Artificial Intelligence In Medicine 139 (2023) 102512

Contents lists available at ScienceDirect

Artificial Intelligence In Medicine

journal homepage: www.elsevier.com/locate/artmed

Research paper

Monitoring hybrid process specifications with conflict management: An automata-theoretic approach

Anti Alman [a], Fabrizio Maria Maggi [b], Marco Montali [b], Fabio Patrizi [c], Andrey Rivkin [b,d,*]

[a] Institute of Computer Science, University of Tartu, Narva mnt 18, Tartu, 51009, Tartumaa, Estonia
[b] Faculty of Computer Science, Free University of Bozen-Bolzano, Dominikanerplatz 3 - piazza Domenicani, 3, Bolzano, 39100, South Tyrol, Italy
[c] Department of Computer, Control and Management Engineering, Sapienza University of Rome, Via Ariosto 25, Rome, 00185, Lazio, Italy
[d] Department of Applied Mathematics and Computer Science, Technical University of Denmark, Richard Petersens Plads 321, Kgs. Lyngby, 2800, Capital Region, Denmark

ARTICLE INFO

Keywords:
Business process monitoring
Data Petri nets
Declare
Automata
Hybrid process
Process model interplay

ABSTRACT

Complexity of medical treatments can vary from prescribing medicine for a specific ailment to managing a complex set of simultaneous medical issues. In the latter case, doctors are assisted by clinical guidelines which outline standard medical procedures, tests, treatments, etc. To facilitate the use of such guidelines, they can be digitized as processes and adopted in complex process engines offering additional help to health providers such as decision support while monitoring active treatments so as to detect flaws in treatment procedures and suggest possible reactions on them. For example, a patient may present symptoms of multiple diseases simultaneously (requiring multiple clinical guidelines to be followed), while also being allergic to some often-used drugs (requiring additional constraints to be respected). This can easily lead to treating a patient based on a set of process specifications which are not fully compatible with each other. While a scenario like that commonly occurs in practice, research in that direction has thus far given little consideration to how to specify multiple clinical guidelines and how to automatically combine their specifications in the context of the monitoring task. In our previous work (Alman et al., 20222), we presented a conceptual framework for handling the above cases in the context of monitoring. In this paper, we present the algorithms necessary for implementing key components of this conceptual framework. More specifically, we provide formal languages for representing clinical guideline specifications and formalize a solution for monitoring the interplay of such specifications expressed as a combination of (data-aware) Petri nets and temporal logic rules. The proposed solution seamlessly handles combination of the input process specifications and provides both early conflict detection and decision support during process execution. We also discuss a proof-of-concept implementation of our approach and present the results of extensive scalability experiments.

1. Introduction

Evidence-based medicine has led to the definition of a number of clinical practice guidelines, containing systematic recommendations on how to handle patient care. The aim of such guidelines is to improve the quality of care, to mitigate unjustified deviations, and to reduce costs. The further transition from clinical practice to *computer-interpretable guidelines* (CIGs) has paved the way towards decision support systems aiding healthcare and administrative professionals in the modeling, execution, analysis, and continuous improvement of clinical guidelines [47].

Clinical practice guidelines and their computer-interpretable counterparts are defined by assuming an ideal execution context [8]. This essentially means that a guideline has unlimited provision of resources for its execution, patients suffer only from conditions targeted by the guideline, and that healthcare providers possess complete medical knowledge of the above medical conditions as well as of the patient to which the guideline is applied.

While these assumptions are reasonable when CIGs are used for documentation, training, and model-driven analysis, they become too restrictive when employed for execution and decision support. At runtime, healthcare providers are in fact confronted with the full complexity of dealing with single patients, each presenting their own specific medical conditions, personal constraints/preferences and co-morbidities, as well as unanticipated exceptions. On a per-patient basis, this calls for:

* Corresponding author.
E-mail address: ariv@dtu.dk (A. Rivkin).

https://doi.org/10.1016/j.artmed.2023.102512
Received 31 May 2022; Received in revised form 7 January 2023; Accepted 31 January 2023
Available online 5 March 2023
0933-3657/

1. integrating multiple CIGs based on the specific medical condition of the patient, especially in the light of their co-morbidities;
2. considering the interplay between such CIGs and background medical knowledge;
3. continuously adapting and personalizing the resulting process depending on the current circumstances, that is, on the history of executed activities, the effects and data they have produced, and other relevant events.

The net effect of this complexity is that, while treating the patient, healthcare providers may *deviate* from the courses of execution prescribed by the guideline(s). Deviations may arise for a wide range of different reasons, from human mistakes to deliberate choices that depart from the pre-defined execution paths to save patient's life. Dedicated techniques have been consequently developed within medical informatics to compare the execution trace and electronic patient record of a specific patient, with one or more corresponding reference CIGs. As surveyed in [47], these techniques fall under the name of *compliance checking* or *critiquing*, depending on the input knowledge they consider and how they use such knowledge to elaborate on the meaning of and reasons behind the detected deviations. Interestingly, these notions incarnate, in the clinical setting, one of the main general analytic tasks of process mining, namely *conformance checking* [13]. Conformance checking is the task of comparing the expected behavior captured in a reference process model with the event data tracing actual instances of that process, and extract (fine-grained) insights from the detected deviations. *Monitoring* techniques for conformance checking at runtime, tracking ongoing process executions and detecting deviations as soon as possible, have also been extensively investigated [38].

As pointed out in [47], the vast majority of approaches for compliance checking and critiquing present two main shortcomings:

1. compliance is evaluated on a single CIG, and even when multiple CIGs at once are considered, their interplay with background medical knowledge is not tackled at all;
2. compliance is evaluated a-posteriori, and the adopted techniques do not lend themselves to be used at runtime, as they cannot handle the inherent incompleteness of an ongoing, evolving execution.

In this work, we tackle these two open challenges, bringing forward a comprehensive *formal framework for monitoring multiple process specifications and anticipatory detection of deviations*. The framework comes with two distinctive features related to specification and monitoring.

As for specification, we adopt and further develop the M3 framework previously introduced in [2]. The M3 framework brings forward a formal multi-model approach where a process specification emerges from the combination of procedural and declarative process components, which can be respectively used to express CIGs/medical procedures and background medical knowledge. The same process instance concurrently goes through the various components at the same level of abstraction, calling for handling their mutual interplay; this results in so-called loosely coupled hybrid models [3]. Procedural components are represented using a data-aware extension of Petri nets [35,43] where transitions are associated to guards that check and manipulate numerical data variables using variable-to-constant comparisons. Declarative rules are specified using a corresponding data-aware extension of linear temporal logic over finite traces [18] that forms a proper fragment of the logics studied in [12,26,27]. Within this logic, of particular interest are patterns expressed in a fragment of data-aware extensions [11,20,39] of the Declare declarative process specification language [44,50]. The focus on finite traces is motivated by the fact that process instances are expected to terminate, possibly executing unboundedly many steps that eventually lead the instance to one among alternative final states.

This approach allows us to handle the interplay of different components considering both the control-flow and data dimensions, thus supporting interaction schemes like those elicited in [8–10] for the medical domain. An example: if a CIG of interest expects that at least 500mg of paracetamol should be administered to a patient, and background medical knowledge indicates that for that patient 1000mg is the maximum quantity, this results in an admissible interval 500-1000mg. Another example: if a CIG for handling bacterial pneumonia prescribes two alternative treatments, one based on macrolid and another on penicillin, and the patient is found to be allergic to penicillin, then the only treatment that conforms with the CIG and with background medical knowledge is that of macrolid. More in general, we show how simultaneously accounting for models from different modeling paradigms allows us to capture sophisticated forms of scoping and interaction among such models, going beyond what is captured so far in the literature on process mining and providing a formal characterization of forms of hybrid processes mixing CIGs and background medical knowledge, such as those in [4,9,10,51,52].

As for monitoring, we resort to methods from *runtime verification* [36], considering in particular the construction of automata-based monitors for finite traces, extending [16] to account for hybrid models with data variables and variable-to-constant comparisons. The adoption of a formal approach to monitoring is not only useful as it guarantees the construction of correct-by-design monitors, but also because it goes far beyond the mere consideration of execution prefixes and the resulting state of affairs. In particular, we can combine the execution prefix indicating what happened so far within a process instance (e.g., the sequence of activities to which a patient has been subject so far) with speculative reasoning on the possible (infinitely many) future continuations of the instance. This is essential to detect, at the earliest moment possible, violations that cannot be directly ascribed as deviations w.r.t. a single procedural component or declarative rule, but instead emerge as a *conflict* among different components considering their mutual interaction given the current execution state. A state of conflict indicates that, while currently none of the considered components is permanently violated, every continuation will inevitably violate at least one of them.

Consider, for example, a CIG that, given the current trace prefix of a patient treatment process, expects the execution to continue through a long sequence of activities, culminating in one where penicillin is administered to the patient. If the patient is currently found to be allergic to penicillin, the monitor should immediately report the presence of a conflict. This aspect has been already extensively studied in the case of declarative process specifications, [16,26,40,42], but never considering the hybrid, multi-model setting studied in this paper. This form of early detection of violations is of particular interest in the medical domain, as conflicts between different CIGs or between CIGs and background medical knowledge can often occur and should in fact be explicitly reported to healthcare professionals, who in turn have responsibility of handling their resolution — which cannot be hardwired upfront [8,9].

Technically, we substantiate this multi-model specification and monitoring framework through three novel contributions, which together provide a complete formalization of the approach outlined in [2]:

1. First and foremost, we tackle the infinity induced by the presence of data, which, in general, leads to undecidability of monitoring [12]. Specifically, we recast in our setting faithful data abstraction techniques for variable-to-constant comparisons, originally developed towards verification of data Petri nets [19,35]. This allows us to obtain finitely representable monitors based on traditional finite-state automata.
2. Second, we construct homogeneous monitors for declarative and procedural components, and define how to combine them into a unique, global monitor for conflict detection; this is obtained by computing a form of automata cross product, which conceptually describes the execution traces of a hybrid specification where all procedural and declarative components are simultaneously

applied. This corresponds to the concurrent execution of all Petri nets contained in the specification, while at the same time checking the actual and possible satisfaction of constraints.

