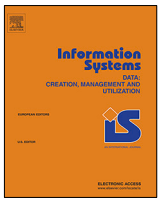
Information Systems 119 (2023) 102271

Contents lists available at ScienceDirect

Information Systems

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



A framework for modeling, executing, and monitoring hybrid multi-process specifications with bounded global–local memory

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, Estonia
[b] Faculty of Computer Science, Free University of Bozen-Bolzano, Piazza Domenicani 3, Bolzano, 39100, Italy
[c] Department of Computer, Control and Management Engineering, Sapienza University of Rome, Via Ariosto 25, Rome, 00185, Italy
[d] Department of Applied Mathematics and Computer Science, Technical University of Denmark, Richard Petersens Plads 321, Copenhagen, 2800, Denmark

ARTICLE INFO

Article history:
Received 24 December 2022
Received in revised form 2 June 2023
Accepted 14 August 2023
Available online 1 September 2023
Recommended by Gottfried Vossen

Keywords:
Business process modeling
Business process enactment
Business process monitoring
Data Petri nets
Declarative processes
Automata-based techniques
Multi-process specifications
Hybrid processes

ABSTRACT

So far, approaches for business process modeling, enactment and monitoring have mainly been based on process specifications consisting of a single process model. This setting aptly captures monolithic scenarios from domains in which all possible behaviors can be folded into a single model. However, the same strategy cannot be applied to domains where multiple interacting (procedural) processes simultaneously work over the same objects, in the presence of additional (declarative) constraints relating activities from the same or different processes. A relevant example for this setting is that of healthcare, where co-morbid patients may be subject to multiple clinical pathways at once, in the presence of additional, general constraints capturing basic medical knowledge. To fill this gap, we have previously presented the M3 Framework and an accompanying monitoring technique, which allows for a hybrid representation of a process using both procedural and declarative models, and supports the modular creation of multi-process specifications where domain experts can focus on specific procedures and domain constraints without being forced to merge them into one single specification. In this paper, we make significant extensions to this framework, allowing us to go from simple toy examples towards addressing practical real-life scenarios. We achieve this by introducing a richer form of integration between the interacting process components, in particular supporting asynchronous and synchronous activities that may operate over local and global (shared) data variables. This is framed by a discussion of the business meaning of these concepts, the introduction of the corresponding modeling patterns, and the application of our approach to real-life business processes, the latter being the driving-force behind this paper.

1. Introduction

A business process specification is a fundamental instrument in Business Process Management (BPM), in particular constituting the main medium for monitoring, enacting, and ultimately improving business processes. Such specifications are commonly created in the form of procedural or declarative process models, depending on the scenario at hand. In general, procedural models are more suitable for relatively structured processes (e.g., an automated manufacturing line), while declarative models are more appropriate to represent flexible and knowledge-intensive [1,2] processes (e.g., the management of a natural disaster).

The majority of existing process modeling approaches rely on the assumption that the full knowledge-base (control-flow, decision rules, temporal aspects, etc.) required for executing each possible process instance can be embedded into a single process specification — possibly taking advantage of modular constructs such as sub-processes. This falls short in all scenarios where:

- the description of the overall behavior of interest cannot be straitjacketed into a single, monolithic process, but is instead best described by considering that a single execution involves, at once, multiple, distinct (possibly independently defined) processes;
- one or multiple interacting processes operate under the effect of additional (declarative) constraints expressing background knowledge on their mutual interplay, possibly involving also additional activities and further behavior that is best described at the declarative level.

* Corresponding authors.
E-mail addresses: anti.alman@ut.ee (A. Alman), maggi@inf.unibz.it (F.M. Maggi), montali@inf.unibz.it (M. Montali), patrizi@diag.uniroma1.it (F. Patrizi), ariv@dtu.dk (A. Rivkin).

https://doi.org/10.1016/j.is.2023.102271

The resulting class of processes is in a way orthogonal to the currently widely investigated class of object-centric processes (see, e.g., [3–5]). In fact, while object-centric processes typically handle the situation where multiple objects and their mutual relationships are co-evolved by a single process (thus calling for abandoning at once a single case notion), we are interested in the simultaneous application of multiple process models to the same case object, dealing with the interplay of the activities they contain, and the control-flow patterns they advocate. A prime example of this latter type of situation can be found in the medical domain, where patients with co-morbidities may be subject to multiple clinical pathways at once (consisting of various medical actions) that can be enriched with additional, context-dependent constraints (e.g., reflecting background medical knowledge [6]). The declarative component hence plays, in this picture, the role of *glue* linking together the procedural components and introducing additional mutual constraints and their simultaneous execution.

A sound and rigorous definition of the execution semantics of business process specifications in such multi-model scenarios is crucial to understand how a business process must be executed to conform to all the (procedural and declarative) components, while also considering their interplay. For example, prescribing a certain treatment, without taking into consideration the contextual and pre-existing conditions of the patient, may result in adverse effects causing a worsening of the patient's state. Ideally, these conflicts should be detected at early stages of the process execution and reported to health providers so as to avoid undesired treatment outcomes. Crucially, this cannot be satisfactorily characterized by only looking into the activities executed so far, but also calls for considering the possible future continuations of the case at hand, a problem that in BPM has been so far only studied in the context of constraint monitoring [7,8].

In our previous works [9,10], we have introduced a semantics for hybrid, multi-process specifications in which multiple (procedural and declarative) components, each equipped with local data conditions, interact only by sharing same-labeled activities. This results in a very limited form of model interplay, which falls short in capturing a variety of modeling patterns required in practice. An example, that cannot be captured in the framework defined there, is the following: the pre-surgery anesthesia activity in a medical process can appear in different process models every time a surgery is performed; however, the pre-surgery anesthesia must be executed anew for each surgery. This poses the following research question: how to lift the framework in [9,10] to a more general one that would support the fine-tuned modeling of how to handle the combined execution of process components mentioning the same activity, dealing at once with asynchronous/synchronous patterns, as well as unified/repeated executions.

This first limitation also relates to a second one of the approach in [9,10]: namely that the scope of data objects is always local to single process components. This prevents the possibility for such components to share data and, in turn, use such shared data to mutually influence each other. A simple yet effective example illustrating the need for such shared data is related to using data to indicate that the presence of the same activity in multiple process components should be handled in an *execute only once* way, that is, executing it one time and skipping the other expected occurrences (e.g., skipping a medical test on a patient if the corresponding information is already known, and specifically if the test has been already conducted as per request of another process component).

These two limitations alone mean that our earlier semantics [9,10], while suitable to demonstrate the overall idea of hybrid multi-process specifications, is not sufficient for many practical scenarios. In this paper, we specifically focus on overcoming these limitations, *introducing a multi-model framework equipped with global/local activities and variables, and demonstrating how these concepts now enable us to tackle real-life business processes*.

We discuss the business meaning of these concepts, introduce the corresponding modeling patterns, and apply our approach to real-life business processes. In addition, we extend our semantics and formalism accordingly, thus providing a solid foundation not only for execution and monitoring of hybrid multi-process specifications (both of which we address in this paper), but also for additional process mining tasks such as model simulation and conformance checking.

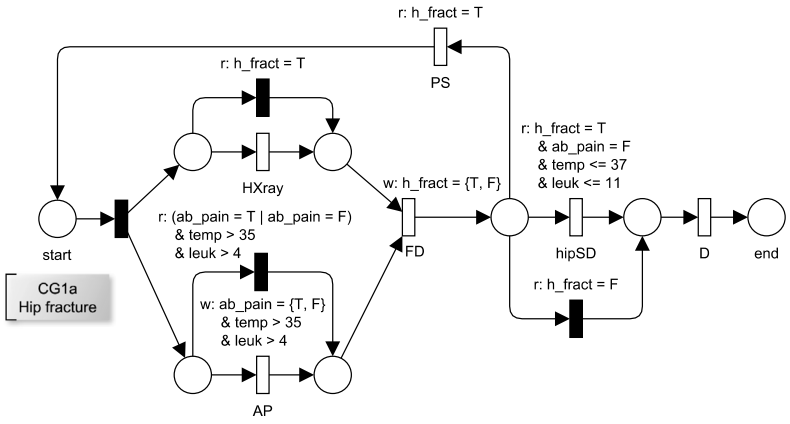
The rest of this paper is structured as follows. Section 2 provides an example scenario where hybrid, multi-process specifications are needed, and discusses the limitations of our earlier monitoring semantics within the business context of that scenario. Section 3 introduces our multi-process framework and its lifecycle. Section 4 discusses the behavior of global/local activities and variables, and the accompanying modeling patterns. The core Section 5 describes the execution semantics of hybrid, multi-process specifications based on data Petri nets and data-aware DECLARE. Section 6 adapts the automata-based technique introduced in [9] to the more sophisticated setting tackled in this paper, showing how it can be used for monitoring and for detecting conflicts, which is also instrumental for enactment. Sections 7–9 conclude the paper by showing how our framework can be applied in real settings, by describing related work, and by spelling out directions for future work respectively.

2. Example scenario

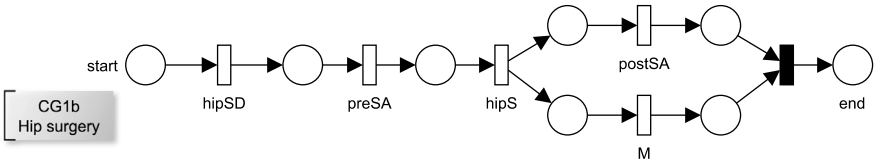
Compared to our earlier works [9,10], we present here a more elaborate example scenario, which allows us to better showcase the advancements made in this paper. As in [9], we draw inspiration from [11] that shows interactions between two seemingly disjoint clinical guidelines (CGs) and basic medical knowledge (BMK). We use the same process model from [11] for a suspected hip fracture (with some modifications), but this time combining it with a process model for a suspected appendicitis. The latter was chosen over the chest infection one used in [11] in order to demonstrate more complex forms of process interplay. In practice, our example scenario corresponds to a case where a patient presents, at the same time, a potential hip fracture, and a suspicion of appendicitis. First, we describe the process models and abbreviations for this scenario. Then, we follow up with a brief discussion on the types of interplay between these models that will be used to introduce global/local activities and variables.

2.1. Process models

The process of handling a suspected hip fracture (CG1) begins with the diagnostic activities shown in Fig. 1(a). More specifically, a hip X-ray (HXray) is executed in parallel with the initial assessment (AP) of the patient's condition. During AP, the patient is checked for abdominal pain, and the patient's temperature and leukocytes level are measured. HXray produces one or more images of the hip (not represented in the model), which are then used as input for formulating a diagnosis (FD). If no hip fracture is detected, then, based on this process model, the patient may be discharged (D). If, instead, a hip fracture is detected, then the doctor may either postpone the surgery (PS) or make a decision to perform the surgery (hipSD). PS only assumes that a hip fracture was detected and allows (but does not require) the doctor to repeat HXray and AP. Note that AP must be repeated when hipSD is needed and the patient either has abdominal pain,



(a) Diagnosis of suspected hip fracture, potentially leading to a hip surgery.



(b) Sub-process for a hip surgery and the corresponding followup care.

Fig. 1. Data Petri net representation of the process of handling a suspected hip fracture.

or temperature higher than 37, or leukocytes level higher than 11 (values of which are determined by AP).

The hip surgery sub-process shown in Fig. 1(b) is intended to be triggered by the execution of hipSD.[1] In fact, hipSD represents, in this case, exactly the same activity in both the main process model (Fig. 1(a)) and also in the hip surgery sub-process. In the overall process, there will be at most one execution of hipSD, which would be followed by administering pre-surgery anesthesia (preSA), and then by performing the hip surgery (hipS) itself. After the surgery, the patient is prescribed with both post-surgery analgesia (postSA) for pain relief and physiotherapy for mobilizing the hip (M).

The process of handling a suspected appendicitis (CG2) begins with the diagnostic activities shown in Fig. 2(a). First, an initial assessment (AP) of the patient's condition is executed in parallel with a computerized tomography scan (CT) of the abdomen. The results of AP and CT (the latter not being modeled) are then used as input for formulating a diagnosis (FD), after which the doctor may start amoxicillin therapy (AT). AT can, in some cases, lead to a successful treatment, which would then lead to discharge (D) the patient. If, however, AT is unsuccessful then a decision to perform an appendectomy (appSD) must be made instead. It is also allowed to skip AT for any reason (e.g., an allergy), and to discharge the patient if appendicitis is not diagnosed.

Analogously to CG1, the appendectomy sub-process shown in Fig. 2(b) is intended to be triggered by the execution of appSD, and appSD represents exactly the same activity in both the main process model (Fig. 2(a)) and the sub-process (Fig. 2(b)). The appendectomy sub-process also follows the same overall structure as the hip surgery. First, the pre-surgery anesthesia (preSA) is delivered, then the appendectomy itself is performed (appS), followed by post-surgery analgesia (postSA) for pain relief and (if possible) amoxicillin therapy (AT).

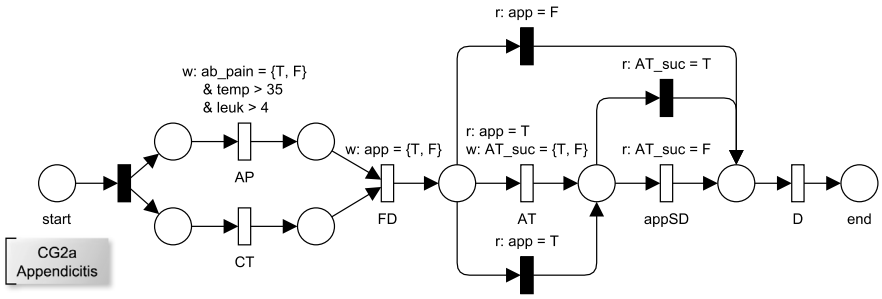
To make the scenario complete, we must also introduce at least one additional rule for connecting CG1 with CG2. In this case, we specify that if both abdominal pain (*ab_pain*) and leukocytes level (*leuk*) above 11 are detected during the activity AP in CG1 (i.e., there is reason to suspect appendicitis), then CG2 must be executed (constraint C1). Finally, to demonstrate the role of BMK in our framework, we introduce a second rule, specifying that executing preSA (in any process model) requires any ongoing amoxicillin therapy to be stopped (stopAT) in order to avoid potential complications during surgeries (constraint C2). Both these rules can be modeled using constraints expressed with the semantics introduced in Section 5.

2.2. Interplay from the business perspective

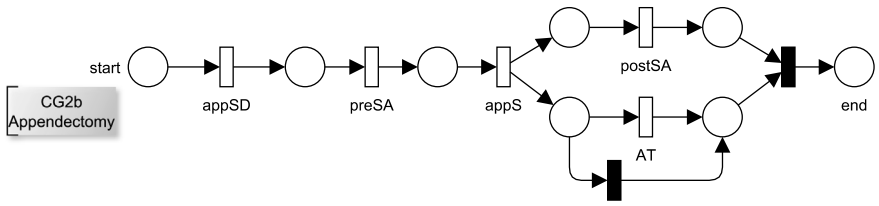
There are multiple types of interplay in the given example scenario. In this section, we give an overview of these from the business perspective. We start with the ones that fit our earlier monitoring semantics [9] precisely. The most obvious of these is the same-labeled activity D which appears in both CGs. In our earlier semantics it corresponds to the rule "if multiple specifications require the same event to occur, then its single occurrence should satisfy all of them", or in other words, each activity always synchronously progresses all process specifications containing that activity (potentially causing a violation of specifications that do not allow the given activity at the given point in time to be executed). This matches the intuition of discharging a patient from a hospital quite well, assuming that the patient will, as a result, physically leave the hospital.

A similar argument can also be made for activity FD, if we assume that all required diagnosis activities (from any CG) must be completed before a full diagnosis and treatment plan can be

[1] The term "sub-process" is used here mainly to assist in the intuitive understanding of the example scenario. In fact, sub-processes are not an explicit part of our execution semantics. Instead, they are treated as independently defined process specifications, which are then integrated into the global process behavior as all other process specifications.



(a) Diagnosis and treatment of suspected appendicitis, potentially leading to an appendectomy.



(b) Sub-process for an appendectomy and the corresponding followup care.

Fig. 2. Data Petri net representation of the process of handling a suspected appendicitis.

compiled. Of course this does not hold in all cases (e.g., when an immediate treatment is required), but we consider a synchronous progression for activity FD reasonable in the given example scenario. We also leverage this type of interplay to trigger the execution of a "sub-process" (activities hipSD and appSD), although a constraint-based approach (similar to the one adopted with the definition of constraint C1) could be used instead.

However, the above semantics does not fit the other same-labeled activities. For example, even if we assume that the pre-surgery anesthesia (preSA) is exactly the same in both CGs, then we still cannot assume that executing preSA for a hip surgery means (synchronously) executing it also for an appendectomy. Instead, activity preSA would always have to be (re)executed before each surgery, and the same also applies to activity postSA. Another example is activity AT, which appears twice in CG2 — first, to treat appendicitis and, second, to assist the post-surgery recovery. And again, we cannot assume that executing AT in the initial treatment makes it not necessary for the post-surgery recovery, and we certainly cannot assume that both happen synchronously.

Activity AP appears in both CGs, and provides values for the same variables (*ab_pain*, *temp*, and *leuk*). However, CG1 allows AP to be skipped, while CG2 does not. This is due to the fact that, when considering CG1 in isolation, postponing the surgery (PS) leads to a state in which values from the previous execution of AP could be reused, while CG2, again in isolation, does not allow for any similar reuse. However, if we consider the combination of these CGs together with constraint C1, for globally valid executions, activity AP in CG2 is executed after it was already executed in CG1. In this case, instead of requiring a second execution of AP (which would be unavoidable in our earlier monitoring semantics [9]), we should allow the second execution of AP to be skipped, and the values already determined in CG1 to be reused.

This, in turn, requires using global (shared) variables. On the one hand, global variables can be used to capture data (e.g., the aforementioned values) about the same object, which is concurrently flowing through multiple processes. On the other hand, global variables can be used for more sophisticated types of interplay, e.g., for the definition of global "control variables". For example, it is reasonable to assume that two surgeries cannot be ongoing at the same time. In the given scenario, this could be achieved by defining an additional global boolean variable (*inS*) both to track if the patient is in surgery or not, and to disable (or enable) activities hipSD and appSD, accordingly.

