F : (P \times T) \cup (T \times P) \rightarrow \mathbb{N}$ is the net's *flow relation*;
- $l : T \rightarrow \mathcal{N}_\mathcal{E} \cup \{\tau\}$ is a total *labeling* function assigning a label from $\mathcal{N}_\mathcal{E} \cup \{\tau\}$ to every transition $t \in T$, with τ denoting a *silent* transition.
- $V \subseteq \mathcal{A}_\mathcal{E}$ is the set of net's *variables*;
- $r : T \rightarrow \mathcal{G}_\mathcal{E}$ and $w : T \rightarrow \mathcal{G}_\mathcal{E}$ are two total *read* and *write guard-assignment* functions, mapping every transition $t \in T$ into a read and write guard from $\mathcal{G}_\mathcal{E}$.

We respectively call *Var*$_r(t)$ and *Var*$_w(t)$ the sets of t's *read* and *write* variables, as a shortcut notation for *Var*($r(t)$ and *Var*($w(t)$). Given a place or a transition $v \in P \cup T$ of D, the *preset* and the *postset* of v are, respectively, the sets ${}^\bullet v = \{w \mid F(w, v) > 0\}$ and $v^\bullet := \{w \mid F(v, w) > 0\}$.

Example 2: Figure 1 shows two DPNs encoding the two clinical guideline fragments discussed in Section II. The two figures employ string constants, which can be easily encoded into dedicated real numbers to fit our formal definition.

We turn to the DPN execution semantics. A *state* of a DPN $D = (P, T, F, l, V, r, w)$ over $\mathcal{E}$ is a pair (M, α), where:

- $M : P \rightarrow \mathbb{N}$ is a total *marking* function, assigning a number $M(p)$ of *tokens* to every place $p \in P$;
- $\alpha : V \rightarrow \mathbb{R}$ is a total *variable valuation* (function) assigning a real value to every variable in V.

Every state, together with a (variable) valuation β inducing an update over (some of) the net variables, yields a set of

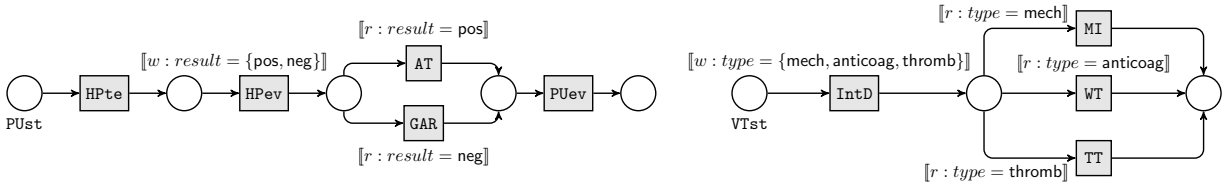


Fig. 1. DPN representations for the peptic ulcer (left) and venous thromboembolism (right) clinical guideline fragments. We use prefixes *r:* and *w:* to distinguish read and write guards respectively. Trivial, true guards are omitted for brevity.

enabled transitions, which can be *fired* progress the net. This requires to augment the usual notions of enablement/firing by considering also the read and write guards.

Definition 8: Consider a DPN $D = (P, T, F, l, V, r, w)$. Transition $t \in T$ is *enabled* in state (M, α) under partial valuation $\beta : V \to \mathbb{R}$, denoted $(M, \alpha)[t, \beta\rangle$, iff:

- β is defined on all variables $v \in \mathit{Var}_r(t) \cup \mathit{Var}_w(t)$;
- for every $v \in \mathit{Var}_r(t)$, we have that $\beta(v) = \alpha(v)$, i.e., β matches α on t's read variables and;
- $\beta \models r(t)$ and $\beta \models w(t)$, i.e., β satisfies the read and write guards of t; and
- for every $p \in {}^\bullet t$, it is the case that $M(p) \geq F(p, t)$.

Given a transition t enabled in state (M, α) under β, a state (M', α') is the result of *firing* t *in* (M, α), written $(M, \alpha)[t, \beta\rangle(M', \alpha')$, iff:

- for every $p \in P$, we have $M'(p) = M(p) - F(p, t) + F(t, p)$; and
- for every $v \in \mathit{Var}_w(t)$, we have $\alpha'(v) = \beta(v)$.
- for every $v \in V \setminus \mathit{Var}_w(t)$, we have $\alpha'(v) = \alpha(v)$.

We refer to the expression $(M, \alpha)[t, \beta\rangle(M', \alpha')$ as *transition firing*. State (M', α') is *reachable* from (M, α), if there exists a sequence of transition firings from (M, α) to (M', α').

In this paper, we deal only with DPNs that are *safe* (i.e., 1-bounded) and *well-formed (over their respective set of event signatures $\mathcal{E}$)*. The former means that for every state (M', α') reachable from a state (M, α), if $M(p) \leq 1$ then $M'(p) \leq 1$. This is done for convenience (our approach seamlessly works for k-bounded nets). The latter means that transitions and event signatures are compatible, in the following sense: *(i)* for every (visible) transition $t \in T$ with $l(t) = n$ for some event signature $\langle n, A\rangle \in \mathcal{E}$, we have that the write guard uses, as variables, precisely those matching with attributes in A, that is, $\mathit{Var}_w(t) = A$; *(ii)* for every (silent) transition $t \in T$ with $l(t) = \tau$, net variables are left untouched, that is, $w(t) \equiv \top$. The first requirement captures the intuition that the payload of an event is used to update the net variables, provided that the corresponding write guard is satisfied. The second requirement indicates that variables are only manipulated when a visible transition, triggered by an event, fires.