3. Third, we further refine the global monitor by acknowledging that monitoring should continue to provide meaningful feedback even after a violation has been detected. Specifically, we ensure that when the global monitor returns a permanent violation (due to an explicit violation of a process specification, or the presence of a conflict), the monitor continues to operate, and can distinguish which continuations may lead to incur in additional violations. We associate a violation cost to each component, equipping the global monitor with the ability of returning the best-possible next events, i.e., events that would lead to the minimum total violation cost that can be obtained based on the events executed thus far.

The presented monitoring approach has been implemented in a proof-of-concept tool, which is used for extensive scalability experiments using a scenario from the medical domain. The experiments cover input specifications of different complexity, expressiveness, and size, showing the feasibility of the approach, but also highlighting its current limitations.

The remainder of this paper is structured as follows. Section 2 provides an example scenario from the medical domain. Section 3 briefly describes the M3 Framework. Section 4 and 5 introduce the formal definitions and algorithmic solutions underlying the proposed monitoring approach. Section 6 presents the evaluation of the proposed monitoring approach. Section 7 discusses the related work and Section 8 concludes the paper.

2. Example scenario

Let us now introduce a simple scenario, taken from the medical domain, that demonstrates two clinical guidelines together with one background knowledge constraint. Both guidelines and the constraint are adopted from the scenario reported in [51]. We use this scenario to showcase when comorbidity issues may arise during the treatment process and how the (monitoring) approach proposed in this paper can help healthcare providers to address such issues. The same guidelines are discussed in more detail in [4] and originally provided by the British National Institute for Health and Care Excellence (www.nice.org.uk).[1]

Scenario. A patient shows up at the emergency department with an acute stomach pain and immediately gets assigned to one of the emergency physicians. After having looked through the patient's medical history and a brief examination, the doctor concludes that the patient experiences a peptic ulcer (PU) relapse and chooses to follow the standard PU treatment clinical guideline (CG). The standard procedure provided for this CG envisions that, first, a helicobacter pylori test ($HPte$) is performed, and then, based on its outcome, either prescribes amoxicillin administration (AT), if the test is positive, or gastric acidity reduction (GAR) otherwise. Afterwards, a peptic ulcer evaluation exam ($PUev$) is performed so as to estimate the effects of the therapy.

After having performed the helicobacter test, the doctor prescribes the amoxicillin-based treatment. However, the doctor does not know that the patient also suffers from venous thromboembolism (VT) and has been already followed by a cardiologist from the same hospital.

In acute phases, VT requires an immediate intervention decision ($IntD$), chosen among three different possibilities based on the situation of the specific patient. Mechanical intervention (MI) 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 possibilities are an anticoagulant therapy based on warfarin (WT), or a thrombolytic therapy (TT). To help the patient coping with the acute phase of VT, the cardiologist had prescribed the warfarin therapy that the patient was already following at the moment when the test was performed.

It is known that any 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, it would be important to alert healthcare providers simultaneously following one patient with multiple conditions about the impossibility of administering warfarin and amoxicillin together (that is, activities AT and WT cannot coexist in the same treatment case). This constraint (we call it C) is an example of basic medical knowledge [9] forming an additional process specification in the given scenario.

As we have mentioned above, the patient was tested positive on helicobacter pylori while undergoing the warfarin therapy. Using constraint C, even without knowing about other therapies, the doctor can be alerted by the hospital medical workflow management system about a conflict arising between two CGs and C. For this outlier, but possible situation the doctor is offered three alternatives to proceed:

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

After have been informed about the presence of the conflict, the emergency physician can assess the patient's case with respect to the other treatment in progress, weigh the implications of one choice over the others, and finally make an informed decision. Since one should avoid by any means complications such as serious bleeding (which could happen if C is ignored), and given that skipping the amoxicillin therapy is rather costly, due the lack of viable alternatives for treating peptic ulcer in case of helicobacter pylori, the doctor chooses to violate the VT procedure as there are other anticoagulants (e.g., heparin) that may be less effective but do not interact strongly with amoxicillin.

The above reasoning can be achieved by assigning violation costs to the CG specifications and background knowledge constraints. Like that, the hospital workflow management system can use this information to provide a more apt solution. For example, the option to violate C should come with the highest cost (among the other process specifications).

Notice that we can also deal with (meta-)constraints [15] that impose conditions on the process execution depending on the truth value of other constraints. As an example, we will specify a *meta-constraint* dictating that if constraint C gets violated, then the patient must be placed under heightened observation, which is represented as requiring an additional activity $HObs$.

3. M3 framework

In our previous work [2], we have introduced the Multi-Model Monitoring Framework (M3 Framework) to address scenarios like the one described in Section 2. In this section we give a brief overview of this framework, thus setting the stage for the monitoring approach presented in Sections 4 and 5. For more details on the framework itself we refer the reader to [2].

The M3 framework is organized in phases which allow for eliciting, managing, and monitoring hybrid process specifications. It supports scenarios with multiple procedural and declarative specifications, where the former can be executed concurrently and the latter work as global constraints that implicitly induce additional dependencies between the procedural specifications. Both declarative and procedural specifications are stored in a specification repository (see Fig. 1), which is constantly updated during the model elicitation phase, and then used

[1] While [4,51] do not cite specific guidelines, based on our research the relevant guidelines are CG184, NG158, and TA287, with the latter highlighting Warfarin as a recommended anticoagulation treatment for venous thromboembolism.

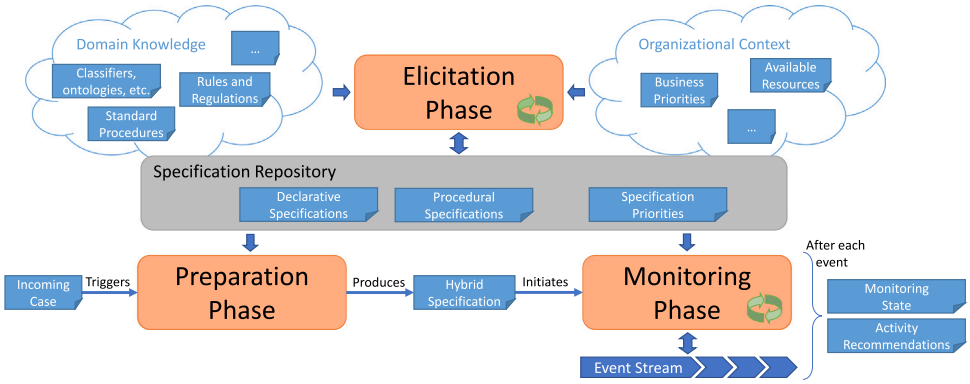


Fig. 1. Conceptual overview of the M3 Framework.

for monitoring individual cases in the case-specific preparation and monitoring phases.

Elicitation Phase. The elicitation phase of the M3 Framework is envisioned as a continuous case-agnostic phase, during which domain knowledge (standard procedures, classifiers, etc.) and organizational context (business priorities, available resources, etc.) are transformed into concrete process specifications, which are then stored in a dedicated specification repository. Both declarative and procedural specifications are supported, and multiple specifications of either paradigm can be used to represent a single global process, thus supporting not only hybrid process specifications, but also, for example, component-based and aspect-oriented modeling approaches [34]. Full agreement between the individual process specifications is not assumed, instead each specification is associated with a priority value that is used during monitoring to provide guidance to the user in resolving any potential conflicts. Each specification is also associated with a priority value that is later used in the monitoring phase to handle conflicts. Concrete approaches for handling the elicitation phase are out of the scope of this paper.

Preparation Phase. The preparation phase of the M3 Framework is envisioned as a case-specific non-recurrent phase in which an incoming case is assessed, relevant process specifications are selected, and a corresponding hybrid specification is automatically created as an input for the following monitoring phase. This hybrid specification will encompass the combined behavior of all selected specifications, the corresponding specification priorities, and, if required, also additional case-specific modifications. The selected process specifications can be smaller fragments of a single business process, but also fragments or full specifications of multiple, separately defined (but concurrently executed) business processes. The main functionalities of this phase are formalized in Section 5 and evaluated in Section 6.

Monitoring Phase. The monitoring phase of the M3 Framework is envisioned as an ongoing case-specific phase, covering the entire duration of the case being monitored. During this phase, the state of the monitor and the set of next recommended actions (with payloads) are updated after the occurrence of each event, therefore providing guidance towards the successful completion of the case. The monitoring phase is also envisioned to allow for optional modifications of the current hybrid process specification so as to handle emerging case characteristics that were unforeseen during the preparation phase. However the implementation of this functionality is currently left for future work. Similarly to the previous phase, the main functionalities of this phase are formalized in Section 5 and evaluated in Section 6 (including concrete examples of the output of this phase).

4. Process components

In this section, we define the models used to specify declarative and procedural data-aware process components based on Multi Perspective-Declare (MP-Declare) [11,20,39]) and data Petri nets (DPNs [35,43]) respectively.

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 $A = \{a_1, \dots, a_\ell\}$ 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 by its name). With $\mathcal{N}_\mathcal{E} = \bigcup_{\langle n,A\rangle\in\mathcal{E}} n$ and $\mathcal{A}_\mathcal{E} = \bigcup_{\langle n,A\rangle\in\mathcal{E}} A$ we respectively denote the sets of all event and attribute names from $\mathcal{E}$.