The above discussion is summarized in Table 1 as an overview of *global activities*, *local activities*, *global variables*, and *local variables*, specifying their meanings, primary use cases, and examples. Practical application of these concepts is further elaborated on in Section 7, after we have introduced the surrounding Multi-Process Framework (Section 3), the prerequisite modeling patterns (Section 4), and the corresponding execution semantics (Section 5.5).

3. The framework

Addressing scenarios like the one described in Section 2 requires not only to consider potential interactions between various combinations of models, but also a framework for handling these combinations. For this purpose, we iterate upon our M3 Framework presented in [9], uplifting it from a monitoring specific framework to a more general purpose framework for eliciting, managing, and executing hybrid process specifications. The main changes (Fig. 3) are (1) changing the Monitoring Phase to Execution Phase, and (2) adding activity and variable scope definitions in the specification repository. The main steps within the phases of the framework are largely unchanged compared to [9].

The *elicitation phase* of the 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 [12]. 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

Table 1
Overview of *global activities*, *local activities*, *global variables*, and *local variables*.

	Common use cases	Example
Global Activity - Considered equivalent to all other same-labeled activities, and always executed concurrently in all process models containing it.	Milestones representing the overall process moving from one stage to the next.	FD and D in the example scenario.
	Synchronous connections between process models.	hipSD and appSD in the example scenario.
Local Activity - Considered distinct from all other same-labeled activities, but may be subject to global constraints.	Activities that have distinct names across all models.	HXray, CT, etc. in the example scenario.
	Same-labeled activities that must always be executed as distinct across all process models.	AP, AT, etc. in the example scenario.
Global Variable - Variable that is shared, both for reading and writing, by every process model referring to that variable by name.	Shared information that characterizes the objects involved in the ongoing case.	*ab_pain*, *temp*, and *leuk* in the example scenario.
	Control variables for enforcing specific types of interplay.	*inS* for handling interactions between surgeries.
Local Variable - Variable that is both distinct and also private to the process model defining that variable.	Private information that characterizes the objects involved in the ongoing case.	*h_fract*, *app*, and *AT_suc* in the example scenario.
	Control variables used to define the behavior within a single process model.	*i* for counting the number of iterations.

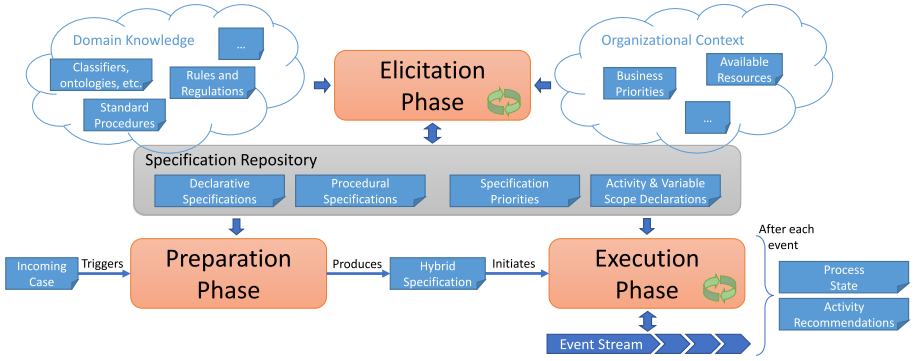


Fig. 3. Conceptual overview of the framework.

provide guidance to the user in resolving any potential conflicts. Individual models can contain both global and local activities and variables, with the default scopes being defined (or modified) within this phase on a per-activity and per-variable basis.

The *preparation phase* of the 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 and provided as input for the subsequent execution phase. This hybrid specification will encompass the combined behavior of all selected specifications, the corresponding specification priorities, activity and variable scope definitions, and, if required, additional case-specific modifications and constraints. 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 *execution phase* of the framework is envisioned as an ongoing case-specific phase, covering the entire duration of the case being monitored and/or executed. During this phase, the state of the process and the set of next recommended actions (with data payloads) are updated after the occurrence of each event, thus providing guidance towards the successful completion of the case. This phase has a conceptual two-way relation with the event stream of the ongoing case. In the previous monitoring-focused iteration, it represented the fact that recommendations (if followed) will indirectly effect the event stream through actions of the process workers. In the new framework, this connection becomes more direct as, at the heart of this phase, there is now an actual process execution engine of the hybrid specification.

4. Interplay and modeling patterns

In Section 2.2, we took the business perspective to demonstrate why, in a multi-process setting, some same-labeled activities and attributes should be considered global, while others should be considered local. In this section, we dive deeper into these differences from the modeling and enactment perspective. We use relatively small toy examples to demonstrate the behavior of *global activities*, *local activities*, *global variables*, and *local variables* in procedural process specifications.[2] We also discuss additional modeling patterns, which we believe would be commonly used in such a setting.

We rely on Petri nets and Data Petri nets, formally introduced later in Section 5.3. Here, it is sufficient to know that a Petri net can be used to model the control flow of a business process through a system of places and transitions. In this system, transitions represent activities of the process, while places track the current state of the process. Data Petri nets extend this by allowing to add read and write guards to transitions, which need to be satisfied in order to execute the corresponding activities. In our setting, a write guard checks and stores values from the activity data payload (i.e., allows activities with a certain data payload to be executed), while a read guard checks the latest values stored by write guards to decide if a corresponding activity can be executed.

[2] The declarative process specifications of the framework continue to behave as in our previous work, however event provenance information is assumed to be specified in the event payload so that events of specific models can be referred to by defining filters on this information.

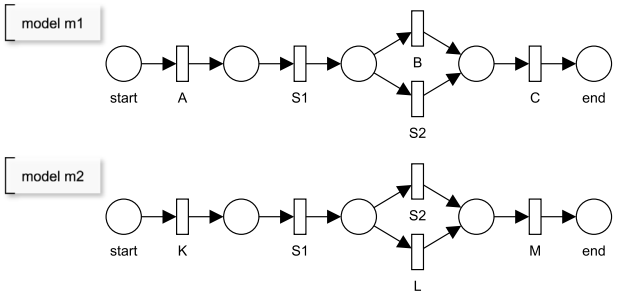


Fig. 4. Example Petri nets, with same-labeled activities S1 and S2.

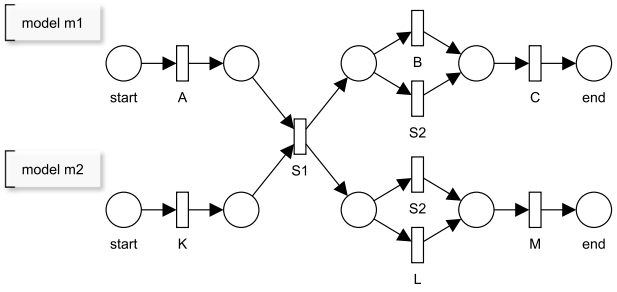


Fig. 5. Petri net representing the combined behavior of *m1* and *m2*, when S1 is defined as a global activity and S2 as a local activity.

4.1. Global and local activities in Petri nets

Global activities function exactly the same as same-labeled activities in [9,10], i.e., executing a global activity means concurrently progressing all nets which contain that activity (possibly violating some of them). In contrast, a local activity always behaves as if it had a distinct label from all other same-labeled activities, i.e., executing a local activity progresses only the net which produced that specific event.

To exemplify this difference further, consider the Petri nets presented in Fig. 4. When taken individually, the first Petri net (*m1*) allows for executions: A, S1, B, C and A, S1, S2, C, while the second Petri net (*m2*) allows for executions: K, S1, S2, M and K, S1, L, M. Meanwhile, the combined behavior of *m1* and *m2* depends on whether the same-labeled activities S1 and S2 are defined as global or local activities.

If both S1 and S2 are defined as local activities, then the combined behavior of *m1* and *m2* will allow for any interleaving of valid executions of both models. For example, the execution A, K, S1, S1, B, L, C, M would be valid, as it does not violate the control flow of either model, and also reaches the end state of both. Notice that activity S1 occurs twice in the given example, first progressing one of the models and then the other. To decide which model to progress first, all local activities are assumed to carry a data value, denoting the provenance of the specific activity instance. An invalid execution would be, instead, the execution A, K, S1, B, C, M, because it contains a single instance of S1, but also because it is missing either S2 or L, one of which is required by *m2* before executing M.

By defining S1 as a global activity, we get the combined behavior presented in Fig. 5. In this case, S1 basically becomes a shared activity between *m1* and *m2*, represented by a single shared transition. This has the obvious impact of requiring only one occurrence of activity S1, which will now progress both models concurrently. However, this also means that the first three

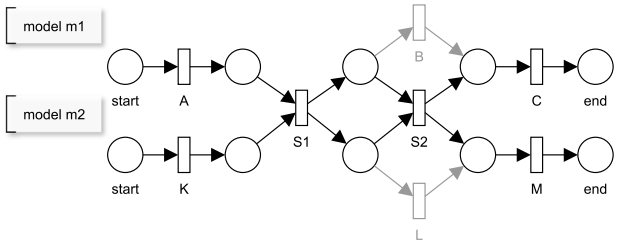


Fig. 6. Petri net representing the combined behavior of *m1* and *m2*, when both S1 and S2 are defined as global activities. Activities B and L are greyed out because, in this case, they cannot occur in any valid execution.

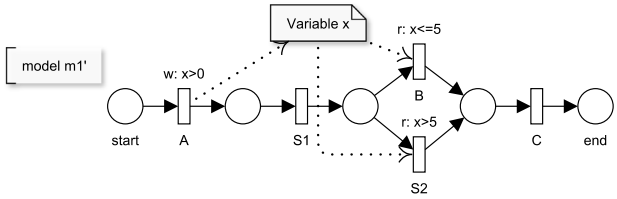


Fig. 7. A Data Petri net with a local variable *x*.

activities of any globally valid execution must now be either A, K, S1 or K, A, S1.[3]

An additional peculiarity of global activities can be shown by also defining S2 as a global activity. This results in prohibiting activity B in model *m1* and activity L in model *m2*, thus deciding in advance the otherwise free-choice points after the S1 in both models. The corresponding combined behavior matches our earlier monitoring semantics [9] and can be represented as the Petri net shown in Fig. 6. This combined behavior implies, on the one hand, that some invalid executions can be automatically detected and handled in advance. On the other hand, it may result in unintended side-effects, which may not be obvious at first glance, especially for more complex models.

In general, global activities behave as synchronization points. Any globally valid execution requires all the models sharing the same global activity to execute that activity synchronously. This means that each model must, at the same time, be in a state which allows that (global) activity to be executed.

4.2. Global and local variables in Petri nets

Differently from the previous section, we begin here by first discussing local variables, as they function exactly the same as all variables in our earlier monitoring semantics [9,10] (and in Data Petri nets in general). We then proceed with global variables and the interaction between global activities and global variables.

To explain local variables, we modify model *m1* from the previous section by adding a write guard (condition $x > 0$) on A, a read guard (condition $x <= 5$) on B, and another read guard (condition $x > 5$) on S2. Based on the resulting model *m1'* (Fig. 7), when A occurs, its data payload is checked for the attribute *x*. If the corresponding value in the activity payload does not match the condition $x > 0$ (or is missing), then A is considered a violation of the model. Otherwise, the value of *x* is assigned to the corresponding variable of the net. The process execution then continues with S1, after which it reaches a decision point, which is now governed by two guards. These guards specify that B

[3] Even in the case of global activities, we still track both global and local execution states. For example, the execution A, S1, K, B, C would result in a global violation, a local satisfaction of *m1*, and a local violation of *m2*.

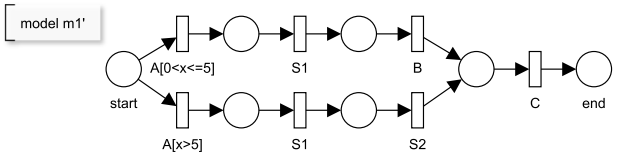


Fig. 8. An alternative version of the Data Petri net *m1'*, which is equivalent in terms of allowed executions and makes the behavior of local variables explicit.

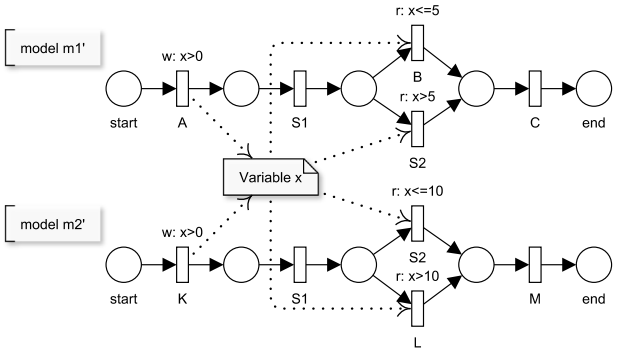


Fig. 9. Data Petri nets *m1'* and *m2'* with a global variable *x*.

should be executed if $x <= 5$, and S2 should be executed if $x > 5$. However, as these are read guards, the check is not performed on the activity payload, but, instead, on the last assignment of variable *x*, which now leads to either B or S2 to be executed next.

Notice that, the decision between B and S2 is actually made when executing A. We can take advantage of that fact by basically embedding the variable value into the structure of the net itself. This would give us the model shown in Fig. 8, which is still equivalent in terms of both possible executions and violations. We showcase this alternative model here primarily to assist in understanding how read and write guards function, however, a similar transformation is also used when constructing the automaton of a Data Petri net in Section 5.

Defining *x* as a global variable, means that it will be shared among all nets referring to that variable. To explain this in more detail, we use the model *m1'* and a similarly modified model *m2'*, and we define *x* as a global variable, resulting in the combined behavior summarized in Fig. 9. Compared to the previous example, the decision between B and S2 is now no longer determined when A is executed. Instead, K could be executed after activities A or S1, potentially changing the assigned value of the (now global) variable *x*. For example, a valid execution could now be A[x=1], K[x=10], S1, S1, S2, S2, M, C. In addition, this example has the side-effect that B and L can only occur in the same execution if B occurs before K or if L occurs before A. This is because the corresponding read guards of B and L are mutually exclusive and only A and K can provide suitable values for these guards.

Finally, we elaborate on the interaction between global activities and global variables by defining S2 in our previous example as a global activity (Fig. 10). As discussed in Section 4.1, activities B and L become prohibited for globally valid executions. In addition, when a global activity has (read or write) guard conditions referring to global variables, then any globally valid execution must match the union of these conditions. In this case, the original conditions were $x > 5$ and $x <= 10$, resulting in the combined condition $5 < x <= 10$. We note that, if a global violation occurs in this case, then the locally non-violated nets will still progress, allowing the process execution to continue.

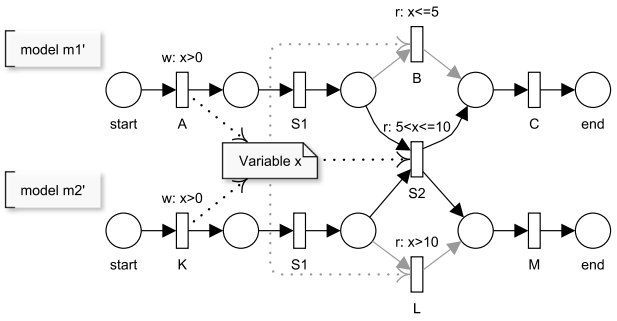


Fig. 10. Data Petri nets *m1'* and *m2'* with a global variable *x* and a global activity S2. Activities B and L are greyed out because, in this case, they cannot occur in any valid execution. The read guard on S2 is updated by combining the corresponding guards in *m1'* and *m2'*.

4.3. Additional modeling patterns

The introduction of global variables enables multiple modeling patterns for specifying the interplay between process models. In the following, we introduce some of these, which will be useful both from the perspective of modeling, and to provide further insights into how our multi-process setting works. In the following subsections, we refer to the example models as primary and secondary: the primary model is the one that either enables or imposes some behavior on the secondary model.

4.3.1. Data-based activity skipping

Global variables allow us to rather easily support, not only the reuse of some values across multiple models, but also to avoid redundant activity executions for (re)determining these values. More specifically, we can use global variables to store these values, and then add additional silent transitions (i.e., transitions that can be taken without performing any activity), with corresponding guards, to skip the potentially redundant activities. An example of this is given in Fig. 11, where L in the secondary model can be skipped if the value of variable *x* is already provided by A in the primary model. An analogous silent transition could also be added for skipping A in the primary model. In Section 7.2, we apply this pattern to solve a similar redundancy of the initial assessment of the patient in our example scenario.

4.3.2. Blocking read guards

With global variables, we can relax the assumption that an individual Data Petri net must provide values for every variable that it may need to read. Instead, these values could be provided by other nets, thus even allowing for nets, which do not contain any write guards nor initial variable assignments. In addition to sharing and reusing global values, this also enables an additional form of synchronization besides global activities. More specifically, read and write guards can be defined such that the execution of one net is effectively blocked until another net provides some value. This is demonstrated in Fig. 12 where the execution of the secondary model can start only after executing activity B in the primary model.

4.3.3. Blocking Petri net regions

In addition to sharing data values between models, it is also possible to use global variables to share values that are explicitly intended to modify the control flow of the combined process.[4] This allows the modeler to create models where a region of one

[4] Our current monitor implementation assumes that such values are still carried in the activity data payloads.

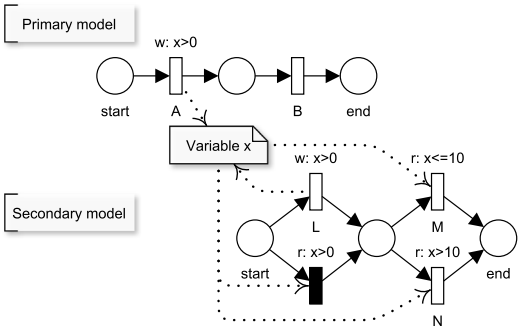


Fig. 11. An example where activity L in the secondary model can be skipped if a valid value of variable *x* is already provided by activity A in the primary model.