To define runs, we fix a *DPN with initial state and final marking* (DPNIF) as a pair $\bar{D} = (D, (M_0, \alpha_0), M_f)$, where D is a DPN, (M_0, α_0) a state of D (called *initial state*), and M_f a marking of D (called *final marking*). A *run* of $\bar{D}$ is a sequence of transition firings of D that starts from (M_0, α_0) and finally leads to a state (M, α) with $M = M_f$.

We are now ready to define when does a trace (in the sense of Def. 2) *comply* with a DPNIF. This captures that the events contained in the trace can be turned into a corresponding run, possibly inserting τ-transitions, while keeping the relative order of events and their correspondence to elements in the run. To do so, we need a preliminary notion. Given two sequences σ_1 and σ_2 such that $|\sigma_2| \geq |\sigma_1|$, an order-preserving injection ι from σ_1 to σ_2 is a total injective function from the elements of σ_1 to those of σ_2, such that for every two elements e_1, e_2 in σ_1 where e_2 comes later than e_1 in the σ_1, we have that $\iota(e_2)$ comes later than $\iota(e_1)$ in σ_2. This notion allow us to easily map traces into (possibly longer) runs of a DPNIF.

Definition 9: A trace $\sigma = e_1 \cdots e_n$, *complies* with a DPNIF $\bar{D}$ with labeling function l if there exist a run ρ of $\bar{D}$ and an order-preserving injection ι from σ to ρ such that:

- every $e = \langle n, \nu\rangle$ in σ is mapped by ι onto a corresponding transition firing in ρ, that is, given $\iota(e) = [t, \beta\rangle$, we have that $l(t) = n$ and β corresponds to ν for the written variables $\mathit{Var}_w(t)$;[1]
- every element $[t, \beta\rangle$ in ρ that does not correspond to any element from σ via ι is so that $l(t) = \tau$.

IV. Monitoring Approach

In this section we provide our main technical contribution: the construction of monitors for hybrid processes. In our context, a hybrid process $\mathcal{H}$ over a set $\mathcal{E}$ of event signatures is simply a set of process components, where each process component is either a LMP-DECLARE constraint over $\mathcal{E}$, or a DPNIF over $\mathcal{E}$. Monitoring a trace against $\mathcal{H}$ basically amounts to running this trace concurrently over all the DPNIFs of $\mathcal{H}$, simultaneously checking whether all constraints in $\mathcal{H}$ are satisfied. When the trace is completed, it is additionally checked that the trace is indeed accepted by the DPNIFs. One important clarification is needed when characterizing the concurrent execution over multiple DPNIFs. In fact, such components may come from different sources, not necessarily employing all the event signatures from $\mathcal{E}$. In this light, it would be counterintuitive to set that a DPNIF rejects an event because its signature is not at all used therein. We fix this by assuming that whenever such a situation happens, the DPNIF simply ignores the currently processed event.

Given this basis, the construction of monitors for such hybrid processes goes through multiple conceptual and algorithmic steps, detailed next.

[1] Recall that β involves both read and written variables. The read variables are used to guarantee that the fired transition is enabled, and it is on the written variables that ν and β must agree.

A. Interval Abstraction

The first challenge that one has to overcome is related to reasoning with data conditions, that is, checking whether a condition is satisfied by an assignment, and checking whether a condition is satisfiable (both operations will be instrumental when constructing automata). The main issue is that, due to the presence of data, there are infinitely many distinct assignments from variables/attributes to values, in turn inducing infinitely many states to consider in the DPNs (even when the net is bounded). To tame this infinity, we build on the faithful abstraction techniques studied in [4], recasting them in our more complex setting. The idea is to avoid referring to single real values, and instead predicate over intervals, in turn showing that we only have a fixed number of intervals to consider, which in turn leads us to propositional reasoning. This is obtained by observing that data conditions can distinguish between only those constants that are explicitly mentioned therein; hence, we simply fetch constants used in the process components (i.e., some atomic condition, guard or initial DPN assignment) to delimit the intervals to consider.

Technically, let $\mathcal{C} = \{c_1, \ldots, c_m\}$ be a finite set of values from $\mathbb{R}$ assuming, without loss of generality, that $c_i < c_{i+1}$, for $i \in \{1, \ldots, m-1\}$. We then partition $\mathbb{R}$ into $\mathcal{P}_\mathcal{C} = \{(-\infty, c_1), (c_m, \infty)\} \cup \{(c_i, c_i) \mid i = 1, \ldots, m\} \cup \{(c_i, c_{i+1}) \mid i \in \{1, \ldots, m-1\}\}$. Notice that $\mathcal{P}_\mathcal{C}$ is finite, with a size that is linear in m. This is crucial for our techniques: we can see $\mathcal{P}_C$ as a set of intervals over the reals or simply as a fixed set of propositions, depending on our needs. Each interval in the partition is an *equivalence region* for the satisfaction of the atomic conditions $a \odot c$ in $\mathcal{L}_\mathcal{E}$, in the following sense: given two valuations α and α' defined over a, such that $\alpha(a)$ and $\alpha'(a)$ are from the same region $R \in \mathcal{P}_\mathcal{C}$, then $\alpha \models v \odot c$ if and only if $\alpha' \models v \odot c$.