An *event* of signature $\langle n, A\rangle$ is a pair $e = \langle n, \nu\rangle$ where $\nu : A \mapsto \mathbb{R}$ is a total value assignment function. For simplicity, we assume attributes ranging over reals equipped with comparison predicates (simpler types such as strings with equality and booleans can be seamlessly encoded). We call a finite sequence $\sigma = e_1 \cdots e_\ell$ of events a *trace*. Given a trace σ, $|\sigma|$ denotes its *length* and $\sigma(i)$, for $1 \le i \le |\sigma|$, denotes e_i.

Multi-Perspective Declare with Local Conditions. For declarative process components, we rely on a multi-perspective variant of the well-known process modeling language Declare [50]. A Declare model consists of template-based *constraints* that must be satisfied throughout the process execution. The template syntax and semantics are formalized using Linear Temporal Logic over finite traces (LTL_f) [44].

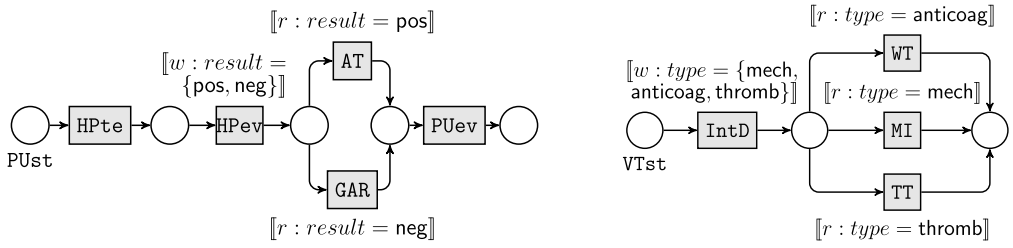
Definition 1. An *LMP-Declare constraint* is an expression of the form: $\Phi := \top \mid x \mid a \odot c \mid \mathbf{X}\Phi \mid \Phi_1 \mathbf{U} \Phi_2 \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2$, where $x \in \mathcal{N}_\mathcal{E}$, $a \in \mathcal{A}_\mathcal{E}$, $\odot \in \{<, =, >\}$ and $c \in \mathbb{R}$.

Formulas of the form $a \odot c$ and x are called *atomic*. We define the usual abbreviations: $a \le c = \neg(a > c)$, $a \ge c = \neg(a < c)$, and $a \ne c = \neg(a = c)$; $\Phi_1 \vee \Phi_2 = \neg(\neg\Phi_1 \wedge \neg\Phi_2)$; $\Phi_1 \rightarrow \Phi_2 = \neg\Phi_1 \vee \Phi_2$; $\mathbf{F}\Phi = \top\mathbf{U}\Phi$ (*eventually*); and $\mathbf{G}\Phi = \neg\mathbf{F}\neg\Phi$ (*globally*).

Notice that the language of boolean combinations of attribute-to-constant comparisons without event variables closely resembles that of variable-to-constant conditions in [35], thus providing a good basis for combining declarative constraints with procedural models expressed with DPNs. 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).

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

- $\sigma, i \vDash \top$;
- $\sigma, i \vDash x$ iff $\sigma(i) = \langle n, \nu\rangle$ and $x = n$;
- $\sigma, i \vDash a \odot c$ iff $\sigma(i) = \langle n, \nu\rangle$, $\nu(a)$ is defined and $\nu(a) \odot c$;



(a) DPN representing peptic ulcer (PU) treatment (b) DPN representing venous thromboembolism (VT) treatment

Fig. 2. 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.

- $\sigma, i \vDash \varphi$ iff $\sigma(i) \vDash \varphi$;
- $\sigma, i \vDash \Phi_1 \wedge \Phi_2$ iff $\sigma, i \vDash \Phi_1$ and $\sigma, i \vDash \Phi_2$;
- $\sigma, i \vDash \neg\Phi$ iff $\sigma, i \nvDash \Phi_1$;
- $\sigma, i \vDash \mathbf{X}\Phi$ iff $i < |\sigma|$ and $\sigma, i+1 \vDash \Phi$;
- $\sigma, i \vDash \Phi_1 \mathbf{U} \Phi_2$ iff there exists j, $1 \leq j \leq |\sigma|$, s.t. $\sigma, j \vDash \Phi_2$ and for every k, $1 \leq k \leq j-1$, we have $\sigma, k \vDash \Phi_1$.

Example 1. Consider two event signatures $\langle a, \{x, y\}\rangle$ and $\langle b, \{z\}\rangle$. The *negation response* LMP-Declare constraint $\mathbf{G}(a \rightarrow \neg\mathbf{XF}(b \wedge z > 10))$ captures that whenever event a occurs then b cannot later occur with its attribute z carrying a value greater than 10.

Data Petri nets. We define data Petri nets (DPNs) by adjusting [35,43] 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 5).

Let $\mathcal{E}$ be a finite set of event signatures. $\mathcal{G}_\mathcal{E}$ is the language of *guards* over $\mathcal{E}$, where a guard is a boolean combination of atomic formulas $a \odot c$. We can then specialize the notion of satisfaction to guards: given an assignment $\alpha : \mathcal{A}_\mathcal{E} \rightarrow \mathbb{R}$ and an atomic condition $a \odot c$, we have that $\alpha \vDash a \odot c$ iff $\alpha(a) \odot c$. Boolean combinations of atomic conditions are defined as usual. We denote by $Var(\gamma)$ the set of attributes mentioned in a guard γ.

Definition 2. A *Petri net with data and variable-to-constant conditions* (DPN) over a set $\mathcal{E}$ of event signatures is a tuple $D = (P, T, F, l, V, r, w)$, where:

- (P, T, F) is a Petri net graph, where P and T are two finite disjoint sets of *places* and *transitions*, and $F : (P \times T) \cup (T \times P) \rightarrow \mathbb{N}$ 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 *read* and *write guard-assignment* functions, mapping every $t \in T$ 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. Given $x \in P \cup T$, the *preset* and the *postset* of x are, respectively, the sets ${}^\bullet x = \{y \mid F(y, x) > 0\}$ and $x^\bullet := \{y \mid F(x, y) > 0\}$.

Example 2. Fig. 2 shows two DPNs encoding the clinical guideline fragments discussed in Section 2. 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: *(1)* $M : P \rightarrow \mathbb{N}$ is a total *marking* function, assigning a number $M(p)$ of *tokens* to every place $p \in P$; *(2)* $\alpha : V \rightarrow \mathbb{R}$ is a total *variable valuation* assigning a real value to every variable in V. A DPN can progress from one state to another if one of its enabled transitions fires. A transition is enabled if all its input places contain sufficiently many tokens to consume and both its read and write guards are satisfied. More formally, given a a DPN $D = (P, T, F, l, V, r, w)$, transition $t \in T$ is *enabled* in state (M, α) under partial valuation $\beta : V \nrightarrow \mathbb{R}$, denoted $(M, \alpha)[t, \beta\rangle$, iff: *(1)* β is defined on all variables $v \in Var_r(t) \cup Var_w(t)$; *(2)* $\beta(v) = \alpha(v)$, for every $v \in Var_r(t)$; *(3)* $\beta \vDash r(t)$ and $\beta \vDash w(t)$; *(4)* $M(p) \geq F(p, t)$, for every $p \in {}^\bullet t$.

Then, if $t \in T$ is enabled in state (M, α) under β, it can *fire* and produce a new state (M', α') s.t. *(1)* $M'(p) = M(p) - F(p, t) + F(t, p)$, for every $p \in P$; *(2)* $\alpha'(v) = \beta(v)$, for every $v \in Var_w(t)$; *(3)* $\alpha'(v) = \alpha(v)$, for every $v \in V \setminus Var_w(t)$. We denote transition firing as $(M, \alpha)[t, \beta\rangle(M', \alpha')$. 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* and *well-formed (over their respective set of event signatures $\mathcal{E}$)*. The former means that in every reachable state, each place can have at most one token. This is done for convenience (our approach seamlessly works for k-bounded nets). The latter means that transitions and event signatures are compatible, that is: *(i)* for every $t \in T$ with $l(t) = n$, where $\langle n, A\rangle \in \mathcal{E}$, we have that $Var_w(t) = A$; *(ii)* for every $t \in T$ 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.

Consider a *DPN with initial state and final marking* (DPNIF), denoted as $\bar{D} = (D, (M_0, \alpha_0), M_f)$, where D is a DPN, (M_0, α_0) is the initial sate and M_f is the 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 $M = M_f$.

Let us now define when a trace *complies* 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, we have that $\iota(e_2)$ comes later than $\iota(e_1)$ in σ_2. This notion allows to map traces into (possibly longer) runs of a DPNIF.

Definition 3. A trace $\sigma = e_1 \cdots e_n$ *complies* with a DPNIF $\bar{D}$ with labeling function l if there exists a run ρ of $\bar{D}$ and an order-preserving injection ι from σ to ρ such that:

- for every $e = \langle n, \nu \rangle$ in σ s.t. $\iota(e) = [t, \beta\rangle$, we have that $l(t) = n$ and β corresponds to ν for the written variables $Var_w(t)$[2];
- every element $[t, \beta\rangle$ in ρ that does not correspond to any element from σ via ι is so that $l(t) = \tau$.

5. 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, that is, LMP-Declare constraints and DPNIFs over $\mathcal{E}$. Then, 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. An important clarification, when characterizing the concurrent execution over multiple DPNIFs, is that 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 occurs, 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.

5.1. 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, which induce infinitely many states to consider in the DPNs (even when the net is bounded). We deal with this infinity by building on the faithful abstraction techniques studied in [35], recasting them in our more complex setting. The idea is to avoid referring to single real values, and instead predicate over a fixed number of intervals, which in turn leads us to propositional reasoning. This comes from the observation that data conditions can distinguish between only those constants that are explicitly mentioned therein; hence, it suffices to consider only the constants used in the process components (i.e., some atomic condition, guard, or initial DPN assignment) to delimit the intervals to consider.