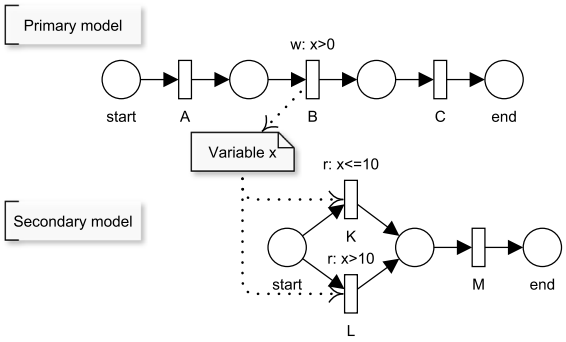


Fig. 12. An example where the decision between activities K and L in the secondary model can only be made after executing B in the primary model.

model is effectively disabled while activities from a region of another model are being executed. For example, in Fig. 13, activities L and M in the secondary model cannot be executed before the execution of the primary model ends. In Section 7.2, we apply a variation of this pattern to prohibit multiple concurrent surgeries in our example scenario.

4.3.4. Forced control flow restart

Another example of using global variables to modify the control flow of the combined process can be seen in Fig. 14. In this example, a global variable is used to force the ongoing execution of the secondary model to restart if activity B is executed in the primary model. However, note that if the secondary model in the given example is fully executed before executing activity B, no restart of the secondary model is required.

4.3.5. Optional data Petri nets

In some cases, like in the example scenario in Section 2, it is not expected for all procedural specifications to be mandatory. In the simple case, the optionality of a procedural specification can be obtained by adding a silent transition from the beginning of the "optional" model to the final state of the model. An example of this is shown in Fig. 15, where the secondary model is modeled as optional, and a "response" constraint is used to specify that, if activity B occurs, activity K must eventually occur after B, thus moving the secondary model into a state that requires to fully execute it.

However, this pattern works only if the "optional" model has no global activities. As shown in Section 7.2, when a model contains global activities, a case-by-case approach must be taken to make sure that the global activities will always be executed correctly.

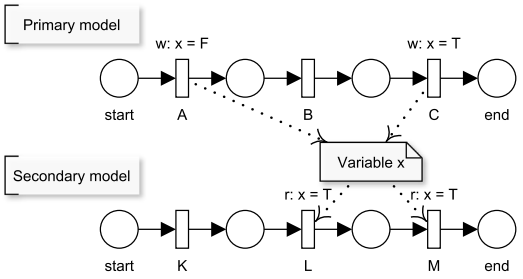


Fig. 13. An example where activities K and L of the secondary model are not allowed to be executed while the primary model is being executed.

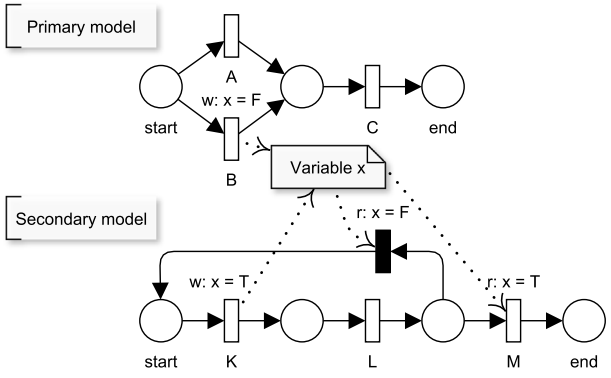


Fig. 14. An example where choosing to execute activity B in the primary model forces a restart of the secondary model after activity L.

5. Formalizing hybrid multi-process specifications

In this section, we discuss the representations of declarative and procedural data-aware process specifications, touching not only upon the formalisms for their representation, but also discussing the types of events they have to process. We then show how such specifications can be integrated into a single model, allowing for synchronous inter-model operations as well as global and local memory management.

The process specifications come from two separate (procedural and declarative) data-aware modeling paradigms, and rely on two formalizations respectively provided by:

- Data Petri nets (DPNs) [13,14], where a Petri net is extended with (categorical and numerical) variables, and transitions are equipped with read–write guards over those variables;

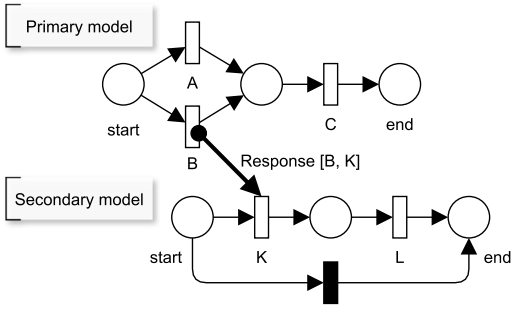


Fig. 15. An example where the secondary model is required to be executed only if activity B in the primary model is executed.

guards are of the form of so-called *interval conditions*, as they are constituted by boolean combinations of atomic conditions comparing variables against constants.
- A variant of DECLARE extended with event attributes and data-aware guards over those attributes [15,16], using the corresponding data-aware extensions of linear temporal logic over finite traces (LTL_f [17]) to formally capture its semantics, following [18]; in particular, consistently with the choice done for DPNs, we focus here on guards expressed as interval conditions, like in [13].

It is important to notice that the variant of DECLARE with interval conditions that we consider in this paper still admits an automata-based characterization grounded on traditional finite-state automata [19], thanks to the fact that interval conditions can be faithfully abstracted using a finite number of propositions, as shown in [13]. This is a crucial observation that, de facto, allows for the seamless adoption of reasoning techniques existing in the DECLARE literature [20], and that will be exploited in Section 6 for monitoring hybrid specifications obtained by merging DPNs with DECLARE constraints defined on top of them.

All in all, the composition of different procedural models and a declarative specification interconnecting them can be seen as a data-aware version of the hybrid approach introduced in [21], extended here with global/local activities/events, global/local variables, and interval conditions over such variables used in the transitions of the procedural models and in the constraints of the declarative one.

Complete, formal definitions of the aforementioned specifications and formalisms can be found in Appendix A, while the following subsections focus on the specifics of the approach presented in this paper.

5.1. Events and traces

An *activity* is a tuple (n, q, A), where n is the activity name, q is the process reference and $A = \{a_1, \ldots, a_\ell\}$ is the set of event attribute (names). As our main goal is to define a multi-process framework, we use the process reference to indicate the *provenance* of the activity, which is the specification to which the activity belongs. This also unambiguously identifies activities belonging to different specifications that share the same name, but may differ in terms of the attributes they carry, and captures situations in which different activities sharing the same name have a different "local" meaning in different specifications.

We also consider "global" activities that *(i)* are either used in multiple process specifications retaining the same meaning, and thus represent shared activities over which the process specifications need to synchronize, or *(ii)* constitute additional activities not mentioned in any process specification (that may be subject to declarative constraints). To distinguish such global activities, we fix a special process reference global, not used to identify any specification.

In the following, we fix a finite set $\mathcal{E}$ of activities, and introduce for it the set of all *activity names* $\mathcal{N}_\mathcal{E}$, the set of all *process references* $\mathcal{P}_\mathcal{E}$, and the set of all *attribute names* $\mathcal{A}_\mathcal{E}$. We call $\mathcal{E}$ *well-formed*, if it does not contain same-named activities for the same process specification, and each global activity is uniquely defined. Hereinafter, the sets of activities are assumed to be well-formed.

Example 1. Consider the medical example scenario of Section 2. Let cg1 be the name of the default process for handling patients with a hip fracture. Here are some of its activities:

- (HXray, cg1, $\emptyset$) represents a hip x-ray test executed in the context of cg1;
- (FD, cg1, {*hip_fract*}) indicates the formulation of a diagnosis for hip fracture, where *hip_fract* is a boolean attribute reporting whether a hip fracture is present or not;
- (AP, global, {*temp*, *ab_pain*, *leuk*}) is a global activity (shared with other possible process specifications) for assessing the patient's health parameters that are listed in the activity's attributes. ◁

An *event* over activity $(n, q, A) \in \mathcal{E}$ is a triple $e = (n, r, \nu)$, such that $n \in \mathcal{N}_\mathcal{E}$, $q \in \mathcal{P}_\mathcal{E}$ and $\nu : A \mapsto \mathbb{R}$ is a total function assigning to every attribute in A an actual value. 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). As usual, we call a finite sequence $\sigma = e_1 \cdots e_\ell$ of events a *trace*, where each e_i is an event over some activity in $\mathcal{E}$.

5.2. Interval conditions

As we defined above, events in our framework carry data payloads, assigning values to attributes. On top of that, we require that process specifications come with local and global data variables. It is natural to assume a language for expressing conditions over such attributes and local/global data variables. As we show later on, such conditions form the basic building blocks to express filters and updates over local/global data variables.

Using the language (inspired by the one from [13]), it is possible to define formulas whose components are atomic expressions of the form $a \odot c$, where $\odot \in \{<, =, >\}$, $c \in \mathbb{R}$ and $a \in V^\mathrm{d}$, where V^d is a generic set of data variables. Using V^d, it is possible to define conditions in a general way, abstracting away from whether they are applied to attributes or local/global variables. Moreover, boolean combinations of multiple $a \odot c$ expressions capture open and closed intervals over $\mathbb{R}$. For example, $x > 0 \wedge x \leq 5$ characterizes the fact that $x \in (0, 5]$. For this reason, we refer to such combinations as *interval conditions*. The language of interval conditions over V^d is denoted as $\mathcal{L}_{V^\mathrm{d}}$.

With some abuse of notation, we use $\mathcal{L}_\mathcal{E}$ to indicate the language of interval conditions over the attributes mentioned in the activities of $\mathcal{E}$. Given $\varphi \in \mathcal{L}_{V^\mathrm{d}}$, we denote by $Var(\varphi) \subseteq V^\mathrm{d}$ the set of data variables mentioned therein.

When an assignment $\nu : V^\mathrm{d} \to \mathbb{R}$ satisfies an interval condition $\varphi \in \mathcal{L}_{V^\mathrm{d}}$, we write $\nu \models \varphi$ (see Appendix A.2 for the formal definition). With some abuse of notation, we say that an event $e = (a, n, \nu)$ satisfies an interval condition φ, written $e \models \nu$, if $\nu \models \varphi$.

Example 2. The following interval condition captures the expected ranges for different patient's health parameters:

$$temp > 35 \wedge (ab\_pain = \top \vee ab\_pain = \bot) \wedge leuk > 4.$$

This condition is satisfied by the assignment $\{temp \mapsto 36, ab\_pain \mapsto \bot, leuk = 5\}$, but not by the assignment $\{temp \mapsto 34, ab\_pain \mapsto \bot, leuk \mapsto 5\}$ (as the temperature condition is violated in the latter case). ◁

5.3. Data Petri nets

To model procedural process specifications, we rely on *Data Petri nets with interval conditions* (simply referred to as DPNs in the paper), following [13,14]. DPNs extend traditional place-transition nets with the possibility to manipulate scalar case variables, which are used as basic building blocks to constrain the evolution of the process through data-aware read–write interval conditions (called *guards*) assigned to transitions. In particular, a *Data Petri net with interval conditions* (DPN) D over a set $\mathcal{E}$ of

activities is a tuple (q, P, T, F, l, V, r, w), where *(i)* (P, T, F) is the Petri net graph; *(ii)* $l : T \to \mathcal{N}_\mathcal{E} \cup \{\tau\}$ is a *labeling* function (here τ denotes a *silent* transition); *(iii)* V is a set of net's data *variables*; *(iv)* $r : T \to \mathcal{L}_V$ (resp., $w : T \to \mathcal{L}_V$) is a *read* (resp., *write*) *guard-assignment* function, mapping every transition $t \in T$ into a read (resp., write) guard from $\mathcal{L}_V$.

The execution semantics of DPNs extends that of place-transition nets with the ability to check and manipulate the net's variables via transition guards. As opposed to classical Petri nets [22], a transition is *enabled* in our setting only if its read and write guards are satisfied under a given "firing mode" – a function that assigns values only to variables of the guards – and all the input places of the transition contain sufficiently many tokens to consume. Here, to check the read guard, the firing mode function picks values currently available in the net's state, while for the write guard it assigns to its variables any real values that would satisfy the guard (this accounts for "constrained" user input). When a transition is enabled, it may fire by consuming the necessary amount of tokens from its input places and producing the necessary amount of tokens in its output places, and by updating all the values assigned to variables in the write guard using the firing mode function. Values assigned to all other variables remain untouched.

In this paper, we deal only with DPNs that are 1-bounded and well-formed over their respective set of event activities $\mathcal{E}$. The well-formedness means that a silent transition cannot update net variables, and the write guard of each non-silent $t \in T$ of D uses variables matching attributes of an activity (n, q, A), for which $l(t) = n$ and either $q = D$ (i.e., (n, q, A) is a local activity for D), or $q = \mathsf{global}$ (i.e., (n, q, A) is a global activity). Such requirements appear natural in the monitoring context: we can always assume that variables are only manipulated by a fired visible transition, which is triggered by the corresponding event. Furthermore, an event does not bring more data than is foreseen by the system design, and its payload is used to update the net variables (proviso that the corresponding write guard is satisfied).

Example 3. Fig. 1 graphically depicts a DPN. Notice that the DPN is well-formed w.r.t. a set of activities containing the three activities introduced in Example 1. ◁

5.4. Declare with local filters

To capture declarative constraints over the different DPNs and global activities, we adopt a data-aware variant of the well-known Declare language [23]. In particular, the data-aware extension, that we consider, is in line with the notion of activity introduced before (including a process reference and a set of attributes, beside the activity name), and with interval conditions specified over attributes. Process references are used in this context to capture: *(i)* multi-process constraints that relate activities of different specifications; *(ii)* constraints involving global activities, thus regulating when and under which conditions they are expected to (not) occur.

A (propositional) Declare specification describes (temporal) *constraints* that must be satisfied throughout the system (or process) execution. Constraints, in turn, are based on *templates*. Templates are patterns that define different types of properties, parameterized by the actual activities on which they are applied. To build constraints, templates are then instantiated on actual activities. The semantics of Declare templates is usually formalized using Linear Temporal Logic over finite traces (LTL_f) [8,24].

In our data-aware extension, activities come with process references and also attributes. It is then natural to consider interval conditions (from Section 5.2) over those attributes, expressing "local filters" that identify which actual events match. We hence call the resulting language LF-Declare and use the following grammar to define its constraints:

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

Here, $q \in \mathcal{P}_\mathcal{E}$ is a process reference, $n \in \mathcal{N}_\mathcal{E}$ is an activity name, $q.n$ indicates the activity with reference q and name n (due to well-formedness, this activity is unique), and φ is an interval condition over $A_\mathcal{E}(q.n)$ (attributes of activity $q.n$). This grammar defines the syntax of an extension of LTL_f with interval conditions, which we refer to as LTL_f^{IC} (LTL over finite traces with interval conditions). Full definitions of the syntax and semantics of this logic are given in Appendix A.4.

As in standard LTL_f, **X** denotes the *strong next* operator (which requires the existence of a next state where the inner formula holds), and **U** denotes the *(strong) until* operator (which requires the right-hand formula to eventually hold, forcing the left-hand formula to hold in all intermediate states). Starting from LTL_f^{IC} and the standard templates of (propositional) Declare, one can derive the complete set of LF-Declare patterns (see Appendix A.4 for more details), each of which comes with its name, graphical representation, and formalization in LTL_f^{IC}.

Example 4. Consider the example scenario given in Section 2, more specifically the logic-based rule specifying that whenever the initial assessment of the patient (AP) from the clinical guideline CG1 is executed, then it must eventually be succeeded by starting the clinical guideline CG2. This rule can be captured as response(CG1.AT,CG2.start), assuming that CG2 contains a special activity CG2.start. This constraint, in turn, corresponds to the LTL_f formula:

$$\mathbf{G}(CG1.AT \to \mathbf{XF}(CG2.start)).$$

However, the above constraint is not sufficient in the given scenario. It is necessary to further specify that this constraint applies only if abdominal pain (*ab_pain*) is detected during AP. This can be captured as response(CG1.AT $\wedge$ *ab_pain* $= \top$,CG2.start), which corresponds to the LTL_f^{IC} formula:

$$\mathbf{G}((CG1.AT \wedge ab\_pain = \top) \to \mathbf{XF}(CG2.start)). \quad \triangleleft$$

Notice that the data conditions of LF-Declare mirror the guard language of DPNs, providing a solid basis for their combination. Differently from other, richer data-aware extensions of Declare, interval conditions cannot mutually relate different attributes of the same or of distinct activities, as done, for example, in [25–27]. Notice that, for the type of enactment and (anticipatory) monitoring we consider in this paper, adding such features would make the resulting underlying temporal logic too expressive, in turn causing undecidability of these two tasks [28].

5.5. Hybrid multi-process systems with local–global memory

We are now ready to integrate in a unique, combined specification the procedural and declarative process components described in Sections 5.3 and 5.4. This results in what we call a *hybrid multi-process system with local–global memory*, which consists of the following elements:

- a set of local and global activities;
- a set of shared *global variables* that are accessed and manipulated by the different process specifications;
- a set of procedural process specifications, each defined as a DPN that uses local and global activities to operate over global and local variables;

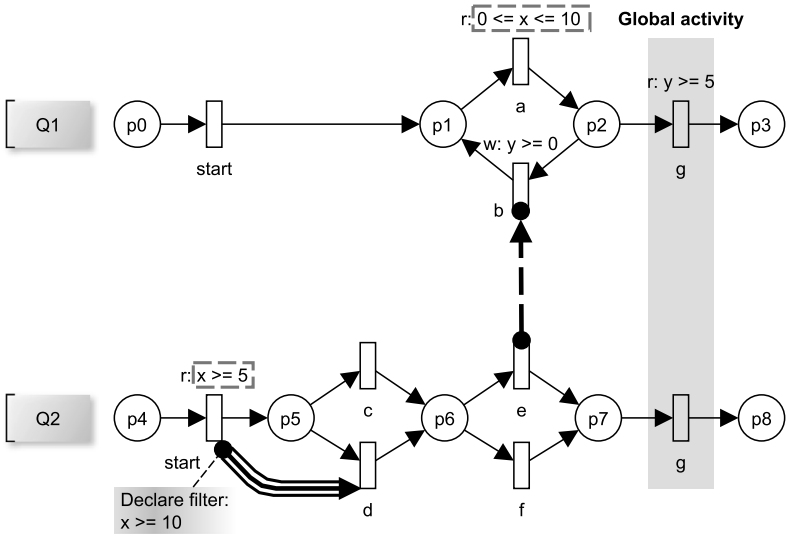


Fig. 16. Graphical representation, in the style of [21], of an HMPS consisting of two procedural (DPN) process specifications and two declarative (LF-DECLARE) constraints. In the figure, g is a global activity, x is a global variable, and y is a variable local to Q1.