We exploit this as follows. We fix a finite set V of variables (referring to attributes) and lift an assignment $\alpha : V \to \mathbb{R}$ into a corresponding region assignment $\tilde{\alpha} : V \to \mathcal{P}_\mathcal{C}$ so that, for every $a \in V$, $\tilde{\alpha}(a)$ returns the unique interval to which $\alpha(a)$ belongs. Given the observation above, we can then use $\tilde{\alpha}$ to check whether a condition holds over α or not as follows: $\alpha(a)$ satisfies condition $a > c$ with $c \in \mathcal{C}$ if and only if $\tilde{\alpha}(a)$ returns a region (c_1, c_2) with $c_1 > c$ (the same reasoning is similarly done for other comparison operators). This carries over more complex conditions used in LMP-DECLARE and DPNs, as they simply consist of boolean combinations of atomic conditions. The key observation here is that doing this check amounts to propositional reasoning, and so does checking satisfiability of conditions: in fact, since both V and $\mathcal{P}_\mathcal{C}$ are finite, there are only finitely many region assignments that can be defined from V to $\mathcal{P}_\mathcal{C}$.

Given the process components of interest, we fix V to the set $\mathcal{A}_\mathcal{E}$ of all the attributes in the event signature $\mathcal{E}$ of the system under study (this contains all variables used in its process components), and $\mathcal{C}$ to the set of all constants used in the initial states of the DPNs, or mentioned in some condition of a process component. We then consistently apply the lifting strategy from assignments to region assignments, when it comes to traces and DPN states. In the remainder, we assume that V and $\mathcal{C}$ are fixed as described above.

B. Encoding into Guarded Finite-state Automata

As a unifying device to capture the execution semantics of process components, we introduce a symbolic automaton whose transitions are decorated with data conditions.

Definition 10: A *guarded finite-state automaton* (GFA) over set $\mathcal{E}$ of event signatures is a tuple $\mathcal{A} = \langle Q, q_0, \to, F \rangle$, where: *(i)* Q is a finite set of *states*; *(ii)* $q_0 \in Q$ is the *initial state*; *(iii)* $\longrightarrow \subseteq Q \times \mathcal{L}_\mathcal{E} \times Q$ is the labeled *transition function*; and *(iv)* $F \subseteq Q$ is the set of *final states*. For notational convenience, we write $q \xrightarrow{\varphi} q'$ for $\langle q, \varphi, q' \rangle \in \longrightarrow$, and call φ *(transition) guard*.

GFA-runs of $\mathcal{A}$ consist of finite sequences of the form $q_0 \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_n} q_n$, where $q_n \in F$. The set of runs accepted by $\mathcal{A}$ is denoted as $\mathcal{L}_\mathcal{A}$. A trace $\sigma = e_1 \cdots e_m$ over $\mathcal{E}$ is accepted by $\mathcal{A}$ if there exists a GFA-run $q_0 \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_m} q_m$ such that for $i \in \{1, \ldots, m\}$, we have $e_i \models \varphi_i$. In general, an event e can satisfy the guards of many transitions outgoing from a state q, as guards are not required to be mutually exclusive. Thus, a trace may correspond to many GFA-runs. In this sense, GFAs are, in general, nondeterministic.

It is key to observe that GFAs can behave like standard finite-state automata. In fact, by setting $\mathcal{C}$ to a finite set of constants including all those mentioned in the automata guards, we can apply the interval abstraction from Section IV-A to handle automata operations. In particular, in place of considering the infinitely many events over $\mathcal{E}$, we can work over the finitely many abstract events defined using region assignments over $\mathcal{P}_\mathcal{C}$. For example, we can check whether a trace $\sigma = \langle n_1, \nu_1 \rangle \cdots \langle n_m, \nu_m \rangle$ is accepted by $\mathcal{A}$ by checking whether the abstract trace $\langle n_1, \tilde{\nu}_1 \rangle \cdots \langle n_m, \tilde{\nu}_m \rangle$ does so. Notice that, to construct this abstract trace, it suffices to represent each event $\langle n, \nu \rangle$ in σ using equivalence regions from $\mathcal{P}_\mathcal{C}$, such that every $\nu(a) = c$ is substituted either with region $[c, c]$, if $[c, c] \in \mathcal{P}_\mathcal{C}$, or with region (c', c'') s.t. $c \in (c', c'')$. For ease of reference, we shall use $\sigma^{\mathcal{P}_\mathcal{C}}$ to represent the abstract trace.

Thanks to this, we can construct GFAs using standard automata techniques (e.g., for LTL$_f$, as discussed below), and also directly apply standard algorithms, coupled with our interval abstraction, to minimize and determinize GFAs.

From LMP-Declare constraints to GFAs. The translation of an LTL$_f$ formula into a corresponding finite-state automaton [7], [6] has been largely employed in the literature to build execution engines and monitors for Declare. In the case of Declare, atomic formulae are simply names of activities, and consequently automata come with transitions labeled by propositions that refer to such names. In the case of LMP-DECLARE, atomic formulae are more complex conditions from $\mathcal{L}_\mathcal{E}$. Thanks to interval abstraction, this is however not an issue: we simply apply the standard finite-state automata construction for a Declare constraint [6], with

the only difference that transitions are labeled by those conditions from $\mathcal{L}_\mathcal{E}$ mentioned within the constraint. We only keep those transitions whose label is a satisfiable condition, which as discussed in Section IV-A can be checked with propositional reasoning. A final, important observation is that the so-constructed GFA is kept *complete* (that is, *untrimmed*), so that each of its states can process every event from $\mathcal{E}$.