Technically, let $C = \{c_1, \ldots, c_m\}$ be a finite set of values from $\mathbb{R}$ assuming, without loss of generality, that $c_i < c_{i+1}$, for $1 \leq i \leq m-1$. We then partition $\mathbb{R}$ into $\mathcal{P}_C = \{(-\infty, c_1), (c_m, \infty)\} \cup \{(c_i, c_i) \mid 1 \leq i \leq m\} \cup \{(c_i, c_{i+1}) \mid 1 \leq i \leq m-1\}$. Notice that $\mathcal{P}_C$ is finite, with a size that is linear in m, and can be also seen as a fixed set of propositions, which is crucial for our approach. Each interval in the partition is an *equivalence region* for the satisfaction of the atomic formulas 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 $R \in \mathcal{P}_C$, then $\alpha \vDash v \odot c$ if and only if $\alpha' \vDash v \odot c$.

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 $\tilde{\alpha} : V \to \mathcal{P}_C$ so that, for every $a \in V$, $\tilde{\alpha}(a)$ returns the unique interval to which $\alpha(a)$ belongs. We can then use $\tilde{\alpha}$ to check whether a formula holds over α. For example, $\alpha(a)$ satisfies $a > c$ with $c \in C$ if and only if $\tilde{\alpha}(a) = (c_1, c_2)$ with $c_1 > c$. The same reasoning can be similarly done for other comparison operators and carries over boolean combinations of atomic formulas. The key observation here is that doing this check amounts to propositional reasoning, and so does checking satisfiability of atomic formulas: in fact, since both V and $\mathcal{P}_C$ are finite, there are only finitely many region assignments that can be defined from V to $\mathcal{P}_C$.

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 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 C are fixed as described above.

5.2. Encoding into guarded finite-state automata

To capture the execution semantics of process components, we introduce a semi-symbolic automaton whose transitions are decorated with boolean combinations of atomic formula. For ease of reference, given an event signature $\mathcal{E}$, we denote with $\mathcal{L}_\mathcal{E}$ the language of such boolean combinations.

Definition 4. A *guarded finite-state automaton* (GFA) over set $\mathcal{E}$ of event signatures is a tuple $\mathcal{A} = \langle Q, q_0, \rightarrow, F \rangle$, 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 relation*; and *(iv)* $F \subseteq Q$ is the set of *final states*. 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 $e_i \vDash \varphi_i$, for $1 \leq i \leq m$. 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 C to a finite set of constants including all those mentioned in the automata guards, we can apply the interval abstraction from Section 5.1. 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 $\mathcal{P}_C$. 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 $\mathcal{P}_C$, s.t. every $\nu(a) = c$ is substituted either with $[c, c]$, if $[c, c] \in \mathcal{P}_C$, or with (c', c'') s.t. $c \in (c', c'')$. From here, we denote the abstract trace as $\sigma^{\mathcal{P}_C}$.

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

From LMP-Declare constraints to GFAs. Finite-state automata have been extensively used for monitoring of LTL_f formula and, in particular, Declare constraints [15,18]. In our context, the latter are represented using GFAs in which guards carry only propositions referring to activity names. In the case of LMP-Declare, the guards are formulas of $\mathcal{L}_\mathcal{E}$ which, however, thanks to the interval abstraction, can be represented using a finite number of proposition. Hence, we can build on top of the standard finite-state automaton construction for a Declare constraint [15], in which transitions are labeled by the formulas from $\mathcal{L}_\mathcal{E}$ mentioned within the constraint. We keep only those transitions whose labels are satisfiable formulas (as discussed in Section 5.1, satisfiability checking is done via propositional reasoning). Lastly, it

[2] 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.

is important to mention that the obtained GFA is kept *complete* so that each of its states can process every event of a signature from $\mathcal{E}$.

From DPNIFs to GFAs. Next, we discuss the key aspects of constructing a GFA which accepts all and only those traces that comply with a given DPNIF $\bar{D} = (D, (M_0, \alpha_0), M_f)$.

The main issue is that the set S of DPN states reachable from (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. The infinity can be tamed using the partitioning strategy defined above, which induces a partition of S into equivalence classes, according to the intervals assigned to the variables of D. Formally, 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 $R \in \mathcal{P}_C$ 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 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 5.1 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 5 (*GFA Induced by a DPNIF*)**.** For a given 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 $\mathcal{A}_D = \langle Q, q_0, \rightarrow, F \rangle$ s.t.:

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) $\psi = \bigwedge_{v \in Var(w(t))} \phi_v$ such that:
      (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 $\{(t, \tilde{\beta}) \mid (M, \alpha)[t, \beta\rangle$ with $t \in T, \beta(v) \in \tilde{\beta}(v), \alpha(v) \in \tilde{\alpha}(v)$, for $v \in V\}$. Here $\tilde{\beta}$ matches only the "allowed" regions. That is, for every t, we construct multiple $\tilde{\beta}$ that account for *all possible combinations* of equivalence regions assigned to each variable in $w(t)$ and $r(t)$ s.t. $\tilde{\beta} \models w(t)$ and $\tilde{\beta} \models r(t)$.
- $guard(t, \tilde{\alpha})$ returns a formula ψ as in Definition 3(c).
- $fire(M, t, \tilde{\beta})$ returns a pair $(M, \tilde{\alpha})$ as in Definition 3.

It is easy to see that the above functions are computable. For *enabled* there are always 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 5 that uses $\tilde{\beta}$ instead of $\tilde{\alpha}'$, and next states returned by $fire$ can be generated using the DPN firing rule, proviso that it has to be invoked in the context of equivalence regions.