- a declarative specification in LF-DECLARE specifying the behavioral boundaries of the system, and in particular constraining global activities and local activities of the procedural process specifications, as well as the attributes of such activities.

Definition 1 (*Hybrid Multi-Process System with Local–Global Memory*)**.** A *hybrid multi-process system with local–global memory*, HMPS for short, is a quadruple $\mathcal{S} = (\mathcal{E}, \mathcal{V}, \mathcal{P}, \mathcal{D})$, where:

- $\mathcal{E}$ is a finite set of activities, with $\mathcal{E}^g = \{(n, q, A) \in E \mid q = \text{global}\}$ the set of global activities;
- $\mathcal{V} \subseteq \mathcal{A}_{\mathcal{E}}$ is the set of global variables;
- $\mathcal{P}$ is a finite set of (procedural) DPN models over activities $\mathcal{E}$ and variables $\mathcal{A}_{\mathcal{E}}$, such that the variables used by two distinct DPNs from $\mathcal{P}$ only intersect over $\mathcal{V}$;
- $\mathcal{D}$ is a LF-DECLARE (declarative) specification component over $\mathcal{E}$. ◁

Example 5. Fig. 16 shows a graphical representation of a relatively simple HMPS. It consists of two procedural process specifications named Q1 and Q2. There is one global variable x, and another variable y local to Q1. Activity g is global, while all other activities are local to the two specifications (including the two start activities — the one for Q1 has no attribute, whereas the one for Q2 has x as attribute). Finally, the HMPS is equipped with two LF-DECLARE constraints:

- chain response(Q2.start $\wedge$ $x \geq$ 10,Q2.d), indicating that whenever Q2 is started in a moment where $x \geq 10$, then the activity d is expected to occur next;
- neg-succession(Q2.e,Q1.b), expressing that it is not possible to execute b in Q1 after e has occurred in Q2. ◁

We now discuss the execution semantics of HMPSs. Achieving a conceptually reasonable solution is far from trivial and cannot be achieved by simply "merging" together the semantics of all the involved specifications.

Let us start by defining a state of a HMPS. The system state needs to keep track of the state of each individual DPN, while also providing enough information to check the satisfaction of the LF-DECLARE constraints (see Appendices A.3 and A.4 respectively for the underlying formal definitions). The latter requires recalling the history of events produced from the beginning of the execution until the current moment, since LF-DECLARE constraints are evaluated over traces.

To this end, a *state of HMPS* $\mathcal{S} = (\mathcal{E}, \mathcal{V}, \mathcal{P}, \mathcal{D})$ is a triple $(Q_{\mathcal{P}}, \alpha_{\mathcal{V}}, \sigma)$, where *(i)* $Q_{\mathcal{P}} = ((M_1, \alpha_1), \ldots, (M_n, \alpha_n))$ is a $\mathcal{P}$*-state* holding states of all the DPNs from $\mathcal{P}$; *(ii)* $\alpha_{\mathcal{V}} : \mathcal{V} \to \mathbb{R}$ is the *global variable valuation*, which must agree with the respective local valuations from $\mathcal{P}$ (i.e., for any $1 \geq i, j \leq |\mathcal{P}|$, it holds that $\alpha_i(v) = \alpha_j(v) = \alpha(v)$); *(iii)* σ is a trace over $\mathcal{E}$ representing the *execution history*. A state is *initial* if its execution history is empty.

Example 6. An initial state of the HMPS in Fig. 16 is the one assigning one token to p_0 in Q1 and to p_4 in Q2, and -1 to both the global variable x and the local variable y. ◁

The enactment for HMPSs is defined by taking into account the following four inter-specification types of interactions.

(1) Enablement in procedural process specifications. In our setting, the procedural specifications are treated as enablers of execution steps (i.e., indicate what *can* be done next) and do not force specific executions.

Example 7. Consider the HMPS of Fig. 16 in the state where place p_5 contains a token. In this state (ignoring all constraints for now), both c and d are enabled and one of them will fire. ◁

(2) Local/global activities and asynchronous/synchronous execution. Silent transitions and transitions labeled by local activities are enabled based on the local state of their DPN, and consequently are also, in a way, fired locally: upon firing, the other procedural specifications stay put. This captures a form of *asynchronous* execution.

The case of a transition t labeled by a global activity g is instead radically different. Since the activity is global, its execution affects the other procedural specifications as well. This calls for a clarification on how this relation should be interpreted. We opt for the following, *synchronous* semantics: transition t is enabled in the whole HMPS if it is enabled in every procedural specification that contains g. The execution of g is then performed *synchronously* for all the involved procedural specifications, mapping one event for g to the simultaneous firing of multiple transitions. This reconstructs, in the more general setting of HMPS, the execution semantics of [9].

Example 8. Consider the HMPS of Fig. 16 and, in particular, the global activity g. Since both procedural specifications Q1 and Q2 actually employ g, an event for g can only occur by simultaneously firing the two transitions of Q1 and Q2 contained in the gray area. This, in turn, can only occur in states where both places p_2 and p_7 contain a token. The result of executing g is that of consuming those tokens while simultaneously producing one token in p_3 and one in p_8. ◁

(3) How declarative constraints force specific executions among the ones allowed by a procedural specification. We can force specific executions of a procedural specification through LF-DECLARE declarative constraints. In particular, constraints can be used to force executions that do not violate the constraints.

Example 9. Consider again the HMPS of Fig. 16, assuming that the start transition of Q2 is fired in a state where global variable x is assigned to 10 (which is indeed satisfying the read guard attached to the transition). In this case, the LF-DECLARE chain response constraint attached to the transition requires that activity d is executed immediately after. This in turn means that even if, from the point of view of the procedural specification, both c and d are enabled (since place p_5 contains a token), only d can fire, since the execution of c is forbidden by the chain response constraint. ◁

(4) Interaction with declarative constraints and anticipation of conflicts. So far, we have been discussing the enablement of transitions only considering the current HMPS state. We now argue that this is a satisfactory approach to enactment only if we accept the possibility that execution may at some point lead to a state where some declarative constraints present in the system are irrevocably violated (i.e., violated without the possibility of further continuing the execution to bring them back to satisfaction).

This problem has been widely studied in runtime verification and (anticipatory) monitoring [29], also in the context of process monitoring for DECLARE constraints [7,8], but has not been explored for hybrid systems.

To ensure that, in an enactment step, the HMPS must agree with that step not only by considering the history collected so far and the current state of affairs, but also by ensuring that there exists a continuation of the execution that satisfies all constraints present in the HMPS. This yields a more sophisticated notion of enablement that combines the past and the present with speculative reasoning on the possible futures. The sophistication of the resulting framework shows the power of hybrid specifications.

Example 10. Consider the HMPS of Fig. 16, and the state assigning one token to p_2 and one token to p_6, with x assigned to 10 and y to -1. Activities e or f appear to be executable in this state in Q2. In fact, their corresponding transitions are enabled in the DPN, and their firing does not lead to the permanent violation of the LF-DECLARE specification. However, by reasoning on the possible future continuations, it becomes instead clear that e cannot be executed, and consequently there is currently no choice for how to continue Q2. In fact:

- Executing e has the effect that the neg-succession constraint of the HMPS forbids to execute b from Q1 afterwards.
- However, executing b is essential, as Q1 needs to update y to some value ≥ 5 to be able to finally trigger the transition of Q1 that has the global activity g as label.

On the other hand, if Q1 progresses first by executing b and assigning y to some value ≥ 5, then it becomes possible to choose e in Q2. ◁

Now, after having discussed heterogeneous elements that are paramount for defining the enactment of HMPSs, we are ready to formalize it. To do so, we introduce the concept of a move that, in turn, refers to one of the three cases:

1. the firing of a silent transition in a procedural process specification;
2. the occurrence of a local event (i.e., an event referring to a local activity), and the firing of a corresponding transition in a procedural process specification;
3. the occurrence of a global event (i.e., an event referring to a global activity), and the simultaneous firing of multiple transitions in the affected procedural specifications.

Definition 2 (*Move*)**.** A *move* is a pair (y, β), where: (i) $\beta : \mathcal{V} \cup \bigcup_{1 \leq i \leq n} V_i \nrightarrow \mathbb{R}$ is a partial valuation function, and (ii) $y \in T^\tau \cup \mathcal{N}_\mathcal{E} \times \mathcal{P}_\mathcal{E}$, where T^τ is the set of all transitions from $\mathcal{P}$ that are labeled with τ. ◁

Given a move, we need to know whether it is enabled in a given system state. To define the enablement, we introduce a notion of complete traces — traces that properly execute procedural specifications, while leading to a final state where all declarative constraints are satisfied.

Definition 3 (*Complete Trace*)**.** Given an HMPS $\mathcal{S} = (\mathcal{E}, \mathcal{V}, \mathcal{P}, \mathcal{D})$ with initial state $s_0 = (Q_{\mathcal{P}_0}, \alpha_{\mathcal{V}_0}, \emptyset)$, a trace σ is called *complete* iff $\sigma \models \mathcal{D}$ and σ can be replayed on S with its initial state s_0, that is, every $(n, q, \nu) \in \sigma$ can be mapped either onto a possibly empty set of transitions (if $q = \text{global}$), where each transition t has a label n and its partial valuation function agrees with ν on $Var_w(t)$, or onto a single transition (if $q \neq \text{global}$) labeled with n and whose partial valuation function agrees with ν on $Var_w(t)$.[5] ◁

The concept of move enablement (and eventually firing) is essentially mirrored from the transition enablement of DPNs.

Definition 4 (*Move Enablement*)**.** Given an HMPS $\mathcal{S} = (\mathcal{E}, \mathcal{V}, \Omega, \mathcal{P}, \mathcal{D})$ and a state $s = (Q_\mathcal{P}, \alpha_\mathcal{V}, \sigma)$ over $\mathcal{S}$, a move (y, β) is *enabled* for $\mathcal{S}$ in s if the following conditions hold:

1. the move refers to an enabled transition labeled with τ or a local activity, that is, $y = t \in T^\tau \cap T_i$ and $(M_j, \alpha_j)[y, \beta\rangle$, for some $D_j \in \mathcal{P}$;
2. the move refers to a global activity, and all procedural specifications mentioning that activity have a corresponding enabled transition, that is, $y = (n, \text{global})$ and for all $D_i \in \mathcal{P}$ and $t_i \in T_i$ such that $\ell(t_i) = n$, we have $(M_i, \alpha_i)[t_i, \beta\rangle$;
3. the move guarantees the possibility of continuing towards a complete trace, that is, there exists a trace suffix σ' such that $\sigma \cdot e \cdot \sigma'$ is complete, where $e = (n, q, \nu)$ is an event obtained from $y \in \mathcal{N}_\mathcal{E} \times \mathcal{P}_\mathcal{E}$, β and s.

Finally, whenever an enabled move is executed, we need to define how it updates the system state and whether the update of the execution history can be completed by further moves into a trace that can be replayed on the HMPS model.

Definition 5 (*Move Enactment*)**.** An enabled move can be *enacted* resulting in a new state $s' = (Q'_\mathcal{P}, \alpha'_\mathcal{V}, \sigma')$, denoted as $s[y, \beta\rangle s'$, such that

- $Q'_\mathcal{P}$ is obtained from $Q_\mathcal{P}$ by firing all enabled transition under the firing mode β;

[5] In the case of $q = \text{global}$, the set of transitions is possibly empty as the global activity may match only against the one in $\mathcal{E}^g$, which does not affect $\mathcal{Q}$.

- $\alpha'_{\mathcal{V}}(x) = \beta(x)$, if $x \in Var_w(t)$ for each t such that $\ell(t) = n$ and $y = (n, q)$, and $\alpha'_{\mathcal{V}}(x) = \alpha_{\mathcal{V}}(x)$, otherwise;
- $\sigma' = \sigma$, if $y \in T^\tau$, and $\sigma' = \sigma \dot{e}$, otherwise. ◁

All the provided definitions are directly translatable into corresponding algorithmic checks, with the exception for condition 3 of Definition 4, which requires an algorithmic technique to check for the existence of a continuation of the current trace towards a complete one. This is directly obtained as a by-product of the construction of a monitor for the HMPS of interest, which is the subject of the next section.

We close this section by an important clarification on the overall execution semantics HMPSs. According to Definition 3, a trace is complete if it contains a sequence of transitions that is compatible with the firing semantics of each procedural component, and is so that, globally, it satisfies all the declarative constraints. This, in turn, means that no specific requirement is given on the expected progression through the procedural components: for each procedural component, a valid trace can stop at any moment. In Petri net terms, this means that we interpret Petri net transitions as *cold* transitions [30], which is compatible with the execution semantics recently defined for fragment-based case management [31,32].

This assumption is not in par with the setting where some procedural component comes with an expected progression from the initial state to a desired *(deadlocking) final marking* - a customary assumption in many approaches, such as, notably, workflow nets. Conceptually, this may indeed be relevant to indicate that a procedural component has to be "fully executed" regardless of the other components and the declarative constraints. In Petri net terms, this would call for a *hot* semantics of transitions therein, where whenever the current marking does not correspond to the final one, the procedural component expects that one of the enabled transitions is selected and fired.

Notably, both semantics can be seamlessly accommodated in the HMPS framework. If one takes the standard semantics of complete trace as per Definition 3, interpreting one procedural component according to the *hot* semantics simply calls for altering that component as follows:

1. a (deadlock) final marking is defined for the procedural component;
2. the procedural component is altered by introducing a special *goal transition* that is enabled only when the final marking is reached (thus being the only one enabled in that marking);
3. a LF-DECLARE constraint postulating the *existence* of such a goal transition is added to the declarative constraints, thus requiring its execution.

All in all, depending on the context, each procedural component can thus be interpreted according to the (standard) cold semantics or the hot one, following this procedure. In the remainder of the article, we adopt a hot semantics for all procedural components. This reflects that, for example, each relevant clinical guideline indeed comes with an expected final marking, which is also used to characterize that it must be (data-aware) sound [13] – namely that every (partial) trace can be extended to reach its final marking in a clean way, and that every modeled transition can be indeed fire in some execution.

6. Automata-based monitoring

Monitoring an HMPS of interest consists of receiving a sequence of events at run-time generated by a black-box system, returning a fine-grained feedback on the satisfaction or violation of each procedural and declarative specification, as well as of their interplay. Dealing with the interplay of specifications is essential for the same reason discussed in Section 5.5 in relation with the possibility of continuing the execution towards a complete trace. In fact, a running execution could lead to a state of permanent violation that cannot be ascribed to any specific specification, but only emerges from their mutual interplay.

Example 11. Consider the monitoring counterpart of the enactment described in Example 10. In particular, assume that the HMPS of Fig. 16 is monitored starting from an initial state assigning one token to p_0 and one token to p_4, with $x = y = -1$. Consider now the following (partial) trace σ consisting of the following sequence of events:

1. (start, Q1, $\emptyset$);
2. (a, Q1, $\{x \mapsto 10\}$);
3. (start, Q2, $\emptyset$);
4. (d, Q2, $\emptyset$)

This can be replayed on both DPNs, and can be extended into a trace that satisfies the two declarative constraints, so the monitor returns a positive feedback both for the single specifications and for their overall interplay.

Suppose now to extend σ with a further event of the form (b, Q1, $\{y \mapsto -1\}$). This violates the procedural specification Q1, as the written guard attached to the transition fired in correspondence of the given event, which indicates that y should be written with a non-negative value, is violated.

Suppose instead to extend σ with a further event of the form (e, Q2, $\emptyset$). Even though no single specification is violated, their interplay is, as there is no way to further continuing the execution and eventually satisfy all declarative constraints, for the same reason described in Example 10. ◁

To construct a suitable monitor, we need to tackle two main issues: *(i)* the assimilation of input process specifications into a single hybrid specification (required for the preparation phase in Section 3); *(ii)* the interpretation of incoming events against this hybrid specification (required for the execution phase in Section 3). From an algorithmic perspective, both challenges can be addressed by encoding each DPN and LF-DECLARE specification into a corresponding finite-state automaton. From the control-flow perspective, this is guaranteed by the fact that we consider safe DPNs, and that LTL_f has an automata-based characterization grounded in standard finite-state automata [8,17]. The main issue is caused by the presence of data, for a twofold reason. On the one hand, there are infinitely many assignments for the variables used in a HMPS; on the other hand, transitions in the automaton cannot only contain the indication of the processed event name, but have also to specify the corresponding guard on its attributes.

To handle these two aspects, we employ a variant of a finite-state automaton called guarded finite-state automaton (GFA), that fully captures the execution semantics of the specification. A GFA is a standard finite-state automaton, with the only difference that the transitions of the automaton are decorated with data conditions, imposing additional restraints on when a specific transition is allowed. The fact that this automaton has finitely many states is guaranteed by the specific shape of guards employed in HMPSs, namely interval conditions (Section 5). In fact, given the constants explicitly mentioned in the HMPS, and those used when defining the initial state, interval conditions partition the real line into finitely many intervals, constructed based on these constants. Such intervals can then be used to propositionalize the representation of the HMPS states (in particular for what concerns the assignments to local and global variables), as well as that of events and their payload. This is based on the approaches presented in [13,33].

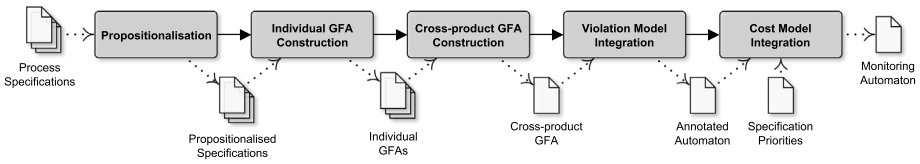


Fig. 17. Main steps and intermediate artifacts for creating the monitoring automaton.