From DPNIFs to GFAs. We show that a DPNIF $\bar{D} = (D, (M_0, \alpha_0), M_f)$ can be encoded into a corresponding GFA that accepts all and only those traces that comply with $\bar{D}$. For space reasons, we concentrate here on the most important aspects.

The main issue is that the set S of DPN states that are reachable from the initial marking (M_0, α_0) is in general infinite even when the net is bounded (i.e., has boundedly many markings). This is due to the existence of infinitely many valuations for the net variables. To tame this infinity, we consider again the partition $\mathcal{P}_\mathcal{C}$ defined above; this induces a partition of S into equivalence classes, according to the intervals assigned to the variables of D. Technically, given two assignments α, α' we say that α is equivalent to α', written $\alpha \sim \alpha'$, iff for every $v \in V$ there exists a region $R \in \mathcal{P}_\mathcal{C}$ s.t. $\alpha(v), \alpha'(v) \in R$. Then, two states $(M, \alpha), (M', \alpha') \in S$ are said to be equivalent, written $(M, \alpha) \sim (M', \alpha')$ iff $M = M'$ and $\alpha \sim \alpha'$. Observe that, by what discussed above, the assignments of two equivalent states satisfy exactly the same net guards. By $[S]_\sim$, we denote the quotient set of S induced by the equivalence relation $\sim$ over states defined above.

Based on Section IV-A we directly get that $[S]_\sim$ is finite. We can then conveniently represent each equivalence class of $[S]_\sim$ by $(M, \tilde{\alpha})$, explicitly using the region assignment in place of the infinitely many corresponding value-based ones. This provides the basis for the following encoding.

Definition 11 (GFA induced by a DPNIF): Given a DPNIF $\bar{D} = (D, (M_0, \alpha_0), M_f)$ with $D = (P, T, F, l, r, w)$, the *GFA induced by* $\bar{D}$ is $\mathcal{A}_D = \langle Q, q_0, \rightarrow, F \rangle$, where:

1) $Q = [S]_\sim$;
2) $q_0 = (M_0, \tilde{\alpha}_0)$, where $\tilde{\alpha}_0(v) = [\alpha_0(v), \alpha_0(v)]$, for all $v \in V$;
3) $\longrightarrow \subseteq Q \times \mathcal{L}_\mathcal{E} \times Q$ is s.t. $(M, \tilde{\alpha}) \xrightarrow{a \wedge \psi} (M', \tilde{\alpha}')$ iff there exists a transition $t \in T$ and a partial valuation β s.t. $(M, \alpha)[t, \beta\rangle(M', \alpha')$, with:
   a) $\alpha(v) \in \tilde{\alpha}(v)$ and $\alpha'(v) \in \tilde{\alpha}'(v)$, for every $v \in V$;
   b) $a = l(t)$;
   c) $\psi = \bigwedge_{v \in Var(w(t))} \phi_v$, where: *(i)* $\phi_v \equiv (v > c_i \wedge v < c_{i+1})$, if $\tilde{\alpha}'(v) = (c_i, c_{i+1})$; *(ii)* $\phi_v \equiv (v = c_i)$, if $\tilde{\alpha}'(v) = (c_i, c_i)$;
4) $F \subseteq Q$ is s.t. $(M, \tilde{\alpha}) \in F$ iff $M = M_f$.

Next we introduce an algorithm that, given a DPNIF $\bar{D} = (D, (M_0, \alpha_0), M_f)$, constructs the GFA $\mathcal{A}_D$ corresponding to it (that is, it represents all possible behaviors of $\bar{D}$). In the algorithm, we make use of the following functions:

- $enabled(M, \tilde{\alpha})$ returns a set of transitions and region assignments $\{(t, \tilde{\beta}) \mid t \in T$ and $(M, \alpha)[t, \beta\rangle$, where $\beta(v) \in \tilde{\beta}(v)$, $\alpha(v) \in \tilde{\alpha}(v)$, for $v \in V\}$. Notice that $\tilde{\beta}$ matches only the "allowed" regions. That is, for every t, we need to construct multiple $\tilde{\beta}$ that account for *all possible combinations* of equivalence regions assigned to each variable in $w(t)$ and $r(t)$ such that $\tilde{\beta} \models w(t)$ and $\tilde{\beta} \models r(t)$.
- $guard(t, \tilde{\alpha})$ returns a formula ψ as in Definition 3c.
- $fire(M, t, \tilde{\beta})$ returns a pair $(M, \tilde{\alpha})$ as in Definition 3.

It is easy to see that all the aforementioned functions are computable. For $enabled$ there are always going to be finitely many combinations of regions from $\mathcal{P}_\mathbb{R}$ satisfying the guards of t, whereas formulas produced by $guard$ can be constructed using a version of the respective procedure from Definition 11 that uses $\tilde{\beta}$ instead of $\tilde{\alpha}'$, and next states returned by $fire$ can be generated via the usual new state generation procedure from Definition 8, proviso that it has to be invoked in the context of equivalence regions.

The actual algorithm is similar to the classical one used for the reachability graph construction of a Petri net (see, e.g., [16]). It is important to notice that, in the proposed algorithm, silent transitions are treated as regular ϵ-transitions (that is, we assume that $\mathcal{L}_\mathcal{E}$ as well as $\longrightarrow$ of the output automaton are suitably extended with τ). We discuss later on how such transitions can be eliminated from resulting GFAs.