The actual algorithm builds on top of the classical one for the Petri net reachability graph construction (see, e.g., [46]). In the algorithm, we treat silent transitions as regular ϵ-transitions (assuming that $\mathcal{L}_\mathcal{E}$ as well as $\longrightarrow$ of the output automaton are suitably extended with τ). We discuss later on how such ϵ-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
```

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

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

(M1) Given that the algorithm treats τ-transitions as normal ones, we need to compile them away from the so-obtained GFA $\mathcal{A}_D$. This is done via the standard procedure for finite-state automata (see, e.g., [33]) adopted to our setting. This procedure allows to collapse sequences of states in which each $q \in Q$ is s.t. there are k runs $q \xrightarrow{\tau} \cdots \xrightarrow{\tau} q_i'$ (where every transition is labeled with τ) and $q_i' \xrightarrow{\varphi} q_i''$, with $\varphi \neq \tau$ and $1 \leq i \leq k$. The removal is done by replacing each $q \in Q$ with the set of all the q_i' states and by adding transitions to predecessors of q (if any) as well as q_i''. 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. This is done by introducing extra loops in $\rightarrow_i$ from Definition 5 as follows: for every $q \in Q_i$ we insert a looping transition $q \xrightarrow{\psi} q$, where $\psi = \bigwedge_{a \in (\mathcal{N}_\mathcal{E} \setminus \bigcup_{t \in T} l(t))} a$. Like that, the GFA can 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 net executions where a partial run cannot be completed into a proper run due to a data-related deadlock. This can occur due to the violation of read versus write guards of all transitions that, from the control-flow perspective, could actually fire. To deal with this issue, 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 we add $(M, \tilde{\alpha}) \xrightarrow{\psi} (M, \tilde{\alpha})$ to $\rightarrow_i$, where ψ is as in Definition 5 (see 3). This is only done when ψ is 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 as in Definition 5 (see 3), with all φ_v being computed now for a combination of equivalence regions from $\mathcal{P}_C$,

each of which is composed into (partial) variable region valuation $\tilde{\beta} : V \to \mathcal{P}_C$ s.t. $\tilde{\beta} \not\models v \odot c$, for every $v \odot c$ in $w(t)$. This is only done if ψ is actually satisfiable. This step can be also optimized 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.

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

5.3. Combining GFAs

Given a hybrid process $\mathcal{H}$ with n components, we know from Section 5.2 how to compute a GFA $\mathcal{A}_i$ for each component h_i of $\mathcal{H}$. Such GFA is addition minimized and determinized, which means that, being $\mathcal{A}_i$ complete, it will have a single trap state capturing all traces that permanently violate h_i.

To perform monitoring, we follow the approach of colored automata [15,40] and label each automaton state q of $\mathcal{A}_i$ with one of four truth values, respectively indicating whether the corresponding process component is *temporarily satisfied* (TS), *temporarily violated* (TV), *permanently satisfied* (PS), or *permanently violated* (PV) in q. As for constraints, these values are interpreted exactly like in [15,40]. 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). At the level of $\mathcal{A}_i$, we define a labeling function ζ_i as follows: *(i)* $\zeta_i(q) = PS$ iff $q \in F$ and all transitions outgoing from q are self-loops; *(ii)* $\zeta_i(q) = TS$ iff $q \in F$ and there is some transition outgoing from q that is not a self-loop; *(iii)* $\zeta_i(q) = PV$ iff $q \notin F$ and all transitions outgoing from q are self-loops; *(iv)* $\zeta_i(q) = TV$ iff $q \notin F$ and there is some transition outgoing from q 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, $\mathcal{A} = \langle Q, q_0, \to, F \rangle$, 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)* $F = F_1 \times \cdots \times F_n$. Notice that guards of $\mathcal{A}$ must be satisfiable by some event. Otherwise, related labeled transitions are omitted as there is no event to trigger them.

Given labeling functions ζ_i for components h_i of $\mathcal{H}$, we define a labeling function ζ on the states $q = \langle q_1, \ldots, q_n \rangle \in Q$ of $\mathcal{A}$, indicating whether all components from $\mathcal{H}$ are overall temporarily/permanently violated/satisfied. Formally, $\zeta : Q \mapsto \{TS, TV, PS, PV\}$ is s.t.: *(i)* $\zeta(q) = PV$ iff $\zeta_i(q_i) = PV$, for some $1 \le i \le n$; *(ii)* $\zeta(q) = PS$ iff $\zeta_i(q_i) = PS$, for all $1 \le i \le n$; *(iii)* $\zeta(q) = TS$ iff $\zeta_i(q_i) = TS$, for all $1 \le i \le n$; *(iv)* $\zeta(q) = TV$, otherwise.

5.4. 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 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.

To augment $\mathcal{A}$ with costs, we associate each of its states $q \in Q$ with two cost indicators: *(i)* a value $cost_{cur}(q)$, which contains the sum of the costs associated with the constraints violated in q; *(ii)* a value $cost_{best}(q)$, which contains the best value $cost_{cur}(q')$, Assuming that c_i is the cost associated with the violation of constraint φ_i, we compute $cost_{cur}$ and $cost_{best}$ as follows:

1. for every state $q = \langle q_1, \ldots, q_n \rangle \in Q$, let
$$cost^0_{best}(q) = cost_{cur}(q) = \sum_{1 \le i \le n} cost(q_i),$$
where $cost(q_i) = 0$ if $q_i \in F_i$, and 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 $q \in Q$, $cost^{i+1}_{best}(q) := \min\{cost^i_{best}(q') \mid q \xrightarrow{\varphi} q'\} \cup \{cost_{cur}(q)\}$;
3. return $cost_{best}$.

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

We can also see that the algorithm is correct, i.e., that if $cost_{best}(q) = \kappa$ then, from q: *(i)* there exists a path in $\mathcal{A}$ to some state q' s.t. $cost_{cur}(q') = \kappa$, and *(ii)* there exists no path to some state q'' s.t. $cost_{cur}(q'') < \kappa$. These come as a consequence of step 2 of the algorithm. By this, we have that, after the ith 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, it means that even considering longer paths will not improve the value, i.e., the value is minimal.

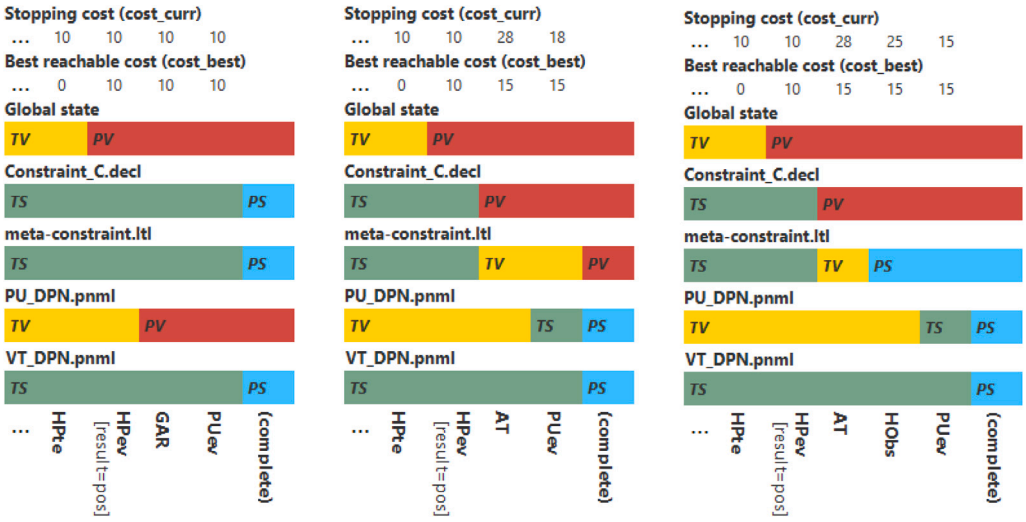
Using $\mathcal{A}$, $cost_{cur}$ and $cost_{best}$, we can find the next "best events", i.e., those events that allow for satisfying the combination of constraints that guarantees the minimum cost. Technically, let $\sigma = e_1 \cdots e_\ell$ be the input trace and consider the set $\Gamma = \{\rho_1, \ldots, \rho_n\}$ of the runs of $\mathcal{A}$ on σ. Let $q_{i\ell}$ be the last state of each ρ_i and let $\hat{q} = \arg\min_{q \in \{q_{1\ell}, \ldots, q_{n\ell}\}} \{cost_{best}(q)\}$.

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^* with $e^* \models \varphi$, for φ 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 5.1. In particular, notice that there may be infinitely many different best events, obtained by different attribute assignments within the same intervals.

6. Evaluation of the monitoring approach

In this section we first provide three concrete examples of the output of the monitoring approach from Section 5, which are then followed by an extensive set of scalability experiments. The source code of the corresponding proof-of-concept implementation along with all input files used in this section are publicly available at https://git.io/JM0iA.



(a) PU guideline violation. (b) Constraint C violation. (c) Meta-constraint fulfillment.

Fig. 3. Monitoring results for the three example traces (activities before $HPte$ omitted)

6.1. Examples of monitoring results

Recall that, in the example scenario (Section 2), (1) the patient was undergoing warfarin therapy (WT) when the treatment of peptic ulcer (PU) started, (2) PU treatment guideline mandates amoxicillin therapy (AT) if the helicobacter pylori test returns positive ($HPev_{[result=positive]}$), (3) AT and WT cannot coexist in the same case (constraint C), (4) and if constraint C is violated, then we require an additional activity $HObs$ (meta-constraint).

Keeping the above in mind, we hand-crafted three traces to demonstrate the output of the monitoring approach from Section 5). For each trace, we used the process specifications from the example scenario with the following violation costs: PU – 10; VT – 5; Constraint C – 15; Meta-constraint – 3.

The monitoring results shown on Fig. 3 omit all activities before $HPte$, as the results up to and including that event are the same for all three traces. More specifically, the stopping cost (i.e., the sum of all violation costs if process execution would stop after $HPte$) is 10 because the PU guideline requires further activities to occur (also reflected by the corresponding monitoring status TV). Meanwhile, in the same state, the best reachable cost of 0 indicates that finishing the treatment without any violations would still be possible (also reflected by the global monitoring status TV).

However, after the event $HPev_{[result=positive]}$, we reach a permanent violation in the global state, despite the fact that no concrete process specifications are violated. This is because, the PU guideline now requires the activity AT to occur, which, given that in our scenario WT has already occurred, would violate constraint C. Our monitoring approach can detect such issues during the ongoing treatment case and provide recommendations on how to proceed based on the violation costs of the input process specifications.

In the first example (Fig. 3(a)), the doctor follows the recommendations of our monitoring approach by first executing GAR (violating the PU guideline) and then $PUev$. This leads to a total violation cost of 10, which is also the lowest reachable total violation cost after the event $HPev_{[result=positive]}$ had occurred. In the second example (Fig. 3(b)) the doctor instead decided to continue following the PU guideline by executing the activity AT, followed by the activity $PUev$. This lead to the total violation cost of 18 since both constraint C and the meta-constraint were violated. Finally, in the third example (Fig. 3(c)) the doctor also decided to continue following the PU guideline, but in this case executed the additional activity $HObs$ thus fulfilling the meta-constraint and leading to total violation cost of 15.

6.2. Scalability experiments

All scalability experiments are based on extending the input specifications from Section 2 (excluding the meta-constraint) to explore four scalability dimensions. The first dimension considers the input size of the DPN components and the number of LMP-Declare constraints. The second dimension investigates the effect of the guard placement and the total number of guards in the DPNs. The third dimension tests the effect of synchronization activities in the DPN specifications. The fourth dimension investigates the performance of the approach based on the number of individual input specifications. The first and the last dimensions are explored considering DPNs with and without guards. All experiments were performed on a 6-core Intel i7 10850H machine with 2 × 16GB of RAM.

Two artificially generated event logs are used for the tests. Both contain 200 traces, including up to 50% of negative examples depending on the specification used in each test. The first log is used to test the first three scalability dimensions, while the second one is used for testing the last dimension.

6.2.1. Input specification size