In the following, we give a conceptual overview of the necessary steps to construct such automata. Correctness is directly obtained by considering:

- the propositionalization abstraction taken from [13,33];
- the standard construction of an automaton for each LTL_f formula;
- the construction of an automaton per DPN, directly using its execution semantics (considered in isolation);
- the fact that finite-state automata can be subject to elimination of silent moves and determinization;
- the fact that we employ the standard notion of cross-product to compute the overall monitor, accounting for the interplay of the different specifications, and dealing with the processing of global activities in line with the execution semantics of HMPSs.

6.1. Monitoring automaton

The GFA representation allows for capturing the semantic meaning of each individual process specification (including the data perspective), while also fully accounting for the execution semantics outlined in Section 5.5. Additionally, the annotated monitoring automaton enables early conflict detection and next activity recommendations as explained next in Section 6.2.

The creation of the monitor automaton occurs at the end of the preparation phase of our framework (Section 3) and consists of the following main steps (see Fig. 17):

Propositionalization — Translation of the original process specifications into propositionalized specifications based on interval abstraction. Activities and attribute-constant combinations (including intervals between the constants) used in the process specifications are extracted and enumerated to form propositions. The activity names and conditions are then replaced in the original process specifications with equivalent sets of propositions to form propositionalized specifications similarly to the approach presented in [13,33].

Individual GFA Construction — Creation of an equivalent GFA for each propositionalized process specification. For each propositionalized LF-DECLARE formula, we construct its LTL_f-based automaton using standard construction techniques [8, 17]. For each propositionalized DPN, we translate its (finite) reachability graph into a corresponding guarded automaton, making sure that, in every state, we keep track of the (propositionalized) assignment of each global variable.

In addition, we apply two additional considerations in view of the execution semantics of HMPSs. First, additional edges have to be added in consideration of global variables. Specifically, since global variables may be asynchronously updated by other specifications, in every state of the DPN automaton one has to consider the presence of additional edges accounting for all possible updates done to the global variables by other specifications. Secondly, additional self loops are added to all states of the GFA for handling events that the DPN should ignore, and additional trap states are added for handling permanent violations.

Cross-product GFA Construction — Combining the individual GFAs into a cross-product GFA. A standard automata cross-product algorithm can be used, given that the guards of the transitions are encoded as simple propositional strings. The only special consideration to be applied is that states of distinct DPN GFAs can only be combined if they agree on the (propositionalized) assignment to global variables. Notice that the asynchronous/synchronous semantics of local/global activities is mirrored in the cross-product thanks to the presence/absence of the self-loops mentioned in the previous step.

Finally, notice that the cross-product GFA should *not* be minimized. In this way, it accounts for distinct violation states, each indicating in a fine-grained way which specifications are actually causing the violation.

Violation Model Integration — Annotation of the cross-product GFA states with a corresponding global monitoring status and a corresponding monitoring status of each individual process specification. In the current implementation, this is performed through the concurrent traversal of the cross-product GFA and the individual GFAs.

Cost Model Integration — The procedure begins by annotating each GFA state with the total cost of temporary and permanent violations incurred in that state. Here, the cost is equivalent to the priority value of the specification (introduced in the elicitation phase of Section 3), thus making the highest priority specifications also the most costly to violate. A copy of these annotations is made and then updated iteratively, based on the corresponding values of the successor states, so that the lowest value is kept after each iteration. As a result, each state of the cross-product GFA is annotated with the total cost of stopping the process execution in the given state and the lowest possible total cost of violations achievable from that state.

6.2. Event processing

Event processing in the proposed monitoring approach is somewhat analogous to other existing automata-based monitoring approaches [7,8]. The main differences are that each incoming event must be translated into a propositional string belonging to the alphabet of the monitoring automaton, and two additional post-processing steps are performed after updating the state of the monitoring automaton, one for determining the new state of the monitor, and one for determining the next activity recommendations (see below). An overview of the event processing steps is provided on Fig. 18.

The processing of each observed event occurs at the beginning of the execution phase of our framework (Section 3) and consists of the following main steps:

Event Propositionalization — Translation of the observed event into the equivalent propositionalized event. The propositionalized event is required to correspond to one of the propositions used in the monitoring automaton (Section 6.1, Propositionalization step) and can therefore be used directly to transition the monitor automata to the corresponding next state.

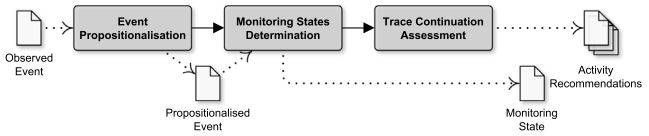


Fig. 18. Main steps and intermediate artifacts for processing a single event.

Monitor State Determination — Determination of the monitoring state based on the propositionalized event. The monitoring state can be looked up directly based on the corresponding GFA annotations and consists of a global monitoring state and the monitoring state of each individual process specification (Section 6.1, Violation Model Integration).

Trace Continuation Assessment — Determination of the next activity recommendations based on the cost model annotations of the GFA. Each immediate successor state of the current monitor automaton state is evaluated to determine the set of states that can lead to the minimum total cost of violations. The propositional labels of the transitions leading to the best successor states are translated into equivalent events (including the data perspective) and returned as the set of activity recommendations.

7. Evaluation of practical feasibility

As a prerequisite of the evaluation, we first outline some general principles for visualizing HMPSs. These principles are then used to describe the HMPS of our example scenario (Section 2), as well as the HMPSs of two additional real-life scenarios. For each scenario, we also describe all modeling steps that were necessary to correctly define the interplay between the initial individual process specifications (base-models). In doing so, we demonstrate that the advancements made in this paper enable us to now go from the artificial examples of our previous works [9,10] towards addressing practical real-life scenarios. For performance and scalability tests, we refer the reader to [10].

7.1. Visualizing the hybrid specification

To visualize the hybrid specifications, we take inspiration from the works on hybrid process modeling and discovery (e.g., [21, 34]). As, for example, shown in Fig. 19, we combine already existing conventions for visualizing DPNs and DECLARE constraints by using the transitions in DPNs as DECLARE constraint endpoints (e.g., Constraint C1). The local filters used in DECLARE constraints are placed in gray-white rectangles, each connected to the transition to which the filter applies (e.g., activity AP in constraint C1). If DECLARE constraints refer to any additional activities, then these are added as rectangles (e.g., activity stopAT in constraint C2).

Local activities (e.g., HXray) and local variables (e.g., *h_fract*) are represented using existing conventions. Global variables are highlighted by surrounding them with a rectangle with a gray dashed line (e.g., *ab_pain*). Differently from Section 4, global activities are not merged, but instead aligned vertically and then highlighted with a shared gray background (e.g., FD). The latter was chosen for improving the readability of larger visualizations. Violation costs are currently omitted in order to reduce the complexity of the visualization.

7.2. Handling the example scenario

Handling the example scenario presented in Section 2 requires not only defining which activities and attributes are global and which are local, but also making some modifications in the base-models themselves. Most, if not all, of these modifications could be avoided by simply creating the models with keeping our framework in mind from the very start. However, given that our framework has been introduced only recently, it is more likely that one would adapt already existing models. To mimic this situation, we deliberately described our example scenario (Section 2.1), and the base-models therein (Fig. 1 and Fig. 2), without any of the interplay-related modifications, instead presenting these modifications as part of the evaluation. In total, five modifications of the base-models were necessary.

The first modification (no. 1 in Fig. 19) is the addition of an artificial activity init in the model CG2a. This is necessary because CG2a originally started with a choice between two initial activities AP and CT, but our current formalism requires each constraint to refer to a single global or local activity. By adding activity init we can define constraint C1 as an LF-DECLARE constraint response(CG1a.AP $\wedge$ *ab_pain* $= \top \wedge$ *leuk* $\geq$ 11, CG2a.init). We could also use CT for constraint C1, which would be equivalent w.r.t. the allowed executions, but not as clear business-wise. An alternative could be adding support for imposing behavior on entire models with constraints (e.g., constraint response(CG1a.AP $\wedge$ *ab_pain* $= \top \wedge$ *leuk* $\geq$ 11, CG2a) to specify when executing CG2a should start, without referencing a specific activity).

The second modification (no. 2 in Fig. 19) is a direct application of the "Data-based Activity Skipping" pattern (Section 4.3.1). Note that the condition for skipping AP in CG2a is stricter than in CG1a.

The third modification (no. 3 in Fig. 19) is almost the direct application of the "Optional Data Petri Nets" pattern (Section 4.3.5). The only difference is that CG1b and CG2b are triggered by global activities instead of by constraints.

The fourth modification (no. 4 in Fig. 19) is an example of a more complex case of optionality. Here, we should modify CG2a to also be optional, but recall that every global activity always progress all process specifications containing that activity. As a result, applying the "Optional Data Petri Nets" pattern (Section 4.3.5) is not sufficient as the (business-wise valid) execution HXray, AP[ab_pain = $\bot$, temp = 36.6, leuk = 5], FD[h_fract = $\top$], ..., would still result in a violation of CG2a. This is because executing the global activity FD always requires tokens in its preceding places in CG2a. To solve this issue, we have added two silent transitions before FD, one simply for readability, and the other to allow for skipping AP and CT in CG2a. The global activity appSD is not a concern since CG1 does not contain that activity and we have already made CG2b optional. The global activity D, instead, requires adding another silent transition that would allow D to be executed after executing FD. Furthermore, we need to make sure that this silent transition will be taken iff FD was executed without starting the execution of CG2a. This is achieved by adding an additional local variable *init* and two corresponding guards.

The fifth modification (no. 5 in Fig. 19) is a variation of the "Blocking Petri Net Regions" pattern (Section 4.3.3). The pattern is applied here with the same global control variable *inS* across the four individual DPNs. Furthermore, it is applied in a way that it not only prohibits multiple concurrent surgeries, but also activity D while a surgery is ongoing. More specifically, either surgery decision (activities hipSD and appSD) can be made only while *inS* $= \bot$. Any surgery decision then sets *inS* $= \top$, thus blocking the other decision and, as a result, the corresponding "sub-process". This block is lifted by activity postSA by setting *inS* $= \bot$ that, in turn, allows either the other surgery decision

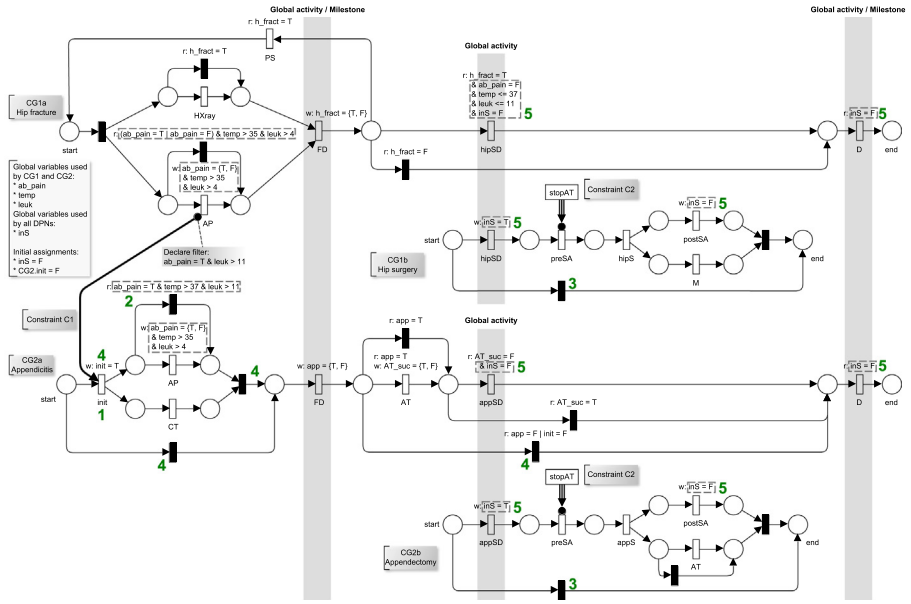


Fig. 19. Visual representation of the combined behavior of our example scenario. Numbers 1–5 highlight the base-model modifications discussed in Section 7.2.

to be made, or the patient to be discharged. Note that, the post-surgery activities M and AT are not directly tied to *inS* and can even occur after the activity D.

Finally, we define the constraint chain precedence(stopAT, preSA) (constraint C2) to specify that activity preSA requires activity stopAT to be executed beforehand. In this case, we do not specify the provenance of the activities, since the constraint applies across all individual models.

In summary, a total of five modifications were necessary in the example scenario, three of which (2, 3, and 5 in Fig. 19) were direct applications or variations of the modeling patterns introduced in Section 4.3. The other two are more complex modifications that point towards some necessary improvements of the proposed semantics with regards to optionality and enablement of individual process specifications.

7.3. Sepsis treatment process

Our first real-life scenario is based on the well-known sepsis treatment event log [35] and on the corresponding extensive case study presented in the PhD thesis of F. Mannhardt [36].

7.3.1. Base-models

For our base-models, we use the Petri net representations of the activity patterns from [36, Section 13.5]. Originally, these patterns are used as input for Guided Process Discovery [36, Section 9.2]. However, each activity pattern, as presented in the thesis, is basically an independent procedural model composed of some low-level activities, which represents the execution of a single high-level activity. In the following, we view these activity patterns as individual sub-processes that will be combined within our framework.

The original activity patterns can be found in [36, Section 13.5]. However, they are also easily recognizable in Fig. 20, as each Petri net of that figure (labeled with P1–P5) is based on exactly one activity pattern. Out of these, P3 was redesigned (while retaining it's behavior) because the original model contained multiple copies of the same activity within the same model, which we currently do not support. We also slightly modified P2 to make it visually more compact.

Originally, the activity patterns did not contain any data conditions, most likely because the corresponding approaches presented in [36, Section 9.2.2] "require that variables are not shared between activity patterns". However, many of the conditions discovered in the same case study [36, Section 13.3] actually do cross the activity pattern boundaries (e.g., *Hypotensie* is written in P1 but read in P3). Given that our approach does not have this limitation, we reintroduced all these data conditions. The only exception is *LacticAcid*, which, based on the text of the case study, is used to simply check if Lactic Acid has been executed or not, and was changed into a boolean attribute accordingly. We also reintroduced the corresponding transitions as needed, e.g., the transition to skip the release activities in P5, if $LacticAcid = \bot$.

7.3.2. Interplay

We first note that, [36, Section 13.5] also presents a Petri net, which composes the individual activity patterns (from which we derived our base-models) into a high level process description. This could be used as the basis for combining our base-models, however, it does not capture all of the intricacies of the actual sepsis treatment process. Instead, we have analyzed both the normative model [36, Section 13.1.3] and the data conditions [36, Section 13.3] of the same case study to combine our base-models in a more accurate manner.

We begin by connecting the base-models P1 and P2 using two constraints (no. 1 in Fig. 20). First, we add an artificial activity P2.init to define constraint precedence(P1.ER Registration, P2.init). This constraint specifies that any activities of P2 can only be executed after P1.ER Registration, as required by the normative model. Second, we define constraint succession(P1.ER Registration $\wedge$ $DiagnosticLacticAcid = \top$, P2.Lactic Acid) to enforce that P2.Lactic Acid can be executed if and only if P1.ER Registration occurs with $DiagnosticLacticAcid = \top$ (as required in [36, Section 13.3]).

We also connect P1 to P3 by using three constraints (no. 2 in Fig. 20). As before, we add a similar artificial activity P3.init to define constraint precedence(P1.ER Sepsis Triage, P3.init). This constraint specifies that any activities of P3 can only be executed after P1.ER Sepsis Triage, as required by the normative model. Additionally, we add constraints neg-succession(P3.init, P1.IV Liquid) and neg-succession(P3.init, P1.IV Antibiotics) to specify that

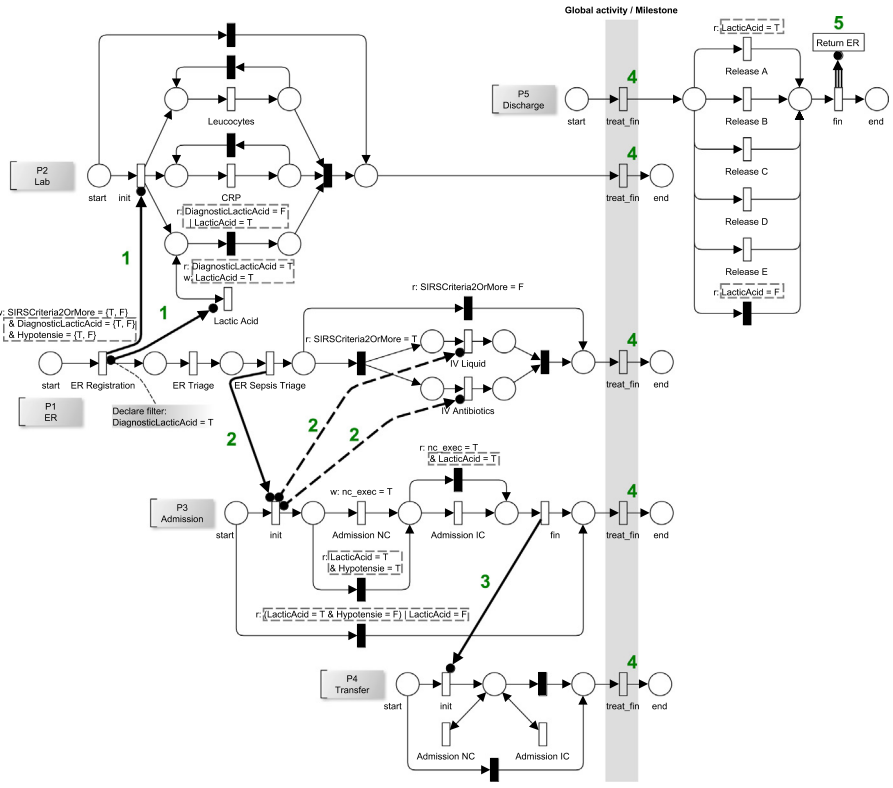


Fig. 20. Visual representation of the combined behavior of the sepsis treatment process. Numbers 1–5 highlight constraints and base-model modifications discussed in Section 7.3.2.

activities P1.IV Liquid and P1.IV Antibiotics can never occur after the execution of P3 has started. This also comes from the normative model where the execution of these two activities is only possible between P1.ER Sepsis Triage and any type of admission.