Algorithm 1 Compute GFA from DPN

Input: DPNIF $\bar{D} = (D, (M_0, \alpha_0), M_f)$ with $D = (P, T, F, l, r, w)$
Output: GFA $\langle Q, q_0, \rightarrow, F \rangle$

```
Q := (M_0, α̃_0), where α̃_0(v) = [α_0(v), α_0(v)], for v ∈ V
W := {(M_0, α̃_0)}
⟶ := ∅
while W ≠ ∅ do
  select (M, α̃) from W
  W := W \ (M, α̃)
  for all (t, β̃) ∈ enabled(M, α̃) do
    (M', α̃') := fire(M, t, β̃)
    if (M', α̃') ∉ Q then
      Q := Q ∪ {(M', α̃')}
      W := W ∪ {(M', α̃')}
    end if
    ψ := l(t)
    if w(t) ≢ ⊤ then
      ψ := ψ ∧ guard(t, α̃')
    end if
    ⟶ := ⟶ ∪ {(M, α̃) --ψ--> (M', α̃')}
  end for
end while
```

The following theorem outlines the main properties of the presented algorithm.

Theorem 1: Algorithm 1 effectively computes a GFA $\mathcal{A}_D$ induced by a DPNIF $\bar{D}$, is sound and terminates.

However, Algorithm 1 is not guaranteed to produce GFAs that are complete. To ensure the completeness of the algorithm output, three additional modifications have to be performed.

(M1) Given that the so-obtained GFA $\mathcal{A}_D$ does not properly handle silent, τ-transitions (they are simply treated as

normal ones by the algorithm), we need to compile them away from $\mathcal{A}_D$, which is done by recasting the standard ϵ-move removal procedure for finite-state automata based on ϵ-closures (see, e.g., [8]) into our setting. This procedure allows to collapse (sequences of) states corresponding to τ-transitions into those that are used for representing only behaviors of all possible traces complying to $\mathcal{E}$. [2] To collapse the aforementioned sequences, each state $q \in Q$ s.t. there are k runs $q \xrightarrow{\tau} \cdots \xrightarrow{\tau} q'_i$, where every transition is labeled only with τ, $q'_i \xrightarrow{\varphi} q''_i$, $\varphi \neq \tau$ and $i = 1, \ldots, k$, has to be replaced with the set of all such q'_i states by creating additional transitions to predecessors of q (if any) as well as q''_i. Notice also that, while the ϵ-transition removal procedure produces deterministic automata, its counterpart working with GFAs may produce an automaton that is non-deterministic.

(M2) The output GFA has to be made "tolerant" to events whose signature is not at all used in the DPNIF, formalising the intuition described at the beginning of Section IV. This is done by introducing additional loops in $\rightarrow_i$ from Definition 11 as follows: for every $q \in Q_i$ we insert a looping transition $q \xrightarrow{\psi} q$, where $\psi = \bigwedge_{a \in (\mathcal{N}_\mathcal{E} \setminus \bigcup_{t \in T} l(t))} a$. By doing so, we allow the GFA to skip irrelevant events that could never be processed by the net.

(M3) The resulting GFA has to be extended with two types of extra transitions.

- The first one tackles invalid (or incorrect) net executions where a partial run cannot be completed into a proper run due to a data-related deadlock. This can occur for two reasons, respectively related to the violation of read versus write guards of all transitions that, from the control-flow perspective, could actually fire. To deal with the read-related issue we proceed as follows: for every state $(M, \tilde{\alpha}) \in Q_i$ and every transition $t \in T_i$, if $\tilde{\alpha} \not\models r(t)$ and $M(p) \geq F_i(p, t)$ for every $p \in P_i$, then add $(M, \tilde{\alpha}) \xrightarrow{\psi} (M, \tilde{\alpha})$ to $\rightarrow_i$, where ψ is as in Definition 11 (see 3). This is only done when ψ is actually satisfiable.
- The second one addresses write-related issues arising when the event to be processed carries values that violate all the write guards of candidate transitions. We handle this as follows: for every $(M, \tilde{\alpha}) \in Q_i$ and every $t \in T_i$ s.t. $w(t) \not\equiv \top$, add $(M, \tilde{\alpha}) \xrightarrow{a \wedge \psi} (M, \tilde{\alpha})$ to $\rightarrow_i$, where, for $t \in T_i$ and every $p \in {}^\bullet t$ s.t. $M(p) \geq F(p, t)$, $a = l(t)$ and ψ is constructed as in Definition 11 (bullet 3, with the only difference that all φ_v are computed for a combination of equivalence regions from $\mathcal{P}_\mathcal{C}$, each of which is composed into (partial) variable region valuation $\tilde{\beta} : V \rightarrow \mathcal{P}_\mathcal{C}$ s.t. $\tilde{\beta} \not\models v \odot c$, for every atomic condition $v \odot c$ in $w(t)$. This is only done if ψ is actually satisfiable. This step can be also optimised by putting all such ψ in one DNF formula, which in turn reduces the number of transitions in the GFA.

Proposition 1: Let $\mathcal{A}_D$ be a GFA induced by a DPNIF $\bar{D}$. Application of modifications **(M1)**, **(M2)** and **(M3)** to $\mathcal{A}_D$ produces a new GFA $\mathcal{A}'_D$ that is complete.

Now, whenever we get a complete GFA for a DPNIF, we can use the former to check whether a log trace is compliant with the net (as by Definition 9).

Theorem 2: A trace $\sigma = e_1 \cdots e_n$ is compliant with a DPNIF $\bar{D} = (D, (M_0, \alpha_0), M_f)$ iff abstract trace $\sigma^{\mathcal{P}_\mathcal{C}}$ is accepted by the GFA $\mathcal{A}_D$ induced by $\bar{D}$.