The size of input specifications greatly affects the performance of our approach as larger specifications induce more states to be considered in the final GFA, which also needs to account for more interleavings. To test scalability along this dimension, we have performed experiments on iteratively increased specifications from Section 6.1 obtained by creating specification copies with renamed activities and appending such copies to original models (notice that for DPNs this is done via sequential composition of nets). We report on the test results in Table 1, where 'Test no. 0' uses the original models and all the consecutive tests use its iteratively increased variants. Two sets of tests were performed, one considering only the control flow (by removing all the DPN guards), and the other considering both control flow and data perspectives.

With the available RAM, we were able to scale from 5 places and 5 transitions in the PU DPN, 3 places and 4 transitions in the VT DPN, and 1 constraint in the Declare model (Test no. 0) up to 29 places and 35 transitions in the extended PU DPN, 15 places and 28 transitions in the extended VT DPN, and 7 constraints in the extended Declare model (Test no. 6). The results related to the GFA (construction time and number of states) indicate that realistic process specifications are feasible even with the current prototype implementation (especially if only the control flow is considered), but RAM required (as well as the

Table 1
Scalability wrt the size of input specifications.

Test no.	Control Flow Only			Control Flow & Data		
	GFA Time (s)	GFA States	Event Avg. (ms)	GFA Time (s)	GFA States	Event Avg. (ms)
0	0.035	54	0.659	0.054	80	0.762
1	0.068	272	0.756	0.175	462	0.883
2	0.187	1026	0.768	0.897	1824	0.940
3	0.610	3484	0.842	3.044	6306	1.009
4	2.096	11278	0.868	10.740	20588	1.116
5	7.644	35560	0.934	43.283	65230	1.228
6	28.552	110298	0.951	199.689	202952	1.304

Table 2
Scalability wrt the number of guards and position of a guard in DPN control flow.

Test no.	Number of guards			Guard position		
	GFA Time (s)	GFA States	Event Avg. (ms)	GFA Time (s)	GFA States	Event Avg. (ms)
0	27.577	110298	0.914	27.938	110298	0.956
1	36.624	112592	0.947	36.417	112592	0.988
2	46.076	116118	1.029	37.379	113816	1.004
3	56.978	121436	1.100	38.716	115572	1.006
4	73.798	129590	1.126	38.323	118292	1.003
5	94.108	142476	1.183	40.134	122652	1.015
6	138.823	163958	1.277	41.463	130076	1.013
7	205.075	202952	1.293	47.936	143952	0.933

Table 3
Scalability wrt the number of synchronizations and position of a synchronization in DPN control flow.

Test no.	Number of syncronisations			Syncronisation position		
	GFA Time (s)	GFA States	Event Avg. (ms)	GFA Time (s)	GFA States	Event Avg. (ms)
0	28.670	110298	0.945	28.699	110298	0.931
1	29.140	107463	0.920	28.162	107463	0.897
2	29.807	103228	0.984	29.341	102891	0.947
3	28.581	97173	0.956	27.235	96521	0.967
4	26.907	88866	0.930	23.900	88323	0.920
5	24.331	78631	0.967	21.137	80073	0.975
6	19.694	69870	0.892	20.845	79949	0.982

Table 4
Scalability wrt the number of input specifications.

Test no.	Control Flow Only			Control Flow & Data		
	GFA Time (s)	GFA States	Event Avg. (ms)	GFA Time (s)	GFA States	Event Avg. (ms)
0	0.017	4	0.785	0.034	6	0.864
1	0.030	36	0.801	0.054	64	1.042
2	0.051	216	0.946	0.137	512	1.304
3	0.124	1296	1.034	0.821	4096	1.433
4	0.484	7776	1.146	5.066	32768	1.640
5	2.950	46656	1.194	43.002	262144	1.865

constriction time) start to quickly ramp up once the input specifications become larger. The average processing time of each incoming event remains below 2 ms in all tests.

6.2.2. DPN guards

The results from Section 6.2.1 show that considering the data perspective can increase the computational complexity significantly. To understand better that increase, we use the largest specifications from Table 1 ('Test no. 6', 'Only Control Flow') as the baseline. For each test, we systematically added read and write guards to both DPNs, constraining the behavior of one of the otherwise free-choice decision points in the control flow. The first set of tests focuses on the number of guards by first constraining the behavior of the earliest decision-point of both DPNs (Test no. 1), then the second earliest (Test no. 2), then the third earliest (Test no. 3), etc. until the behavior of all decision points is fully guarded (Test no. 7). The second set of tests follows the same pattern, but instead of adding extra guards, we move the same guards forward by one decision point in each test (see Table 2).

As expected, both the time required for constructing the GFA as well as the number of its states increase as the number of guards grows. Notice that the increasing number of states is not only due to the additional guards in the specifications, but also due to their positions moving more towards the end of the process models. This is confirmed by the second set of tests where the number of GFA states (and also required time) increases in each test despite no additional guards are added (only the position of the guards changes). This indicates that data dependant decision points should be placed as early as possible in the control flow to increase the performance of our approach.

6.2.3. Synchronization activities

In our framework, process specifications can be connected both via declarative constraints as well as activities they share. Each of such activities becomes a synchronization point between the specifications that contain it, since the respective GFA will be able to progress on this activity so that violations are avoided only when executions of all involved specifications containing it will reach a state in which this activity can be executed simultaneously by all such specifications. We performed two sets of tests to check the effect of synchronization points on scalability. Results for both sets are reported in Table 3 make use of the same model from Table 1 ('Test no. 6', 'Only Control Flow') as baseline. There, the overlapping activities were always placed between repetitions of the original control flow pattern and unique names were used for each added activity. The first set of tests focuses on the number of synchronization points by adding one additional overlapping activity in each test, whereas the second set concentrates on the position of synchronization by moving a single overlapping activity towards the end of the control flow.

As expected, both the time required for constructing the GFA and also the number of states in the GFA decrease as the number synchronization points increases. A single synchronization point near the beginning of the process model reduces the number of GFA states by 2835 ('Test no. 0' vs. 'Test no. 1'). This trend becomes more significant when more synchronization points are added. With 6 synchronization points (Test no. 6), the number of GFA states decreases by $\sim 37\%$. However, the placement of the synchronization points also affects the resulting GFA (specifically, the trend is reversed when compared to DPN guard placement): a single synchronization point near the end of the control flow reduces the number of GFA states by $\sim 28\%$ ('Test no. 0' vs. 'Test no. 6'), which is better than having four synchronization points at the beginning of the control flow in the first set of tests.

6.2.4. Number of input specifications

For the sake of completeness, we also present updated results on the scalability w.r.t. the number of input specifications already presented in [2], i.e., we increase the number of procedural specifications by creating copies of the same DPN (with renamed activities/attributes), and we connect each pair of consecutive copies with a declarative process specification consisting of a single constraint. More specifically, we use the VT DPN (Section 2) and a Not Co-Existence constraints between the WT activities of two consecutive copies. 'Test no. 0' contains only the original VT DPN, while each following test adds one copy of the VT DPN and one Declare constraint containing one Not Co-Existence constraint. As in Section 6.2.1, the tests are performed both with and without considering the data perspective (see Table 4).

As in [2], the number of GFA states rapidly increases together with the number of models in input specifications. Interestingly, the

inclusion of the data perspective seems to have a greater impact here than in the previous tests, both in terms of the number of GFA states and time requirements. However, as in [2], the average time required for processing the events, while slightly increasing, remains nearly instantaneous in all the tests.

7. Related work

We provide an account of related work by considering approaches for modeling and monitoring processes specified using procedural and declarative approaches. We point out the lack of hybrid proposals that cover the modeling complexity addressed in our work while providing monitoring capabilities. We give particular consideration to relevant contributions developed within the medical domain and dealing with clinical guidelines.

Monitoring declarative specifications. Our approach has its root in the formal approach to monitoring through runtime verification [36]. This amounts to take a declarative specification of a property of interest and automatically construct a correct-by-design monitor that checks the state of a property on the basis of the trace monitored so far and of all its possible continuations. This realizes a form of *reactive* monitoring that is complementary to *predictive* monitoring [23]. Predictive monitoring implicitly learns from historical (complete) traces a function mapping trace prefixes to some indicator about the (most likely) future outcome(s). As surveyed in [23], the most common indicators are predictions to categorical or numerical values (such as the expected overall cost or completion time of an instance), or to sequences of future events (guessing the most likely next steps of an instance). Hence, while predictive monitoring can learn a probability distribution over the (most likely) next steps given a monitored partial trace, reactive monitoring exhaustively reasons on all its (infinitely many) possible continuations.

Traditionally, runtime verification deals with declarative specifications expressed linear- [6] and branching-time [28] logics interpreted over infinite traces/computation trees. As said, the monitoring output is emitted by the monitor not only depending on the trace monitored so far, but also on its possible continuations and how they interact with the monitored property. Dealing with continuations of infinite length poses the challenge that not all properties are monitorable, in the sense that the monitor may not be able to determine to which output a given trace should be mapped. This is related to the intrinsic nondeterminism of ω-structures and, in turn, calls for an extensive investigation aimed at singling out the largest monitorable fragments of the considered temporal logic [1,6].