Base-models P3 and P4 consist of the same two same-labeled activities. However, both of these must still be defined as local activities because P3 refers to the initial admission of the patient, while P4 refers to transfers between different wards of the hospital [36, Section 13.5]. To connect P3 and P4, we add two artificial activities P3.fin and P4.init to define constraint precedence(P3.fin, P4.init) specifying that transfers of the patient may only occur after the patient has been admitted (no. 3 in Fig. 20).

Based on the normative model, all the activities in P1–P4 must be either executed or skipped before any of the activities in P5 can be executed. This is represented by a single silent transition in the normative model. The most natural way of replicating this behavior in our framework is by introducing an artificial activity treat_fin (no. 4 in Fig. 20) at the end of base-models P1–P4 and at the beginning of P5, and defining it as a global activity. Additionally, we need to add a silent transition into base-models P2 and P4 to allow treat_fin to be executed correctly even if the execution of these models was not triggered for the given case.

We also need to handle activity Return ER which was not part of any of the activity patterns, and thus is not included in our base-models. We could create an additional model only containing that activity. However, a more elegant solution (no. 5 in Fig. 20) is to add an artificial activity P5.fin and then a constraint chain precedence(P5.fin, Return ER) specifying that activity Return ER requires P5.fin to occur immediately beforehand.

Finally, we also need to define which variables are global and which are local. In this case, there are no same-labeled variables with a different meaning or usage within different models. Therefore, we can take the simple approach of defining all variables, which are shared by multiple base-models, as global variables and all other variables as local variables.

In summary, we were able to successfully represent the behavior of a real-life sepsis treatment process using the modeling patterns and semantics discussed in this paper. Achieving this required seven constraints and five artificial activities across all base-models. This is a promising result in terms of the practical feasibility of the approach. However, the use of artificial activities still adds modeling complexity. A common theme of artificial activities in this case study is that they were almost always added either at the beginning or at the end of the base-models, highlighting the need for better handling both enablement and also fulfillment of individual process specifications.

7.4. Loan application process

Our second real-life scenario is based on the loan application event log [37] from BPIC 2017. As in Section 7.3, we do not create the base-models for the evaluation from scratch. Instead, we use the analysis results presented in [38,39], and also our own insights gained with the tools RuM [40][6] and Apromore [41][7]. The resulting combined behavior is shown in Fig. 21.

7.4.1. Base-models

The BPIC 2017 loan application process is described in [38] as three separate, but interconnected, business processes: loan applications, loan offers, and workflow activities. For each process, the authors provide a Petri net model discovered via automated process discovery. For the original models, we refer the reader to [38]. However, they are also easily recognizable as M1, M2, and M3 in Fig. 21. All three models were usable in our approach as-is, but further interplay related modifications were still necessary (Section 7.4.2).

6 https://rulemining.org/
7 https://apromore.com/

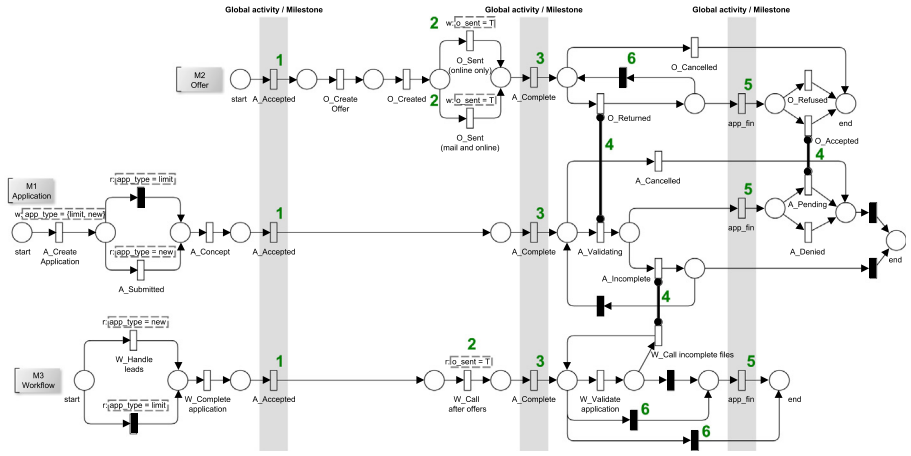


Fig. 21. Visual representation of the combined behavior of the loan application process. Numbers 1–6 highlight constraints and base-model modifications discussed in Section 7.4.2.

The models did not contain any data conditions, however, based on the text of [38], we added conditions for skipping activities A_Submitted and W_Handle leads based on the loan application type (guards referencing the variable *app_type* in Fig. 21). We note here that, while loan application type is a case attribute in the original event log, we require each variable to be associated with a specific event. In this case, we associated the loan application type to activity A_Create Application which is always the first activity of the process in the original event log.

7.4.2. Interplay

In [38] the authors highlight a total of 17 connections between the three models. However, all of these connections are presented in an informal manner and most of them are not described in detail. Furthermore, we discovered that some of these connections have numerous counterexamples in the original event log. Therefore, we complemented the analysis from [38] with the information found in [39], which provides an overview of the main phases of the same process, and with our own process analysis to validate the correctness of the process behavior.

Based on [39], the process starts with submitting the loan application, which is followed by a series of initial checks. Any loan offer can then be made only after these initial checks have completed (represented by activity A_Accepted). This, in turn, gives us the first natural milestone of this process, which we model by first adding the activity A_Accepted into models M2 and M3, and then making that activity global (no. 1 in Fig. 21).

The next interplay related connection (no. 2 in Fig. 21) is based on [38] and specifies that activity W_Call after offers can only be executed after an offer has been sent (O_Sent (online only) or O_Sent (mail and online)). We model this requirement by applying the "Blocking Read Guards" pattern (Section 4.3.2) together with a global boolean variable *o_sent*. More specifically, activities O_Sent (online only) and O_Sent (mail and online) will set the value of *o_sent* to $\top$, thus satisfying the guard of W_Call after offers and allowing it to be executed. As an alternative, we could instead use a target-branched precedence constraint [42].

Based on the activity descriptions in [39], the application is considered complete (represented by activity A_Complete) once the orders have been sent. We found that activity W_Call after offers should also occur before that point. Therefore, A_Complete is the second natural milestone of this process, which we again model by first adding the activity A_Complete into models M2 and M3 and then making that activity global (no. 3 in Fig. 21).

Next, we rely on the general descriptions of the process [38,39] to define some additional relations (no. 4 in Fig. 21) between the activities that follow A_Complete. More specifically, we define constraints co-existence(M2.O_Returned, M1.A_Validating) to specify that an order can be validated iff it is returned, co-existence(M1.A_Incomplete, M3.W_Call incomplete files) to specify that a call about incomplete files can only occur iff the application is determined to be incomplete, and co-existence (M2.O_Accepted, M1.A_Pending) to specify that an offer can be accepted iff the application is accepted (the latter is represented by activity A_Pending).

Finally, we add an artificial activity app_fin (no. 5 in Fig. 21) to represent the point after which all loan offers have been processed and a final decision on granting the loan must be made (A_Pending if the loan is granted, and A_Denied otherwise). This added activity also requires adding two additional silent transitions in M3 (no. 6 in Fig. 21) in order to support both the cases that reach activity app_fin, and also the cases that may be canceled beforehand.

We also note that, in the BPIC 2017 log, there may be multiple loan offers for each loan application, which leads to the issues of divergence and convergence [3] that our approach is currently not equipped to handle. In fact, two consecutive executions of O_Create Offer for different offers would violate the model M2. We could mitigate this modeling problem to some extent by modeling M2 as a LF-DECLARE specification, as this would allow us to compactly represent the interleavings among multiple loan offers. However, this would also require to specify additional rules over sets of loan offers. For example, if a loan application is accepted, then exactly one loan offer must be accepted for that application, while all other loan offers must be rejected. These rules would require to correlate applications and offers, calling for modeling of advanced constraints merging the temporal and the data dimension, which are in general undecidable to monitor [43].

8. Related work

We discuss related work along two directions: one oriented towards the main support task of interest in this article, namely *process monitoring*, and the other oriented towards the need of integrating multiple perspectives when dealing with processes, namely *integrated process modeling*. We structure the discussion along the main lines of research and types of approaches dealing with these two problems.

Monitoring Petri net models. There are various works that rely on different classes of Petri nets for capturing (in)appropriate system behaviors. One of the first works studies a monitoring approach that detects errors by controlling the number of tokens

inside P-invariants [44]. Another work [45] proposes a workflow management system that encompasses workflow monitoring and delay prediction modules based on resource-aware Petri nets. These approaches all share the common assumption that the full process specification is given as a single monolithic Petri net, which can be insufficient in domains with highly flexible and knowledge-intensive processes.

Monitoring declarative specifications. Here, we restrict ourselves to approaches related to DECLARE (for comparisons with other approaches please refer to [46]). While existing DECLARE approaches in general assume that a single specification is used, splitting it into multiple specifications (e.g., one specification per constraint) would be semantically equivalent. This is well exemplified in [7], where the truth value of each constraint is monitored individually and possible global conflicts are handled by the conjunction of all the constraints being monitored. The DECLARE language has also been extended to account for multiple perspectives by considering activity payloads, and corresponding monitoring approaches have been developed (see, e.g.,[27,47]). However, none of these works studied monitoring of LMP-DECLARE constraints in hybrid specifications.

Conformance checking. The task of monitoring can be considered as the runtime version of *conformance checking* [48]. The vast majority of conformance checking approaches in the process mining spectrum exclusively focus on a single modeling paradigm (declarative or procedural), and do not take into account multiple perspectives, in particular data, with some relevant exceptions briefly reviewed next.

Techniques for data-aware conformance checking have been studied both in the procedural and declarative setting. Within the procedural realm, the most investigated approach focuses on data Petri nets [13,36], introducing data-aware alignments that do not only consider the events contained in the log, but also the data payload they carry. Computing such alignments is combinatorially hard, and has been attacked using AI techniques based on optimized state–space search [14] and through logical encodings into Satisfiability and Optimization Modulo Theory [49]. In the declarative setting, conformance checking of data-aware variants of DECLARE dates back to the seminal work in [50], with a naive yes/no output, while more sophisticated forms of feedback are defined through ad-hoc algorithms in [26].

The interplay of multiple process specifications (both procedural and declarative) has been addressed in a few approaches. A recent work [21] 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 to conformance checking, the authors focus on alignment of complete traces and not on monitoring ongoing incomplete process executions. In the medical domain, [6] has brought forward the need of combining procedural knowledge encapsulated in CGs with general medical knowledge, and to consider their interplay when checking conformance of actual behaviors. In [11], this interplay is studied with the goal of obtaining explanations on the actual behavior, i.e., how to automatically explain why certain actions were taken during the treatment of a specific patient, in the light of multiple CGs and BMK. All these approaches consider the interplay of procedural and declarative models when analyzing historical data, and hence cannot be applied at online to provide runtime feedback on running executions.

Object-centric process models. Object-centric process models have been recently introduced to handle processes that do not have a single notion of case, but inherently co-evolve multiple objects, which are mutually related through one-to-many and many-to-many relationships. Tackling such processes as first-class citizens requires avoiding the "flattening" effect caused by a notion of a single case when applying modeling, analysis, and process mining techniques [51].

Different approaches to capture the control-flow backbone of object-oriented processes have been studied in literature, ranging from declarative [52] to procedural approaches [53–56].

In [52], activities are related to objects structured in a data model. DECLARE constraints over such activities can then be suitably scoped by expressing co-references through the classes and relations present in the data model.

In the procedural spectrum, approaches differentiate from each other mainly depending on whether they support explicit or implicit object manipulation. The most prominent example of implicit object manipulation is the model of proclets [54], where each object type comes with a Petri net specifying its lifecycle, and joint transitions can be modeled to express multi-type split and synchronization points. Approaches equipped with explicit object manipulation typically rely on fragments of Colored Petri nets equipped with the ability of generating new object identifiers, in the style of ν-Petri nets [57]. Such identifiers are combined into tuples to represent relationships between objects, tuples that are carried by tokens. Transitions are consequently equipped with inscriptions to manipulate such tuples and their components [56,58], while additional data constraints, expressed in a data model [55] or in a persistent database [53,56], can be used to identify the intended executions and how data are manipulated therein. Research on discovering such models from data is at its infancy, and it is currently targeting either weaker versions of object-centric models [59], or well-behaved fragments of them [60].

All these approaches are orthogonal to the framework studied here, where we currently do not consider the presence of multiple co-evolving objects, but instead focus on the simultaneous progression of single case objects through multiple process components, which may interact with each other. This is a generalization of so-called hybrid or mixed-paradigm process models, which are reviewed next.

Hybrid models. Hybrid business process representations (HBPRs) combine 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.

Originally, the combination of two paradigms was loosely coupled, i.e., distinct parts of the same process would be represented procedurally or declaratively depending on the nature of those parts. This is well exemplified in the model of pockets of flexibility, where flexible subprocesses of a procedural specification are captured declaratively through constraints [61]. This approach has been extended in [62], where a process is modeled with an arbitrary nesting of subprocesses, each specified as a Petri net or a DECLARE specification, whose activities can be in turn recursively decomposed into a Petri net or DECLARE specification. Discovery of such layered specification is tackled in [63].

While these approaches provide a weak integration of the two modeling paradigms through layering, intertwined HBPRs have been studied more recently, essentially proposing to enrich a Petri net with DECLARE constraints that may be applied to the net transitions or to further activities. This leads to a tightly coupled integration of the two components, leading to a refined notion of transition enablement for the Petri net, which also needs to take into account the declarative constraints insisting on that transition [21,64]. Discovery of (fragments of) such intertwined approach is tackled in [34], while conformance checking is targeted in [21]. The HMPS framework introduced here can be seen as a generalization of this family of approaches along three

dimensions. First, we consider multiple procedural components instead of a single, monolithic one. Second, we infuse procedural components as well as declarative constraints with global/local data variables. Third, when considering the intertwined state space emerging from their integration, we handle, in a fine-grained way, asynchronous and synchronous execution modes for activities that are shared among components.

We close by stressing that studying HBPRs is an active area of research, as witnessed by the recent proposal of a conceptual framework and a common terminology for these types of models [65], as well as the identification of a number of open research challenges in this spectrum [66].

9. Conclusion

The main focus of this paper is to transform the semantics of hybrid specifications we introduced in [9] (and fully formalized and tested later in [10]) into a mature instrument for modeling, executing and monitoring real-life processes. To do this, we made significant advancements over the original semantics by defining a comprehensive framework for modeling, executing and monitoring data-aware, hybrid multi-process specifications, combining procedural and declarative components operating over local and global variables, and equipped with local and global activities. This work was further enhanced by defining the execution semantics of these specifications, which support enactment and monitoring (as shown in this paper), but also potentially other process mining tasks, such as model simulation and conformance checking, which we will investigate as future work. Finally, we have used already existing real-life process models to demonstrate that the concepts, and by extension the semantics, introduced in this paper can be applied to real-life settings.

The avenues for future work can be divided into four categories. First, we want to develop a complete toolchain for modeling, enactment, and monitoring of HMPSs. In doing so, we also want to study algorithmic techniques to tame the state-explosion problem still present in our current technical approach. Second, the approach presented in this paper should be extended to support recovery strategies and/or runtime modifications of a given HMPS. These would respectively allow the process analysts to continue monitoring after a permanent violation (thus enabling the detection of multiple violations of a single component), and handle unforeseen circumstances (e.g., additional illnesses diagnosed during an ongoing treatment case). The overall effect would be to deal with violations of individual process components in a more fine-grained way, as in the current approach, once a violation occurs in some process component, that component will remain in a permanently violated state. This means, for example, that less impacting issues like skipping an activity cannot be distinguished from more severe violations. Third, while we have employed here a simple language for expressing conditions and updates over data variables, based on the so-called interval conditions [13,33], more sophisticated conditions based on (controlled) forms of arithmetics can seamlessly be employed in our framework, using the data abstraction techniques introduced in [18,67]. Finally, there are additional socio-technical aspects that should be explored, e.g., how to integrate the framework into an organizational context, and what skills would be necessary to support its different phases.

Declaration of competing interest

No conflicts to report.

Acknowledgments

The work of A. Alman was supported by the European Social Fund via "ICT programme" measure, Estonian Research Council grant PRG1226, and Kristjan Jaak national scholarship programme. The work of F.M. Maggi was supported by the UNIBZ project CAT. The work of M. Montali was supported by the UNIBZ project ADAPTERS and the PRIN MIUR project PINPOINT Prot. 2020FNEB27. The work of F. Patrizi was supported by the ERC Advanced Grant WhiteMech (No. 834228), the PNRR MUR project PE0000013-FAIR, and the Sapienza Project MARLeN. The work of A. Rivkin was supported by the UNIBZ project WineID.

Appendix A. Process components

Here we provide complete formal definitions of process components discussed in Section 5.

A.1. Events and traces

In this section we fix preliminary notions related to activities, events and traces.

Definition 6 (*Activity*)**.** An *activity* is a tuple (n, q, A), where:

- n is the *activity* name;
- $q \in \mathcal{P} \cup \{\text{global}\}$ is the *process reference* carrying either a value from $\mathcal{P}$, the set of all (local) process references, or a special global reference global;
- $A = \{a_1, \ldots, a_\ell\}$ is the set of event *attribute (names)*.

We denote by $\mathcal{E}$ a finite set of activities and define for it three additional sets:

- the set $\mathcal{N}_\mathcal{E}$ of all *activity names from* $\mathcal{E}$,
- the set $\mathcal{P}_\mathcal{E}$ of all *process references from* $\mathcal{E}$;
- the set $\mathcal{A}_\mathcal{E}$ of all *attribute names occurring in* $\mathcal{E}$.

The next definition introduces well-formednes needed to regulate the proper use of activities and references for multiple process specifications.

Definition 7 (*Well-formed Activity Set*)**.** A set $\mathcal{E}$ of activities is *well-formed* if the following two conditions hold:

1. for every pair $(n, q, A), (n', q', A') \in \mathcal{E}$, if $n = n'$ and $q = q'$ then $A = A'$;
2. for every $(n, q, A) \in \mathcal{E}$, if $q = \text{global}$ then there is no $(n', q', A') \in \mathcal{E}$ such that $n = n'$ and $q' \neq \text{global}$.