C. Combining GFAs

Given a hybrid process $\mathcal{H} = \{h_1, \ldots, h_n\}$ with n components, we now know how to compute a GFA for each of its components. Let $\mathcal{A}_i$ be the GFA obtained as described in Section IV-B, depending on whether h_i is a LMP-DECLARE constraint or DPNIF, and in addition minimized and determinized. This means that, being $\mathcal{A}_i$ complete, it will have a single trap state capturing all those traces that permanently violate the process component.

To perform monitoring, we follow the approach of colored automata [11], [6] and label each automaton state with one among fourth truth values, respectively indicating whether, in such state, the corresponding process component is *temporarily satisfied* (TS), *temporarily violated* (TV), *permanently satisfied* (PS), or *permanently violated* (PV). As for constraints, these are interpreted exactly like in [11], [6]. As for DPNIF, TS means that the current trace is accepted by the DPNIF, but can be extended into a trace that is not, while TV means that the current trace is a good prefix of a trace that will be accepted by the DPNIF (PV and PS are defined dually).

Considering the way our GFAs are obtained, this labeling is done as follows: *(i)* $v_i(q_i) = PS$ iff $q_i \in F_i$ and all transitions outgoing from q are self-loops; *(ii)* $v_i(q_i) = TS$ iff $q_i \in F_i$ and there is some transition outgoing from q that is not a self-loop; *(iii)* $v_i(q_i) = PV$ iff $q_i \notin F_i$ and all transitions outgoing from q are self-loops; *(iv)* $v_i(q_i) = TV$ iff $q_i \notin F_i$ and there is some transition outgoing from q_i that is not a self-loop.

The so-obtained labeled GFAs are local monitors for the single process components of $\mathcal{H}$. To monitor $\mathcal{H}$ as a whole and do early detection of violations arising from conflicting components, we need to complement such automata with a global GFA $\mathcal{A}$, capturing the interplay of components. We do so by defining $\mathcal{A}$ as a suitable *product automaton*, obtained as a cross-product of the local GFAs, suitably annotated to retain some relevant information.

Technically, $\mathcal{A} = \langle Q, q_0, \rightarrow, F \rangle$, where: *(i)* $Q = Q_1 \times \cdots \times Q_n$; *(ii)* $q_0 = \langle q_{10}, \ldots, q_{n0} \rangle$; *(iii)* $\longrightarrow$ is s.t. $\langle q_1, \ldots, q_n \rangle \xrightarrow{\varphi} \langle q'_1, \ldots, q'_n \rangle$ iff $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_n$, with $q_i \xrightarrow{\varphi_i}_i q'_i$ and φ is satisfiable by exactly one event; *(iv)* $F = F_1 \times \cdots \times F_n$. Observe that the definition requires checking whether guards of $\mathcal{A}$ are satisfiable by some event (otherwise the labeled transition could not be triggered by any event and should be omitted).

[2] Since we assume to be working with well-formed nets only, τ-transitions do not come with write guards, and therefore do not alter the net variables.

The v_i labeling functions from above induce a labeling on the states $q = \langle q_1, \ldots, q_n \rangle \in Q$ of the product automaton, which tells us whether all constraints and nets nets from Φ are overall temporarily/permanently violated/satisfied. Specifically, we define the labeling $v : Q \mapsto \{TS, TV, PS, PV\}$ s.t.: *(i)* $v(q) = PV$ iff $v_i(q_i) = PV$, for some $i \in 1, \ldots, n$; *(ii)* $v(q) = PS$ iff $v_i(q_i) = PS$, for all $i \in 1, \ldots, n$; *(iii)* $v(q) = TS$ iff $v_i(q_i) = TS$, for all $i \in 1, \ldots, n$; *(iv)* $v(q) = TV$, otherwise.

D. Best Event Identification

It is crucial to notice that, differently from local GFAs, the global GFA $\mathcal{A}$ is *not* minimized. This allows the monitor to distinguish among states of permanent violations arising from different combinations of permanently violated components, in turn allowing for fine-grained feedback on what are the "best" events that could be processed next. To substantiate this, we pair a hybrid process $\mathcal{H}$ with a *violation cost function* that, for each of its components, returns a natural number indicating the cost incurred for violating that component.

$\mathcal{A}$ is augmented as follows. Each state $q \in Q$ of $\mathcal{A}$ is associated with two cost indicators:

- a value $cost_{cur}(q)$ containing the sum of the costs associated with the constraints violated in q;
- a value $cost_{best}(q)$ containing the best value $cost_{cur}(q')$, for some state q' (possibly q itself) reachable from q.

The functions $cost_{cur}$ and $cost_{best}$ can be easily computed through the fixpoint computation described below, where c_i is the cost associated with the violation of constraint φ_i.

1) For every state $q = \langle q_1, \ldots, q_n \rangle \in Q$, let
$$cost^0_{best}(q) = cost_{cur}(q) = \sum_{i=1,\ldots,n} cost(q_i),$$
where $cost(q_i) = 0$ if $q_i \in F_i$, c_i otherwise.
2) Repeat the following until $cost^{i+1}_{best}(q) = cost^i_{best}(q)$, for all $q \in Q$: for every state $q \in Q$,
$cost^{i+1}_{best}(q) := \min\{cost^i_{best}(q') \mid q \to q'\} \cup \{cost_{cur}(q)\}$;
3) return $cost_{best}$.