A radically different spectrum emerges when continuations have a finite (though unbounded) length, as considered in this work. In this case, specifications expressed in LTL_f and linear dynamic logic over finite traces [18] correspond to properties of regular languages and are always monitorable. More specifically, an arbitrary property can be encoded into a deterministic monitor expressed as a conventional finite-state automaton, enriched with a labeling that maps every automaton state into a single, provably correct monitoring output [16,20,40]. This constitutes the technical backbone of our approach, and has in fact been already applied to monitor (propositional) Declare specifications. In particular, [40] first showed how to deal at once with monitoring of single constraints and their interplay, while [42] continued [40] by showing how this automata-theoretic approach can be used to return fine-grained feedback on conflicts involving multiple constraints. This approach has been further developed in [16,20], formally proving the correctness of this approach, while also showing how to monitor meta-constraints that predicate on the monitoring state of other constraints (e.g., expressing compensation properties on what is expected to hold when a given Declare constraint is permanently violated). The Declare language has also been extended to account for multiple perspectives by considering activity payloads and data-aware constraints, with corresponding monitoring approaches defined either posing assumption on the data domain [21], or accepting incompleteness of the emitted verdict [39]. None of these works has considered a data-aware extension of Declare that indeed admits finitely representable monitors, which we do here focussing on LMP-Declare. Recent works have pushed the boundaries of data-aware extensions of LTL_f constraints [26,27] for which monitors can be constructed. Such approaches subsume LMP-Declare but have not considered the immersion of declarative constraints inside hybrid specifications, as done here.

Monitoring Petri nets. Many Petri net-based monitoring approaches focus on diagnosis of discrete event systems, such that with each observed string of events the diagnosis procedure at hand associates a diagnosis state (such as 'normal', 'faulty' or 'uncertain'). One of the first monitoring approaches dealing with Petri nets is studied in [55]. There, the author proposes an on-line fault detection technique based on monitoring the number of tokens in P-invariants that also allows to "predict" the future system behavior whenever the fault is being detected. A similar idea is also used in [32], where the authors extend original system models with additional places so as to capture P-invariants allowing to detect and isolate system failures. Notice that in both of these works the failure states are "hardwired" in the system models and failures are detected only when loss/duplication of tokens happens in related places. Another seminal work [60] considers system Petri net models without failures and treats each event occurrence that does not match firing conditions properly as a failure. There are also works related to the domain of workflow management systems. For example, [54] proposes a workflow management system that encompasses workflow monitoring and delay prediction modules based on resource-aware Petri nets.

There are many more monitoring/diagnosis approaches based on Petri nets. One of the main features they have in common is that the system is represented as a single monolithic process (although there are works, such as [29,30], that study monitoring approaches for distributed/modular specifications, synchronizations between modules therein have to be encoded a priori), which can be insufficient in domains with highly flexible and knowledge-intensive processes.

Conformance and compliance checking for CIGs. The interplay of multiple process specifications (both procedural and declarative) has however been addressed to some extent from the perspective of conformance checking. A recent work [24] studies the conformance checking task for mixed-paradigm process models that integrate Petri nets and Declare constraints. However, this setting does not consider any activity payloads and, as customary in conformance checking, the authors focus on alignment of complete traces and not on (runtime) monitoring of ongoing incomplete process executions. There are two other research lines that consider multiple process and constraint specifications, both related to the medical domain and often studied in the context of devising adequate treatment for comorbid patients. The first line focuses on interactions between CGs and BMK from the view-point of the conformance checking problem [9] (i.e., how to handle cases where the non-conformance with respect to a clinical guideline is caused by the correct application of basic medical knowledge). The second one studies the same interactions within the GLARE framework, but from the perspective of explainability [52,53,62] (i.e., how can the actions taken during the treatment of a specific patient be automatically explained given the presence of multiple CGs and BMK). While these approaches to a certain extent consider the interplay of procedural and declarative models, their respective tasks of interest are performed on historical data and do not consider streams of events. Interestingly, regardless the main distinction between our on-line monitoring framework and the a posteriori analysis studied along the research lines mentioned above, there are some points in common. First of all, in order to account for all possible interactions between actions, both approaches rely on structures that compute all such interactions explicitly. The only difference lies in their type: while our approach has to account for all possible states of the system, the approach in GLARE would focus only

on concrete CIGs that are relevant to a given patient. GLARE is also able to merge parts of multiple CIGs in order to provide adequate treatments for given comorbid patients. Currently, our monitoring approach does not support this feature, but we plan to work on an automated generation of possible "repair" strategies that could be then selected by the user. Finally, in [10], the authors discuss how GLARE can be used to account for concurrent executions of CIGs being simultaneously applied to one patient, and how to treat exceptions raised by incompatible CIGs. Such exceptions are pre-computed and stored in a special knowledge base. This is very similar to our approach in which we create a product automaton, accounting for all possible interactions between all the hybrid specification components, and assigning to each of its states a violation/satisfaction label.

Formalisms for representing CIGs. Many different formalisms and notations have been developed to represent CIGs. The vast majority adopts a flow-chart paradigm, with atomic activities composed through sequencing, choices, concurrency operators, and more complex control-flow patterns [14,48]. Several efforts have been made towards formalizing the core elements of such proposals in a notation-agnostic way, by employing variants of (colored) Petri nets [7,31,49]. This relates directly to our approach, where procedural process components are represented using data Petri nets, which form a well-behaved fragment of colored Petri nets, amenable to (runtime) verification.

The long-standing debate on how to infuse flexibility in process modeling languages and management systems [57] has pervaded research on CIGs as well. Several approaches have been devised to provide a more flexible execution of CIGs, ranging from adaptive runtime support [58] to exception handling [56]. At the same time, flexibility by design has also been studied through declarative approaches, in particular adapting Declare to CIGs [45]. This proposal is subsumed by the extension of Declare covered in our work.

Given this account, domain-specific hybrid languages combining notations for procedural and declarative CIG modeling could be naturally studied starting from our contribution.

Hybrid models. In addition to the aforementioned approaches, research in relation to hybrid business process representations (HBPRs) is currently ongoing. The term hybrid refers, in this case, to combining declarative and procedural modeling paradigms into a unified modeling approach which would allow expressing both strict and flexible aspects of a single process in the same model. A conceptual framework and a common terminology for these types of models has been proposed recently [3] and a number of open research challenges related to HBPRs have been identified [61]. Some process mining approaches for HBPRs [22,41,59] have also been developed. However, to the best of our knowledge, there are currently no monitoring approaches suitable for hybrid settings.

8. 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 (such as allergies, prior conditions etc.) which further complicates the process execution.

This paper proposes and formalizes a monitoring approach that can take into consideration the interplay of multiple process specifications (both procedural and declarative), and can anticipate possible violations that may occur when executing such specifications all together. Such anticipation can help either to avoid violations or (if avoiding the violations is not possible) to minimize their effect on the whole execution (by considering a total violation cost computed using special violation costs assigned to every single specification). The proposed approach is limited in that it can only provide recommendations "locally" by considering only immediate next events. However, in the future we plan to extend our technique with the ability of providing non-local recommendations on continuations of the trace, which is readily supported by automata-based techniques. We also want to explore different possible execution semantics for concurrent execution of multiple process models, in particular for what concerns local and shared activities.

Another, natural continuation of this work is to adopt a more sophisticated language to express conditions on data attributes, going beyond variable-to-constant comparisons adopted here. We plan to do so by exploiting recent results on data Petri nets and LTL_f [25–27]. However, the more general languages adopt therein call for data abstraction techniques that are more sophisticated than the interval-based propositionalization approach used here. Hence, extending along this direction calls for further investigation, especially from the algorithmic point of view.

To test our approach in practice, we performed various scalability experiments in a proof-of-concept implementation of the proposed approach. On the one hand, they revealed good performance in terms of monitoring incoming events, with average event processing times remaining near-instantaneous in all of the tests. On the other hand, the experiments demonstrated that the main limiting factor is, as expected, the construction of the complete hybrid specification used for monitoring, which requires considerable amount of time and memory and, consequently, limits the size and number of input procedural and declarative specifications. However, we note that these drawbacks can be overcome by incorporating the optimization techniques widely investigated in the automata construction for LTL_f specifications, such as those in [5,17,37,63]. These can indeed be all integrated directly in our approach. Another possibility is to shift some of the computational complexity towards event processing, which is per se very fast and only mildly affected by both the size and number of the input specifications. Furthermore, even without these improvements, the proposed approach is already able to handle processes of realistic, though relatively small, sizes (~ 60 activities across process specifications).

Acknowledgments

The work of A. Alman was supported by the European Social Fund via "ICT programme" measure and by the Estonian Research Council grant PRG1226. F.M. Maggi was supported by the UNIBZ project CAT. A. Rivkin was parially supported by the UNIBZ project WineID.

References

[1] Aceto L, Achilleos A, Francalanza A, Ingólfsdóttir A, Lehtinen K. Adventures in monitorability: from branching to linear time and back again. Proc ACM Program Lang 2019;3(POPL):52:1–29.