Events and trace are defined as follows.

Definition 8 (*Event, trace*)**.** An *event* over activity $(n, q, A) \in \mathcal{E}$ is a triple $e = (n, r, \nu)$, such that $n \in \mathcal{N}_\mathcal{E}$, $q \in \mathcal{P}_\mathcal{E}$ and $\nu : A \mapsto \mathbb{R}$ is a total function assigning to every attribute in A a corresponding value.[8] A *trace* over $\mathcal{E}$ is a finite sequence $\sigma = e_1 \cdots e_\ell$ of events, where each e_i is an event over some activity in $\mathcal{E}$.

A.2. Interval conditions

In this section we define the language of conditions over event attributes as well as local/global variables of process specifications (discussed in detail in the next sections). To this end, we fix a generic set V^d of data variables.

Definition 9 (*Interval Condition*)**.** An *interval condition* φ over a set V^d of data variables is an expression of the form:

$$\varphi := a \odot c \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2,$$

where: (i) $x \in V^\text{d}$; (ii) $\odot \in \{<, =, >\}$; (iii) $c \in \mathbb{R}$.

[8] While here the attributes are of type $\mathbb{R}$, simpler types such as strings with equality and booleans can be seamlessly encoded.

We make use of the following standard abbreviations: (i) $\varphi_1 \vee \varphi_2 = \neg(\neg\varphi_1 \wedge \neg\varphi_2)$; (ii) $\varphi_1 \rightarrow \varphi_2 = \neg\varphi_1 \vee \varphi_2$; (iii) $a \leq c = \neg(a > c)$; (iv) $a \geq c = \neg(a < c)$; and (v) $a \neq c = \neg(a = c)$. Interval conditions of the form $a \odot c$ are called *atomic*. We denote by $\mathcal{L}_{V^d}$ the language of interval conditions over V^d. With some abuse of notation, for a set $\mathcal{E}$ of activities, we use $\mathcal{L}_{\mathcal{E}}$ to indicate the language of interval conditions over the attributes mentioned in the activities of $\mathcal{E}$. Finally, given an interval condition $\varphi \in \mathcal{L}_{V^d}$, we denote by $\text{Var}(\varphi) \subseteq V^d$ the set of data variables mentioned therein.

We now define when an interval condition is satisfied by an assignment, formalising the intuitive interpretation of interval conditions.

Definition 10 (*Satisfaction of an Interval Condition*)**.** Given a set V^d of data variables, an interval condition $\varphi \in \mathcal{L}_{V^d}$ is *satisfied* by an assignment $\nu : V^d \rightarrow \mathbb{R}$, written $\nu \models \varphi$, if the following conditions hold:

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

With some abuse of notation, we say that an event $e = (a, n, \nu)$ satisfies an interval condition φ, written $e \models \varphi$, if ν does so according to Definition 10.

A.3. Data petri nets

Procedural process specifications are represented using DPNs [13,14] (also referred to as *data Petri nets with interval conditions*).

Definition 11 (*Petri Net with Data and Interval Conditions*)**.** A *Petri net with data and interval conditions* (DPN) over the set $\mathcal{E}$ of activities is a tuple $D = (q, P, T, F, l, V, r, w)$, where:

- $q \in \mathcal{P}_{\mathcal{E}}$ is the *name* of the DPN;
- P and T are two finite disjoint sets of *places* and *transitions*, respectively;
- $F : (P \times T) \cup (T \times P) \rightarrow \mathbb{N}$ is the *flow relation*;
- $l : T \rightarrow \mathcal{N}_{\mathcal{E}} \cup \{\tau\}$ is a total *labeling* function, with τ denoting a *silent* transition that does not correspond to a (visibile) activity from $\mathcal{E}$.
- V is a set of net data *variables*;
- $r : T \rightarrow \mathcal{L}_V$ and $w : T \rightarrow \mathcal{L}_V$ are two total *read* and *write guard-assignment* functions, mapping every transition $t \in T$ respectively into a read and write guard from $\mathcal{L}_V$.

The *preset* and *postset* of an element $x \in P \cup T$ of D are, respectively, the sets ${}^\bullet x = \{y \mid F(y, x) > 0\}$ and $x^\bullet := \{y \mid F(x, y) > 0\}$.

To simplify the notation, for a transition $t \in T$ we denote the variables read and written by t respectively by $\text{Var}_r(t)$ and $\text{Var}_w(t)$, as a shortcut for $\text{Var}(r(t))$ and $\text{Var}(w(t))$.

The execution semantics of DPNs extends the traditional one of place-transition nets by taking the case variables and transition guards into consideration. The state of a DPN combines two components:

- the classical notion of state of place-transition nets, that is, a marking that assigns tokens to the net places, implicitly indicating which transitions are enabled from the control-flow perspective;
- the state of the data component of the DPN, which assigns case variables to corresponding values, and implicitly indicates which transitions are enabled from the data perspective.

Definition 12 (*DPN State*)**.** A *state* of a DPN $D = (q, P, T, F, l, V, r, w)$ is a pair (M, α), where:

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

Consider a DPN $D = (q, P, T, F, l, V, r, w)$. Conceptually, a transition $t \in T$ is enabled in a given DPN state (M, α) if it is enabled from the control-flow point of view, namely M assigns at least one token to every input place of t, and it is enabled from the data perspective, namely its read guard is satisfied by the variable valuation α. When an enabled transition fires, it updates α by assigning to those variables mentioned in the write guard of t corresponding values that collectively satisfy the write guard. In general, many different combinations of values can be chosen, as long as they satisfy the write guard. In this respect, the write guard can be seen as a form of constrained user input associated with the transition. Upon firing the transition, it is in turn necessary to indicate which specific combination of values is chosen.

This calls for ensuring that transitions, labels, and guards are used consistently with the activities in $\mathcal{E}$. In particular, we aim at a correspondence between a sequence of transitions in a DPN and a corresponding trace over $\mathcal{E}$, in such a way that the trace provides all the necessary information to reconstruct which activities have been executed, and with which assigned values for the constrained user inputs. Towards this goal, we need to assume that every *visible* transition t of D corresponds to some activity $(n, q, A) \in \mathcal{E}$, in the following precise sense:

- the process reference q must either correspond to the name of D (hence configuring (n, q, A) as a local activity for D), or to global (hence indicating that (n, q, A) is a global activity);
- the activity name n matches the label $l(t)$ of the transition;
- the variables $\text{Var}(w(t))$ written by the transition exactly correspond to the attributes A.

The net progression is defined in terms of transition firing, which, in turn, can only happen if a selected transition is enabled. As opposed to classical Petri nets [22], a transition is *enabled* in our setting only if its read and write guards are satisfied under a given "firing mode" – a partial variable valuation function that assigns values only to variables of the guards – and all the input places of the transition contain sufficiently many tokens to consume. Given a DPN $D = (q, P, T, F, l, V, r, w)$ and $t \in T$, we call $\beta : V \nrightarrow \mathbb{R}$ a *partial valuation function of* t if β is defined on all variables $v \in \text{Var}_r(t) \cup \text{Var}_w(t)$. This is used to define the firing semantics of DPNs as follows.

Definition 13 (*Firing Rule*)**.** Consider a DPN $D = (q, P, T, F, l, V, r, w)$ and a transition $t \in T$. We say that t is *enabled* in state (M, α) under partial valuation $\beta : V \nrightarrow \mathbb{R}$, denoted $(M, \alpha)[t, \beta\rangle$, if:

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

If t is enabled in state (M, α) under β, it may *fire*, producing a new state (M', α'), written $(M, \alpha)[t, \beta\rangle(M', \alpha')$, with:

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

Notice that the last condition in the firing semantics captures inertia, that is, that variables not affected by the firing simply keep their current value.

We refer to the expression $(M, \alpha)[t, \beta\rangle(M', \alpha')$ as *transition firing*. We say that state (M', α') is *reachable* from (M, α), if there exists a sequence of transition firings from (M, α) to (M', α'). Importantly, for a well-formed DPN, a transition firing $(M, \alpha)[t, \beta\rangle(M', \alpha')$ corresponds to an event (n, q, ν), where:

- $n = l(t)$;
- q is the name of the DPN or global (depending on how n is unambiguously defined in $\mathcal{E}$);
- $\nu = \beta|_{\mathrm{Var}_w(t)}$.

By the same line of reasoning, a sequence of transition firings can be seen as a trace (as per Definition 8).

Since the monitoring task requires that we match events against transition firings of related DPNs, we introduce the notion of well-formedness. In the nutshell, it ensures that transitions, labels and guards are used consistently with the activities in $\mathcal{E}$, and thus allows to put in correspondence a sequence of transition firings and a trace over $\mathcal{E}$ in such a way that the trace provide all the necessary information to reconstruct which activities have been executed and which values were used for constrained user inputs (that is, assignments to variables in write guards).

Definition 14 (*Well-formed DPN*)**.** A DPN $D = (q, P, T, F, l, V, r, w)$ over $\mathcal{E}$ is *well-formed* if, for every $t \in T$, the following conditions are satisfied:

1. if $l(t) \neq \tau$, there exists $(n, q', A) \in \mathcal{E}$ such that
   (a) $q' = q$ or $q' =$ global;
   (b) $n = l(t)$;
   (c) $\mathrm{Var}_w(t) = A$.
2. if $l(t) = \tau$, then $w(t) \equiv \top$.

The last condition, which applies to silent transitions, ensures that they do not modify the current variable valuation, and hence they do not actually call for the existence of a corresponding event signature. This is important as silent transitions are not explicitly indicate in a trace of events, and consequently should not implicitly assign non-recorded values to the case variables of the DPN.

A.4. Declare with local filters

To capture declarative constraints over the different DPNs and the global activities, we adopt a data-aware variant of the the well-known Declare language [23] called LF-Declare (Declare with "local filters"). In particular, the data-aware extension we consider is in line with the notion of activity introduced before (this including a process reference and a set of attributes, beside the activity name), and with interval conditions specified over attributes. Process references are used in this context to capture:

- multi-process constraints that relate activities of different components;
- constraints involving global activities, thus regulating when and under which conditions they are expected to (not) occur.

Whereas the semantics of Declare is usually formalised using Linear Temporal Logic over finite traces (LTL_f) [8,24], to capture the semantics of LF-Declare we propose an extension of LTL_f with interval conditions as atomic formulas called LTL_f^{IC} (LTL over finite traces with interval conditions). Notice that models of LTL_f^{IC} specifications are traces of the form given in Definition 8.

We now provide syntax and semantics of LTL_f^{IC}. To this end, given an activity set $\mathcal{E}$, a process reference $q \in \mathcal{P}_\mathcal{E}$, and an activity name $n \in \mathcal{N}_\mathcal{E}$, we employ a dot notation $q.n$ to indicate the activity with reference q and name n defined in $\mathcal{E}$. Recall that, due to well-formedness, this activity is unique, and hence its attributes (which we refer as $A_\mathcal{E}(q.n)$) are univocally identified.

Definition 15. An LTL_f^{IC} *formula* over activity set $\mathcal{E}$ and data variables V^{d} is defined according to the following syntax:

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

where $q \in \mathcal{P}_\mathcal{E}$ is a process reference, $n \in \mathcal{N}_\mathcal{E}$ is an activity name, and φ is an interval condition over $A_\mathcal{E}(q.n)$.

Intuitively:

- $q.n \wedge \varphi$ is an activity with local filter, matching events that refer to $q.n$ and carry a payload that satisfies φ;
- as in LTL_f, $\mathbf{X}$ denotes the *strong next* operator, where $\mathbf{X}\Phi$ postulates the existence of a successor state in the trace, requiring that Φ holds there;
- as in LTL_f, $\mathbf{U}$ denotes the *(strong) until* operator, where $\Phi_1 \mathbf{U} \Phi_2$ postulates that there exists a future state in the trace where Φ_2 holds, and requires that Φ_1 holds in every intermediate state from the current state to that state.

We define, as customary in LTL_f, the usual abbreviations:

- booleans - $\Phi_1 \vee \Phi_2 \doteq \neg(\neg\Phi_1 \wedge \neg\Phi_2)$, and $\Phi_1 \rightarrow \Phi_2 \doteq \neg\Phi_1 \vee \Phi_2$;
- operator *eventually* - $\mathbf{F}\Phi \doteq \mathit{true}\mathbf{U}\Phi$, postulating that there must exist a future state in the trace where Φ holds;
- operator *globally* - $\mathbf{G}\Phi \doteq \neg\mathbf{F}\neg\Phi$, postulating that Φ holds in every state from the current one to the last one marking the end of the trace;
- operator *weak next* - $\bar{\mathbf{X}}\Phi \doteq \neg\mathbf{X}\neg\Phi$, postulating that if there exists a successor state in the trace (i.e., the current state is not the last one), then Φ must hold there;
- operator *weak until* - $\Phi_1 \mathbf{W} \Phi_2 \doteq (\Phi_1 \mathbf{U} \Phi_2) \vee \mathbf{G}\Phi_1$, postulating that Φ_1 holds until Φ_2 holds, or forever in case Φ_2 never holds in the future.

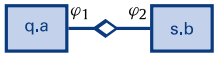
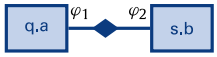
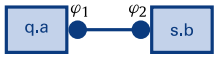
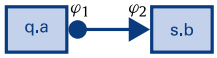
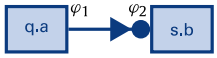
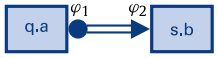
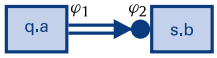
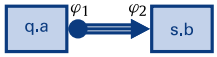
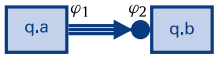
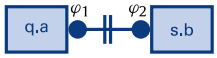
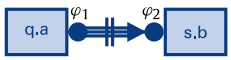
As already anticipate, the semantics of LTL_f^{IC} is given over finite traces of the form given in Definition 8.

Definition 16. We inductively define when an LTL_f^{IC} formula Φ is *satisfied* by a trace σ at position $1 \leq i \leq |\sigma|$, written $\sigma, i \models \Phi$, as follows:

- $\sigma, i \models \top$;
- $\sigma, i \models q.n$ if $\sigma(i)$ (extracting the event of σ in position i) returns an event (q', n', ν) with $q' = q$ and $n' = n$;
- $\sigma, i \models q.n \wedge \varphi$ if $\sigma, i \models q.n$ and, given $\sigma(i) = (n, q, \nu)$, we have that $\nu \models \varphi$ (as per Definition 10);
- $\sigma, i \models \Phi_1 \wedge \Phi_2$ iff $\sigma, i \models \Phi_1$ and $\sigma, i \models \Phi_2$;
- $\sigma, i \models \neg\Phi$ iff $\sigma, i \not\models \Phi_1$;
- $\sigma, i \models \mathbf{X}\Phi$ iff $i < |\sigma|$ and $\sigma, i+1 \models \Phi$;
- $\sigma, i \models \bar{\mathbf{X}}\Phi$ iff $i + 1 = |\sigma|$ or $\sigma, i+1 \models \Phi$;
- $\sigma, i \models \Phi_1 \mathbf{U} \Phi_2$ iff there exists j with $1 \leq j \leq |\sigma|$, such that $\sigma, j \models \Phi_2$, and $\sigma, k \models \Phi_1$ for every $1 \leq k < j$.

Starting from LTL_f^{IC} and the standard templates of (propositional) Declare, we show in Table A.2 the full set of LF-Declare template patterns, each of which comes with its name, graphical representation, and formalisation in LTL_f^{IC}. Notice that template parameters now refer to activities with process references and local filters.