It is immediate to see that a fixpoint is eventually reached in finite time, thus the algorithm terminates. To this end, observe that, for all $q \in Q$, $cost_{best}(q)$ is integer and non-negative. Moreover, at each iteration, if $cost_{best}(q)$ changes, it can only decrease. Thus, after a finite number of steps, the minimum $cost_{best}(q)$ for each state $q \in Q$ must necessarily be achieved, which corresponds to the termination condition.

We can also see that the algorithm is correct, i.e., that if $cost_{best}(q) = v$ then, from q: *(i)* there exists a path to some state q' s.t. $cost_{cur}(q') = v$, and *(ii)* there exists no path to some state q'' s.t. $cost_{cur}(q'') < v$. These come as a consequence of step 2 of the algorithm. By this, indeed, we have that, after the i-th iteration $cost_{best}(q)$ contains the value of the state with the minimum cost achievable from q through a path containing at most i transitions. When the fixpoint is reached, that means that even considering longer paths will not improve the value, that is, it is minimal.

Using $\mathcal{A}$, $cost_{cur}$ and $cost_{best}$, we can find the next "best events", i.e., those events that allow for satisfying the combination of constraints that guarantees the minimum cost. Technically, let $\sigma = e_1 \cdots e_\ell \in {\mathcal{V}_\mathcal{E}}^*$ be the input trace and consider the set $\Gamma = \{\rho_1, \ldots, \rho_n\}$ of the runs of $\mathcal{A}$ on σ. Let $q_{i\ell}$ be the last state of each ρ_i and let
$$\hat{q} = \operatorname*{argmin}_{q \in \{q_{1\ell}, \ldots, q_{n\ell}\}} \{cost_{best}(q)\}.$$
If, for some i, $\hat{q} = q_{i\ell}$, then $\hat{q}$ is the best achievable state and no event can further improve the cost. Otherwise, take a successor q' of $\hat{q}$ s.t. $cost_{best}(q') = cost_{best}(\hat{q})$. Notice that by the definition of $cost_{best}$, one such q' necessarily exists, otherwise $\hat{q}$ would have a different cost $cost_{best}(\hat{q})$.

The best events are then the events e^* s.t.:
$$e^* \models \varphi, \text{ for } \varphi \text{ s.t. } q \xrightarrow{\varphi} q'$$

Notice that, to process the trace in the global GFA $\mathcal{A}$, and to detect the best events, we again move back and forth from traces/events and their abstract representation based on intervals, as discussed in Section IV-A. In particular, notice that there may be infinitely many different best events, obtained by different groundings of the attributes within the same intervals.

V. Evaluation

In this section, we present an evaluation of the proposed monitoring approach using a prototype implementation (available at https://github.com/antialman/model-interplay-monitoring-code). We reuse the process specifications presented in Section II with the following violation costs: PU - 10; VT - 5; C - 15; Meta-constraint - 3.

A. Violation of VT procedure

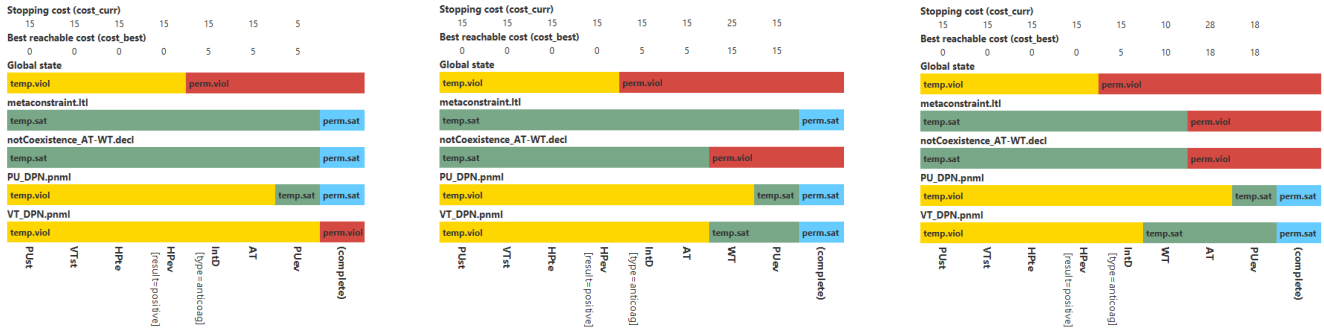
In the first scenario (shown on Fig. 2(a)), after the events $PUst$, $VTst$, $HPte$ and $HPev_{[result=positive]}$, the cost of stopping is 15 and best reachable cost of 0, meaning that stopping the process execution would result in violations that can be avoided at this point in time.

The next event $IntD_{[type=anticoag]}$, which, while not conflicting with any process specifications directly, causes a permanent violation in the global state. This is a result of the PU procedure requiring the occurrence of AT, the VT procedure requiring the occurrence WT, and constraint C requiring that AT and WT can not occur in the same trace. Given the cost model specified earlier, the set of next recommended events (not shown on the figure) is MI, TT and AT, the first two of which would be a violation of the VT procedure, which is the least costly option at this point.

The final events in the trace are AT and $PUev$, which complete the PU procedure. VT procedure is however violated, resulting in a total violation cost of 5, which was also the best reachable cost after the event $IntD_{[type=anticoag]}$.

B. Violation of constraint C

The results of the second scenario (shown on Fig. 2(b)) are the same as in the first scenario (Section V-A) up until the event WT. The event WT completes the VT procedure, but also causes a permanent violation of constraint C. As a result,



(a) Scenario with VT procedure violation. (b) Scenario with constraint C violation. (c) Scenario with meta-constraint violation.