[2] Alman A, Maggi FM, Montali M, Patrizi F, Rivkin A. Multi-model monitoring framework for hybrid process specifications. In: Advanced information systems engineering - 34th international conference, CAiSE 2022, Leuven, Belgium, June 6-10, 2022, Proceedings. Lecture notes in computer science, vol. 13295, Springer; 2022, p. 319–35.

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

[4] Anselma L, Piovesan L, Terenziani P. Temporal detection and analysis of guideline interactions. Artif Intell Med 2017;76:40–62.

[5] Bansal S, Li Y, Tabajara LM, Vardi MY. Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications. In: AAAI. AAAI Press; 2020, p. 9766–74.

[6] Bauer A, Leucker M, Schallhart C. Comparing LTL semantics for runtime verification. Logic Comput 2010.

[7] Beccuti M, Bottrighi A, Franceschinis G, Montani S, Terenziani P. Modeling clinical guidelines through Petri nets. In: Combi C, Shahar Y, Abu-Hanna A, editors. Proceedings of the 12th conference on artificial intelligence in medicine. LNCS, vol. 5651, 2009, p. 61–70.

[8] Bottrighi A, Chesani F, Mello P, Molino G, Montali M, Montani S, et al. A hybrid approach to clinical guideline and to basic medical knowledge conformance. In: Combi C, Shahar Y, Abu-Hanna A, editors. Proceedings of the 12th conference on artificial intelligence in medicine. Lecture notes in computer science, vol. 5651, 2009, p. 91–5.

[9] Bottrighi A, Chesani F, Mello P, Montali M, Montani S, Terenziani P. Conformance checking of executed clinical guidelines in presence of basic medical knowledge. In: Proc. of BPM workshops. LNBIP, vol. 100, Springer; 2011, p. 200–11.

[10] Bottrighi A, Piovesan L, Terenziani P. A general framework for the distributed management of exceptions and comorbidities. In: Zwiggelaar R, Gamboa H, Fred ALN, i Badia SB, editors. Proceedings of the 11th international joint conference on biomedical engineering systems and technologies (BIOSTEC 2018) - Volume 5: HEALTHINF, Funchal, Madeira, Portugal, January 19-21, 2018. SciTePress; 2018, p. 66–76. http://dx.doi.org/10.5220/0006552800660076.

[11] Burattin A, Maggi FM, Sperduti A. Conformance checking based on multi-perspective declarative process models. Expert Syst Appl 2016;65:194–211.

[12] Calvanese D, de Giacomo G, Montali M, Patrizi F. Verification and monitoring for first-order LTL with persistence-preserving quantification over finite and infinite traces. In: Raedt LD, editor. Proceedings of the 31st international joint conference on artificial intelligence. ijcai.org; 2022, p. 2553–60.

[13] Carmona J, van Dongen BF, Solti A, Weidlich M. Conformance checking - relating processes and models. Springer; 2018.

[14] De Clercq PA, Kaiser K, Hasman A. Computer-interpretable Guideline Formalisms. In: ten Teije A, Miksch S, Lucas PJF, editors. Computer-based medical guidelines and protocols: a primer and current trends. Studies in health technology and informatics, vol. 139, IOS Press; 2008, p. 22–43.

[15] de Giacomo G, de Masellis R, Grasso M, Maggi FM, Montali M. Monitoring business metaconstraints based on LTL and LDL for finite traces. In: BPM. LNCS, vol. 8659, Springer; 2014, p. 1–17.

[16] de Giacomo G, de Masellis R, Maggi FM, Montali M. Monitoring constraints and metaconstraints with temporal logics on finite traces. ACM Trans Softw Eng Methodol 2022;31(4):68:1–44.

[17] de Giacomo G, Favorito M. Compositional approach to translate LTLf/ldlf into deterministic finite automata. In: Biundo S, Do M, Goldman R, Katz M, Yang Q, Zhuo HH, editors. Proceedings of the 31st international conference on automated planning and scheduling. AAAI Press; 2021, p. 122–30.

[18] de Giacomo G, Vardi MY. Linear temporal logic and linear dynamic logic on finite traces. In: IJCAI. IJCAI/AAAI; 2013, p. 854–60.

[19] De Leoni M, Felli P, Montali M. Strategy synthesis for data-aware dynamic systems with multiple actors. In: KR. 2020, p. 315–25.

[20] de Masellis R, Maggi FM, Montali M. Monitoring data-aware business constraints with finite state automata. In: ICSSP. ACM; 2014, p. 134–43.

[21] de Masellis R, Maggi FM, Montali M. Monitoring data-aware business constraints with finite state automata. In: ICSSP. ACM; 2014, p. 134–43.

[22] De Smedt J, De Weerdt J, Vanthienen J. Fusion Miner: Process discovery for mixed-paradigm models. Decis Support Syst 2015;77:123–36.

[23] Di Francescomarino C, Ghidini C. Predictive process monitoring. In: van der Aalst WMP, Carmona J, editors. Process mining handbook. LNBIP, vol. 448, Springer; 2022, p. 320–46.

[24] van Dongen BF, De Smedt J, Di Ciccio C, Mendling J. Conformance checking of mixed-paradigm process models. Inf Syst 2021;102.

[25] Felli P, De Leoni M, Montali M. Soundness verification of data-aware process models with variable-to-variable conditions. Fundam Inform 2021;182(1):1–29.

[26] Felli P, Montali M, Patrizi F, Winkler S. Monitoring arithmetic temporal properties on finite traces. In: Proceedings of the 37th AAAI conference on artificial intelligence. AAAI Press; 2023, (in press).

[27] Felli P, Montali M, Winkler S. Linear-time verification of data-aware dynamic systems with arithmetic. In: Proceedings of the 36th AAAI conference on artificial intelligence. AAAI Press; 2022, p. 5642–50.

[28] Francalanza A, Aceto L, Ingólfsdóttir A. Monitorability for the Hennessy-Milner logic with recursion. Formal Methods Syst Des 2017;51(1):87–116.

[29] Genc S, Lafortune S. Distributed diagnosis of discrete-event systems using Petri nets. In: van der Aalst WMP, Best E, editors. Applications and theory of petri nets 2003. Berlin, Heidelberg: Springer Berlin Heidelberg; 2003, p. 316–36.

[30] Genc S, Lafortune S. Distributed diagnosis of place-bordered Petri nets. IEEE Trans Autom Sci Eng 2007;4(2):206–19. http://dx.doi.org/10.1109/TASE.2006.879916.

[31] Grando MA, Glasspool D, Fox J. Petri nets as a formalism for comparing expressiveness of workflow-based clinical guideline languages. In: Ardagna D, Mecella M, Yang J, editors. Proceedings of the business process management workshops. LNBIP, vol. 17, Springer; 2008, p. 348–60.

[32] Hadjicostis CN, Verghese GC. Monitoring discrete event systems using Petri net embeddings. In: Donatelli S, Kleijn J, editors. Application and theory of petri nets 1999. Berlin, Heidelberg: Springer Berlin Heidelberg; 1999, p. 188–207.

[33] Hopcroft JE, Motwani R, Ullman JD. Introduction to automata theory, languages, and computation. Pearson international edition, third ed.. Addison-Wesley; 2007.

[34] Jalali A, Maggi FM, Reijers HA. A hybrid approach for aspect-oriented business process modeling. J Softw Evol Process 2018;30(8).

[35] de Leoni M, Felli P, Montali M. A holistic approach for soundness verification of decision-aware process models. In: ER. LNCS, vol. 11157, Springer; 2018, p. 219–35.

[36] Leucker M, Schallhart C. A brief account of runtime verification. J Log Algebr Methods Program 2009;78(5):293–303. http://dx.doi.org/10.1016/j.jlap.2008.08.004.

[37] Li J, Zhang L, Zhu S, Pu G, Vardi MY, He J. An explicit transition system construction approach to LTL satisfiability checking. Form Aspects Comput 2018;30(2):193–217.

[38] Ly LT, Maggi FM, Montali M, Rinderle-Ma S, van der Aalst WMP. Compliance monitoring in business processes: Functionalities, application, and tool-support. Inf Syst 2015;54:209–34.

[39] Maggi FM, Montali M, Bhat U. Compliance monitoring of multi-perspective declarative process models. In: EDOC. IEEE; 2019, p. 151–60.

[40] Maggi FM, Montali M, Westergaard M, van der Aalst WMP. Monitoring business constraints with linear temporal logic: An approach based on colored automata. In: BPM. LNCS, vol. 6896, Springer; 2011, p. 132–47.

[41] Maggi FM, Slaats T, Reijers HA. The automated discovery of hybrid processes. In: BPM. LNCS, vol. 8659, Springer; 2014, p. 392–9.

[42] Maggi FM, Westergaard M, Montali M, van der Aalst WMP. Runtime verification of LTL-based declarative process models. In: RV. LNCS, vol. 7186, Springer; 2011, p. 131–46.

[43] Mannhardt F, de Leoni M, Reijers HA, van der Aalst WMP. Balanced multi-perspective checking of process conformance. Computing 2016;98(4):407–37.

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

[45] Mulyar N, Pesic M, van der Aalst WMP, Peleg M. Declarative and procedural approaches for modelling clinical guidelines: Addressing flexibility issues. In: ter Hofstede AHM, Benatallah B, Paik H, editors. Proceedings of the business process management workshops. LNCS, vol. 4928, Springer; 2007, p. 335–46.

[46] Murata T. Petri nets: Properties, analysis and applications. Proc IEEE 1989;77(4):541–80.

[47] Peleg M. Computer-interpretable clinical guidelines: A methodological review. J Biomed Inform 2013;46(4):744–63.

[48] Peleg M, Mulyar N, van der Aalst WMP. Pattern-based analysis of computer-interpretable guidelines: Don't forget the context. Artif Intell Med 2012;54(1):73–4.

[49] Peleg M, Tu SW, Manindroo A, Altman RB. Modeling and analyzing biomedical processes using workflow/Petri net models and tools. In: Fieschi M, Coiera EW, Li YJ, editors. Proceedings of the 11th world congress on medical informatics. Studies in health technology and informatics, vol. 107, IOS Press; 2004, p. 74–8.

[50] Pesic M, Schonenberg H, van der Aalst WMP. DECLARE: full support for loosely-structured processes. In: EDOC. IEEE Computer Society; 2007, p. 287–300.

[51] Piovesan L, Terenziani P, Dupré DT. Temporal conformance analysis and explanation on comorbid patients. In: HEALTHINF. SciTePress; 2018, p. 17–26.

[52] Piovesan L, Terenziani P, Dupré DT. Conformance analysis for comorbid patients in answer set programming. J Biomed Inform 2020;103:103377. http://dx.doi.org/10.1016/j.jbi.2020.103377.

[53] Piovesan L, Terenziani P, Molino G. GLARE-SSCPM: an intelligent system to support the treatment of comorbid patients. IEEE Intell Syst 2018;33(6):37–46. http://dx.doi.org/10.1109/MIS.2018.2886697.

[54] Pla A, Gay P, Meléndez J, López B. Petri net-based process monitoring: a workflow management system for process modelling and monitoring. J Intell Manuf 2014;25(3):539–54.

[55] Prock J. A new technique for fault detection using Petri nets. Automatica 1991;27(2):239–45.

[56] Quaglini S, Stefanelli M, Lanzola G, Caporusso V, Panzarasa S. Flexible guideline-based patient careflow systems. Artif Intell Med 2001;22(1):65–80. http://dx.doi.org/10.1016/S0933-3657(00)00100-7.

[57] Reichert M, Weber B. Enabling flexibility in process-aware information systems - challenges, methods, technologies. Springer; 2012.

[58] Reuter C, Dadam P, Rudolph S, Deiters W, Trillsch S. Guarded process spaces (GPS): A navigation system towards creation and dynamic change of healthcare processes from the end-user's perspective. In: Daniel F, Barkaoui K, Dustdar S, editors. Proceedings of the business process management workshops. LNBIP, vol. 100, Springer; 2011, p. 237–48.

[59] Sadiq SW, Orlowska ME, Sadiq W. Specification and validation of process constraints for flexible workflows. Inf Syst 2005;30(5):349–78.

[60] Sahraoui A, Atabakhche H, Courvoisier M, Valette R. Joining Petri nets and knowledge based systems for monitoring purposes. In: Proceedings. 1987 IEEE international conference on robotics and automation, Vol. 4. 1987, p. 1160–5. http://dx.doi.org/10.1109/ROBOT.1987.1087858.

[61] Slaats T. Declarative and hybrid process discovery: Recent advances and open challenges. J Data Semant 2020;9(1):3–20.

[62] Spiotta M, Terenziani P, Theseider Dupré D. Temporal conformance analysis and explanation of clinical guidelines execution: An answer set programming approach. IEEE Trans Knowl Data Eng 2017;29(11):2567–80.

[63] Zhu S, Tabajara LM, Li J, Pu G, Vardi MY. In: Sierra C, editor. Symbolic LTLf synthesis. ijcai.org; 2017, p. 1362–9.