Table A.2
LF-DECLARE constraint patterns.

existence($q.a \wedge \varphi$)		$\mathbf{F}(q.a \wedge \varphi)$
ⓘ an event matching $q.a \wedge \varphi$ must occur at least once		
absence($q.a \wedge \varphi$)		$\neg\mathbf{F}(q.a \wedge \varphi)$
ⓘ no event matching $q.a \wedge \varphi$ can occur at all		
absence2($q.a \wedge \varphi$)		$\neg\mathbf{F}((q.a \wedge \varphi) \wedge \mathbf{XF}(q.a \wedge \varphi))$
ⓘ at most one event matching $q.a \wedge \varphi$ can occur		
choice($q.a \wedge \varphi_1$,$s.b \wedge \varphi_2$)		$\mathbf{F}((q.a \wedge \varphi_1) \vee (q.b \wedge \varphi_2))$
ⓘ at least one event matching $q.a \wedge \varphi_1$ or $s.b \wedge \varphi_2$ must occur		
ex-choice($q.a \wedge \varphi_1$,$s.b \wedge \varphi_2$)		$\mathbf{F}((q.a \wedge \varphi_1) \vee (s.b \wedge \varphi_2)) \wedge$ $\neg(\mathbf{F}(q.a \wedge \varphi_1) \wedge \mathbf{F}(q.b \wedge \varphi_2))$
ⓘ at least one event matching either $q.a \wedge \varphi_1$ or $s.b \wedge \varphi_2$ must occur		
resp-existence ($q.a \wedge \varphi_1$,$s.b \wedge \varphi_2$)		$\mathbf{F}(q.a \wedge \varphi_1) \rightarrow \mathbf{F}(s.b \wedge \varphi_2)$
ⓘ if an event matching $q.a \wedge \varphi_1$ occurs, then an event matching $s.b \wedge \varphi_2$ must also occur (either before or later)		
coexistence($q.a \wedge \varphi_1$,$s.b \wedge \varphi_2$)		$(\mathbf{F}(q.a \wedge \varphi_1) \rightarrow \mathbf{F}(s.b \wedge \varphi_2)) \wedge$ $(\mathbf{F}(s.b \wedge \varphi_2) \rightarrow \mathbf{F}(q.a \wedge \varphi_1))$
ⓘ either events matching $q.a \wedge \varphi_1$ and $s.b \wedge \varphi_2$ occur, or none of them occurs		
response($q.a \wedge \varphi_1$,$s.b \wedge \varphi_2$)		$\mathbf{G}(q.a \wedge \varphi_1 \rightarrow \mathbf{XF}(s.b \wedge \varphi_2))$
ⓘ whenever an event matching $q.a \wedge \varphi_1$ occurs, then an event matching $s.b \wedge \varphi_2$ must occur later		
precedence($q.a \wedge \varphi_1$,$s.b \wedge \varphi_2$)		$\neg(s.b \wedge \varphi_2)\mathbf{W}(q.a \wedge \varphi_1)$
ⓘ an event matching $s.b \wedge \varphi_2$ can only occur if an event matching $q.a \wedge \varphi_1$ has already occurred		
alt-response($q.a \wedge \varphi_1$,$s.b \wedge \varphi_2$)		$\mathbf{G}((q.a \wedge \varphi_1) \rightarrow$ $\mathbf{X}(\neg(q.a \wedge \varphi_1)\mathbf{U}(s.b \wedge \varphi_2)))$
ⓘ every event matching $q.a \wedge \varphi_1$ must be followed by an event matching $s.b \wedge \varphi_2$, without any occurrence of other events matching $q.a \wedge \varphi_1$ in between		
alt-precedence($q.a \wedge \varphi_1$,$s.b \wedge \varphi_2$)		$(\neg(s.b \wedge \varphi_2))\mathbf{W}(q.a \wedge \varphi_1) \wedge$ $\mathbf{G}((s.b \wedge \varphi_2) \rightarrow$ $\bar{\mathbf{X}}((\neg(s.b \wedge \varphi_2))\mathbf{W}(q.a \wedge \varphi_1)))$
ⓘ every event matching $s.b \wedge \varphi_2$ must be preceded by an event matching $q.a \wedge \varphi_1$, without any other occurrence of events matching $s.b \wedge \varphi_2$ in between		
chain-response($q.a \wedge \varphi_1$,$s.b \wedge \varphi_2$)		$\mathbf{G}((q.a \wedge \varphi_1) \rightarrow \mathbf{X}(s.b \wedge \varphi_2))$
ⓘ every event matching $q.a \wedge \varphi_1$ must be immediately followed by an event matching $s.b \wedge \varphi_2$		
chain-precedence($q.a \wedge \varphi_1$,$s.b \wedge \varphi_2$)		$\mathbf{G}(\mathbf{X}(s.b \wedge \varphi_2) \rightarrow q.a \wedge \varphi_1)$
ⓘ events matching $s.b \wedge \varphi_2$ can only occur immediately after events matching $q.a \wedge \varphi_1$ occur		
not-coexistence($q.a \wedge \varphi_1$,$s.b \wedge \varphi_2$)		$\neg(\mathbf{F}(q.a \wedge \varphi_1) \wedge \mathbf{F}(s.b \wedge \varphi_2))$
ⓘ two events respectively matching $q.a \wedge \varphi_1$ and $s.b \wedge \varphi_2$ cannot both occur		
neg-succession($q.a \wedge \varphi_1$,$s.b \wedge \varphi_2$)		$\mathbf{G}((q.a \wedge \varphi_1) \rightarrow \neg\mathbf{F}(s.b \wedge \varphi_2))$
ⓘ two events respectively matching $q.a \wedge \varphi_1$ and $s.b \wedge \varphi_2$ cannot occur one after the other		
neg-chain-succ.($q.a \wedge \varphi_1$,$s.b \wedge \varphi_2$)		$\mathbf{G}((q.a \wedge \varphi_1) \rightarrow \bar{\mathbf{X}}\neg(s.b \wedge \varphi_2))$
ⓘ two events respectively matching $q.a \wedge \varphi_1$ and $s.b \wedge \varphi_2$ cannot occur next to each other		

References

[1] C. Di Ciccio, A. Marrella, A. Russo, Knowledge-intensive processes: Characteristics, requirements and analysis of contemporary approaches, J. Data Semant. 4 (1) (2015) 29–57.

[2] M. Reichert, B. Weber, Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies, Springer, 2012.

[3] W.M.P. van der Aalst, Object-centric process mining: Dealing with divergence and convergence in event data, in: P.C. Ölveczky, G. Salaün (Eds.), Proceedings of the 17th International Conference on Software Engineering and Formal Methods, SEFM 2019, in: Lecture Notes in Computer Science, vol. 11724, Springer, 2019, pp. 3–25.

[4] A. Artale, A. Kovtunova, M. Montali, W.M.P. van der Aalst, Modeling and reasoning over declarative data-aware processes with object-centric behavioral constraints, in: T.T. Hildebrandt, B.F. van Dongen, M. Röglinger, J. Mendling (Eds.), Proceedings of the 17th International Conference on Business Process Management, BPM 2019, in: Lecture Notes in Computer Science, vol. 11675, Springer, 2019, pp. 139–156.

[5] D. Fahland, Multi-dimensional process analysis, in: C. Di Ciccio, R.M. Dijkman, A. del-Río-Ortega, S. Rinderle-Ma (Eds.), Proceedings of the 20th International Conference on Business Process Management, BPM 2022, in: Lecture Notes in Computer Science, vol. 13420, Springer, 2022, pp. 27–33.

[6] A. Bottrighi, F. Chesani, P. Mello, M. Montali, S. Montani, P. Terenziani, Conformance checking of executed clinical guidelines in presence of basic medical knowledge, in: BPM Workshops, 2011, pp. 200–211.

[7] F.M. Maggi, M. Montali, M. Westergaard, W.M.P. van der Aalst, Monitoring business constraints with linear temporal logic: An approach based on colored automata, in: BPM, in: LNCS, vol. 6896, Springer, 2011, pp. 132–147.

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

[9] A. Alman, F.M. Maggi, M. Montali, F. Patrizi, A. Rivkin, Multi-model monitoring framework for hybrid process specifications, in: CAiSE, in: Lecture Notes in Computer Science, vol. 13295, Springer, 2022, pp. 319–335.

[10] A. Alman, F.M. Maggi, M. Montali, F. Patrizi, A. Rivkin, Monitoring hybrid process specifications with conflict management: An automata-theoretic approach, Artif. Intell. Med. 139 (2023) 102512, http://dx.doi.org/10.1016/j.artmed.2023.102512.

[11] M. Spiotta, P. Terenziani, D. Theseider Dupré, Temporal conformance analysis and explanation of clinical guidelines execution: An answer set programming approach, IEEE Trans. Knowl. Data Eng. 29 (11) (2017) 2567–2580.

[12] A. Jalali, F.M. Maggi, H.A. Reijers, A hybrid approach for aspect-oriented business process modeling, J. Softw. Evol. Process. 30 (8) (2018).

[13] M. de Leoni, P. Felli, M. Montali, A holistic approach for soundness verification of decision-aware process models, in: ER, in: LNCS, vol. 11157, Springer, 2018, pp. 219–235.

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

[15] R. De Masellis, F.M. Maggi, M. Montali, Monitoring data-aware business constraints with finite state automata, in: ICSSP, ACM, 2014, pp. 134–143.

[16] F.M. Maggi, M. Montali, U. Bhat, Compliance monitoring of multi-perspective declarative process models, in: EDOC, IEEE, 2019, pp. 151–160.

[17] G. De Giacomo, M.Y. Vardi, Linear temporal logic and linear dynamic logic on finite traces, in: F. Rossi (Ed.), Proceedings of the 23rd International Joint Conference on Artificial Intelligence, IJCAI 2013, IJCAI/AAAI, 2013, pp. 854–860.

[18] P. Felli, M. Montali, S. Winkler, Linear-time verification of data-aware dynamic systems with arithmetic, in: Proceedings of the 36th AAAI Conference on Artificial Intelligence, AAAI 2022, AAAI Press, 2022, pp. 5642–5650.

[19] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction To Automata Theory, Languages, and Computation, third ed., Addison-Wesley, 2007.

[20] C. Di Ciccio, M. Montali, Declarative process specifications: Reasoning, discovery, monitoring, in: W.M.P. van der Aalst, J. Carmona (Eds.), Process Mining Handbook, in: Lecture Notes in Business Information Processing, vol. 448, Springer, 2022, pp. 108–152, http://dx.doi.org/10.1007/978-3-031-08848-3_4.

[21] B.F. van Dongen, J. De Smedt, C. Di Ciccio, J. Mendling, Conformance checking of mixed-paradigm process models, Inf. Syst. 102 (2021).

[22] T. Murata, Petri nets: Properties, analysis and applications, Proc. IEEE 77 (4) (1989) 541–580, http://dx.doi.org/10.1109/5.24143.

[23] M. Pesic, H. Schonenberg, W.M.P. van der Aalst, DECLARE: full support for loosely-structured processes, in: EDOC, IEEE Computer Society, 2007, pp. 287–300.

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

[25] M. Montali, F. Chesani, P. Mello, F.M. Maggi, Towards data-aware constraints in declare, in: S.Y. Shin, J.C. Maldonado (Eds.), Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, ACM, 2013, pp. 1391–1396.

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

[27] F.M. Maggi, M. Montali, U. Bhat, Compliance monitoring of multi-perspective declarative process models, in: EDOC, IEEE, 2019, pp. 151–160.

[28] D. Calvanese, G. De Giacomo, M. Montali, F. Patrizi, Verification and monitoring for first-order LTL with persistence-preserving quantification over finite and infinite traces, in: L. De Raedt (Ed.), Proceedings of the 31st International Joint Conference on Artificial Intelligence, IJCAI 2022, ijcai.org, 2022, pp. 2553–2560.

[29] M. Leucker, C. Schallhart, A brief account of runtime verification, J. Log. Algebraic Methods Program 78 (5) (2009) 293–303.

[30] W. Reisig, Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies, Springer, 2013.

[31] M. Hewelt, M. Weske, A hybrid approach for flexible case modeling and execution, in: M. La Rosa, P. Loos, O. Pastor (Eds.), Proceedings of the Business Process Management Forum, BPM Forum 2016, in: Lecture Notes in Business Information Processing, vol. 260, Springer, 2016, pp. 38–54.

[32] S. Haarmann, M. Montali, M. Weske, Refining case models using cardinality constraints, in: M. La Rosa, S.W. Sadiq, E. Teniente (Eds.), Proceedings of the 33rd International Conference on Advanced Information Systems Engineering, CAiSE 2021, in: Lecture Notes in Computer Science, vol. 12751, Springer, 2021, pp. 296–310.

[33] G. Bergami, F.M. Maggi, A. Marrella, M. Montali, Aligning data-aware declarative process models and event logs, in: BPM, 2021, pp. 235–251.

[34] J. De Smedt, J. De Weerdt, J. Vanthienen, Fusion Miner: Process discovery for mixed-paradigm models, Decis. Support Syst. 77 (2015) 123–136.

[35] F. Mannhardt, Sepsis Cases - Event Log, Eindhoven University of Technology, 2016, http://dx.doi.org/10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460, URL https://data.4tu.nl/articles/dataset/Sepsis_Cases_-_Event_Log/12707639/1.

[36] F. Mannhardt, Multi-Perspective Process Mining (Ph.D. thesis), Technical University of Eindhoven, 2018.

[37] B.F. van Dongen, BPI Challenge 2017, Eindhoven University of Technology, 2017, http://dx.doi.org/10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b, URL https://data.4tu.nl/articles/dataset/BPI_Challenge_2017/12696884/1.

[38] M. Fani Sani, H. Sotudeh, Business understanding using process mining, 2017, BPI Challenge, URL https://www.win.tue.nl/bpi/2017/bpi2017_paper_25.pdf.

[39] E. Povalyaeva, I. Khamitov, A. Fomenko, BPIC 2017: Density analysis of the interaction with clients, 2017, BPI Challengev, URL https://www.win.tue.nl/bpi/2017/bpi2017_winner_student.pdf.

[40] A. Alman, C. Di Ciccio, D. Haas, F.M. Maggi, A. Nolte, Rule mining with RuM, in: ICPM, IEEE, 2020, pp. 121–128.

[41] M. La Rosa, H.A. Reijers, W.M.P. van der Aalst, R.M. Dijkman, J. Mendling, M. Dumas, L. García-Bañuelos, APROMORE: An advanced process model repository, Expert Syst. Appl. 38 (6) (2011) 7029–7040.

[42] C. Di Ciccio, F.M. Maggi, J. Mendling, Discovering target-branched declare constraints, in: BPM, in: Lecture Notes in Computer Science, vol. 8659, Springer, 2014, pp. 34–50.

[43] D. Calvanese, G. De Giacomo, M. Montali, F. Patrizi, Verification and monitoring for first-order LTL with persistence-preserving quantification over finite and infinite traces, in: L. De Raedt (Ed.), Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022, ijcai.org, 2022, pp. 2553–2560, http://dx.doi.org/10.24963/ijcai.2022/354.

[44] J. Prock, A new technique for fault detection using Petri nets, Automatica 27 (2) (1991) 239–245.

[45] A. Pla, P. Gay, J. Meléndez, B. López, Petri net-based process monitoring: a workflow management system for process modelling and monitoring, J. Intell. Manuf. 25 (3) (2014) 539–554.

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

[47] R. De Masellis, F.M. Maggi, M. Montali, Monitoring data-aware business constraints with finite state automata, in: ICSSP, ACM, 2014, pp. 134–143.

[48] J. Carmona, B.F. van Dongen, A. Solti, M. Weidlich, Conformance Checking - Relating Processes and Models, Springer, 2018, http://dx.doi.org/10.1007/978-3-319-99414-7.

[49] P. Felli, A. Gianola, M. Montali, A. Rivkin, S. Winkler, CoCoMoT: Conformance checking of multi-perspective processes via SMT, in: BPM, 2021, pp. 217–234.

[50] F. Chesani, P. Mello, M. Montali, F. Riguzzi, M. Sebastianis, S. Storari, Checking compliance of execution traces to business rules, in: D. Ardagna, M. Mecella, J. Yang (Eds.), Business Process Management Workshops, BPM 2008 International Workshops, Milano, Italy, September 1–4, 2008. Revised Papers, in: Lecture Notes in Business Information Processing, vol. 17, Springer, 2008, pp. 134–145, http://dx.doi.org/10.1007/978-3-642-00328-8_13.

[51] W.M.P. van der Aalst, Object-centric process mining: Dealing with divergence and convergence in event data, in: P.C. Ölveczky, G. Salaün (Eds.), Proc. of SEFM 2019, in: Lecture Notes in Computer Science, vol. 11724, Springer, 2019, pp. 3–25, http://dx.doi.org/10.1007/978-3-030-30446-1_1.

[52] A. Artale, A. Kovtunova, M. Montali, W.M.P. van der Aalst, Modeling and reasoning over declarative data-aware processes with object-centric behavioral constraints, in: Proc. of BPM 2019, in: LNCS, vol. 11675, Springer, 2019, pp. 139–156.

[53] M. Montali, D. Calvanese, Soundness of data-aware, case-centric processes, Int. J. Softw. Tools Technol. Transf. (2016) http://dx.doi.org/10.1007/s10009-016-0417-2.

[54] D. Fahland, Describing behavior of processes with many-to-many interactions, in: Proc. of Petri Nets 2019, Springer, 2019, pp. 3–24.

[55] A. Polyvyanyy, J.M.E.M. van der Werf, S. Overbeek, R. Brouwers, Information systems modeling: Language, verification, and tool support, in: Proc. of CAiSE 2019, in: LNCS, vol. 11483, Springer, 2019, pp. 194–212, http://dx.doi.org/10.1007/978-3-030-21290-2_13.

[56] S. Ghilardi, A. Gianola, M. Montali, A. Rivkin, Petri nets with parameterised data - Modelling and verification, in: D. Fahland, C. Ghidini, J. Becker, M. Dumas (Eds.), BPM'20, in: LNCS, vol. 12168, Springer, 2020, pp. 55–74, http://dx.doi.org/10.1007/978-3-030-58666-9_4.

[57] F. Rosa-Velardo, D. de Frutos-Escrig, Decidability and complexity of Petri nets with unordered data, Theoret. Comput. Sci. 412 (34) (2011) 4439–4451.

[58] K.M. van Hee, N. Sidorova, M. Voorhoeve, J.M.E.M. van der Werf, Generation of database transactions with Petri nets, Fund. Inform. 93 (1–3) (2009) 171–184.

[59] W.M.P. van der Aalst, A. Berti, Discovering object-centric Petri nets, Fund. Inform. 175 (1–4) (2020) 1–40, http://dx.doi.org/10.3233/FI-2020-1946.

[60] D. Barenholz, M. Montali, A. Polyvyanyy, H.A. Reijers, A. Rivkin, J.M.E.M. van der Werf, There and back again - On the reconstructibility and rediscoverability of typed jackson nets, in: Proc. of PETRI NETS 2023, in: Lecture Notes in Computer Science, 13929, Springer, 2023, pp. 37–58.

[61] S.W. Sadiq, M.E. Orlowska, W. Sadiq, Specification and validation of process constraints for flexible workflows, Inf. Syst. 30 (5) (2005) 349–378.

[62] T. Slaats, D.M.M. Schunselaar, F.M. Maggi, H.A. Reijers, The semantics of hybrid process models, in: C. Debruyne, H. Panetto, R. Meersman, T.S. Dillon, E. Kühn, D. O'Sullivan, C.A. Ardagna (Eds.), On the Move To Meaningful Internet Systems: OTM 2016 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2016, Rhodes, Greece, October 24-28, 2016, Proceedings, in: Lecture Notes in Computer Science, vol. 10033, 2016, pp. 531–551, http://dx.doi.org/10.1007/978-3-319-48472-3_32.

[63] F.M. Maggi, T. Slaats, H.A. Reijers, The automated discovery of hybrid processes, in: BPM, in: LNCS, vol. 8659, Springer, 2014, pp. 392–399.

[64] J. De Smedt, J. De Weerdt, J. Vanthienen, G. Poels, Mixed-paradigm process modeling with intertwined state spaces, Bus. Inf. Syst. Eng. 58 (1) (2016) 19–29, http://dx.doi.org/10.1007/s12599-015-0416-y.

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

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

[67] P. Felli, M. Montali, F. Patrizi, S. Winkler, Monitoring arithmetic temporal properties on finite traces, in: Proceedings of the 37th AAAI Conference on Artificial Intelligence, AAAI 2023, AAAI Press, 2023, pp. 6346–6354.