Fig. 2. Monitoring results for the three scenarios of the evaluation

the best reachable cost after the event WT is 15. Stopping cost at that point is 25, because the PU procedure is not yet completed. Notice also that the meta-constraint was not violated in this trace because AT occurred before WT.

The final event of the trace $PUev$ completes the PU procedure and as a result the stopping cost becomes equal to the best reachable cost of 15.

C. Violation of the meta-constraint

The trace in the third scenario is almost the same as in the second scenario (Section V-B) with the only difference being the reverse order of the events AT and WT. This results in an additional violation of the meta-constraint after the event AT which in turn adds the meta-constraint violation cost of 3 to both the stopping cost (28 compared to 25 in the second scenario) and the best reachable cost (18 compared to 15 in the second scenario) after the event AT.

VI. Conclusion

The ability to monitor the interplay of different process models is useful in domains where process instances tend to have high variability. An example of this is the medical domain where standard treatment procedures are described as clinical guidelines and multiple guidelines need to be often executed simultaneously, therefore giving rise to interplay and possible conflicts. Furthermore, because a clinical guideline cannot account for all possible preconditions that a patient may have, it is also necessary to employ declarative knowledge (allergies, prior conditions etc.) which further complicates the process execution.

This paper proposes a monitoring approach that can take into consideration the interplay of multiple process specifications (both procedural and declarative). Additionally, the approach includes a recommendation component which helps either to avoid violations or (if avoiding the violations is not possible) to minimize the total cost of the violations. The proposed approach is limited in that it can only provide recommendations on the immediate next events, however in the future we plan to extend the approach to provide recommendations as full continuations of the trace and to explore different possible execution semantics for concurrent execution of multiple process models.

Acknowledgements. The work of A. Alman was supported by the Estonian Research Council (project PRG1226) and ERDF via the IT Academy Program.

References

[1] A. A. Andaloussi, A. Burattin, T. Slaats, E. Kindler, and B. Weber. On the declarative paradigm in hybrid business process representations: A conceptual framework and a systematic literature study. *Inf. Syst.*, 91:101505, 2020.

[2] A. Bottrighi, F. Chesani, P. Mello, M. Montali, S. Montani, and P. Terenziani. Conformance checking of executed clinical guidelines in presence of basic medical knowledge. In *Proc. of BPM Workshops*, volume 100 of *LNBIP*, pages 200–211. Springer, 2011.

[3] A. Burattin, F. M. Maggi, and A. Sperduti. Conformance checking based on multi-perspective declarative process models. *Expert Syst. Appl.*, 65:194–211, 2016.

[4] M. de Leoni, P. Felli, and M. Montali. A holistic approach for soundness verification of decision-aware process models. In *ER*, volume 11157 of *LNCS*, pages 219–235. Springer, 2018.

[5] M. de Leoni, P. Felli, and M. Montali. Strategy synthesis for data-aware dynamic systems with multiple actors. In *KR*, pages 315–325, 2020.

[6] G. D. Giacomo, R. D. Masellis, M. Grasso, F. M. Maggi, and M. Montali. Monitoring business metaconstraints based on LTL and LDL for finite traces. In *BPM*, volume 8659 of *LNCS*, pages 1–17. Springer, 2014.

[7] G. D. Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI*, pages 854–860. IJCAI/AAAI, 2013.

[8] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007.

[9] L. T. Ly, F. M. Maggi, M. Montali, S. Rinderle-Ma, and W. M. P. van der Aalst. Compliance monitoring in business processes: Functionalities, application, and tool-support. *Inf. Syst.*, 54:209–234, 2015.

[10] F. M. Maggi, M. Montali, and U. Bhat. Compliance monitoring of multi-perspective declarative process models. In *EDOC*, pages 151–160. IEEE, 2019.

[11] F. M. Maggi, M. Montali, M. Westergaard, and W. M. P. van der Aalst. Monitoring business constraints with linear temporal logic: An approach based on colored automata. In *BPM*, volume 6896 of *LNCS*, pages 132–147. Springer, 2011.

[12] F. M. Maggi, M. Westergaard, M. Montali, and W. M. P. van der Aalst. Runtime verification of ltl-based declarative process models. In *RV*, volume 7186 of *LNCS*, pages 131–146. Springer, 2011.

[13] F. Mannhardt, M. de Leoni, H. A. Reijers, and W. M. P. van der Aalst. Balanced multi-perspective checking of process conformance. *Computing*, 98(4):407–437, 2016.

[14] R. D. Masellis, F. M. Maggi, and M. Montali. Monitoring data-aware business constraints with finite state automata. In *ICSSP*, pages 134–143. ACM, 2014.

[15] M. Montali, M. Pesic, W. M. P. van der Aalst, F. Chesani, P. Mello, and S. Storari. Declarative specification and verification of service choreographiess. *ACM Trans. Web*, 4(1):3:1–3:62, 2010.

[16] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[17] M. Pesic, H. Schonenberg, and W. M. P. van der Aalst. DECLARE: full support for loosely-structured processes. In *EDOC*, pages 287–300. IEEE Computer Society, 2007.

[18] L. Piovesan, P. Terenziani, and D. T. Dupré. Temporal conformance analysis and explanation on comorbid patients. In *HEALTHINF*, pages 17–26. SciTePress, 2018.

[19] P. Terenziani, G. Molino, and M. Torchio. A modular approach for representing and executing clinical guidelines. *Artif. Intell. Medicine*, 23(3):249–276, 2001.