Declarative Process Modeling in BPMN

Giuseppe De Giacomo[1], Marlon Dumas[2],
Fabrizio Maria Maggi[2], and Marco Montali[3]

[1] Sapienza Università di Roma, Italy
`degiacomo@dis.uniroma1.it`
[2] University of Tartu, Estonia.
`{f.m.maggi, marlon.dumas}@ut.ee`
[3] Free University of Bozen-Bolzano, Italy.
`montali@inf.unibz.it`

Abstract. Traditional business process modeling notations, including the standard Business Process Model and Notation (BPMN), rely on an imperative paradigm wherein the process model captures all allowed activity flows. In other words, every flow that is not specified is implicitly disallowed. In the past decade, several researchers have exposed the limitations of this paradigm in the context of business processes with high variability. As an alternative, declarative process modeling notations have been proposed (e.g., Declare). These notations allow modelers to capture constraints on the allowed activity flows, meaning that all flows are allowed provided that they do not violate the specified constraints. Recently, it has been recognized that the boundary between imperative and declarative process modeling is not crisp. Instead, mixtures of declarative and imperative process modeling styles are sometimes preferable, leading to proposals for hybrid process modeling notations. These developments raise the question of whether completely new notations are needed to support hybrid process modeling. This paper answers this question negatively. The paper presents a conservative extension of BPMN for declarative process modeling, namely BPMN-D, and shows that Declare models can be transformed into readable BPMN-D models.

Keywords: BPMN, declarative process modeling, Declare

1 Introduction

The standard Business Process Model and Notation (BPMN) [13] and related approaches rely on an imperative paradigm wherein the process model captures all allowed activity flows. Underpinning these notations is a "closed world" assumption, meaning that the process model captures all possible activity flows and hence any unspecified activity flow is disallowed. This paradigm has proved suitable in the context of regular and predictable processes, where there is in essence one primary way of performing a process, with relatively few and well-scoped variations.

In the past decade, several researchers have exposed the limitations of this imperative paradigm in the context of business processes with high variability, such as customer lead management processes, product design processes, patient treatment and related healthcare processes [19, 17]. As an alternative, declarative process modeling notations have been proposed, e.g., Declare [14, 1], Guard-Stage-Milestones (GSM) [9]

and the Case Management Model and Notation (CMMN) [11]. Unlike their imperative counterparts, a declarative model captures a process under an "open world" assumption, such that everything is allowed unless it is explicitly forbidden by a rule. In this context, a rule may take the form of a binary relation between pairs of tasks that must be satisfied in every execution of a process, like for example "task B can only be performed if task A has been previously performed in the same case".

More recently, it has been recognized that the boundary between imperative and declarative process modeling is not crisp. Instead, mixtures of declarative and imperative modeling styles are sometimes preferable, leading to proposals for hybrid process modeling notations [21, 6]. These developments raise the question of whether completely new notations are needed to support hybrid process modeling.

Given this question, this paper analyzes the possibility of seamlessly extending BPMN with declarative constructs. The main contribution of the paper is an extension of BPMN, namely BPMN-D. BPMN-D is a conservative extension in the sense that it only adds constructs, such that any BPMN model is a BPMN-D model. Furthermore, BPMN-D is a macro-extension, i.e., it is designed so that any BPMN-D model can be translated into a (larger and potentially less readable) BPMN model. The paper also shows that any Declare model can be translated into a readable BPMN-D model via constraint automata. More generally, any declarative process modeling language defined in terms of Linear Temporal Logic over finite traces (LTL_f) can be translated into BPMN-D using the proposed translation method.

The paper is structured as follows. Section 2 provides an overview of declarative process modeling – specifically the Declare notation – and discusses previous research on linking declarative and imperative process modeling approaches. Section 3 introduces the BPMN-D notation and shows how the extended constructs of BPMN-D can be re-written into standard BPMN. Next, Section 4 outlines the translation from Declare to BPMN-D. Finally, Section 5 draws conclusions and outlines future work.
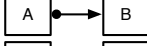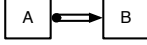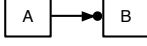
2 Background and Related Work

Declare [14, 1] is a declarative process modeling language wherein a process is specified via a set of constraints between activities, which must be satisfied by every execution of the process. Declare constraints are captured based on templates. Templates are patterns that define parameterized classes of properties, while constraints are their concrete instantiations. Herein, we write template parameters in upper-case and concrete activities in their instantiations in lower-case. Constraints have a graphical representation. The semantics of templates can be formalized using different logics [12], for example LTL_f.

Table 1 summarizes some Declare templates and their corresponding formalization in LTL_f. (The reader can refer to [1] for a full description of the language.) The $\Diamond$, $\bigcirc$, $\Box$, and $\sqcup$ LTL_f operators have the following intuitive meaning: formula $\Diamond\phi_1$ means that ϕ_1 holds sometime in the future, $\bigcirc\phi_1$ means that ϕ_1 holds in the next position, $\Box\phi_1$ says that ϕ_1 holds forever, and, lastly, $\phi_1 \sqcup \phi_2$ means that sometime in the future ϕ_2 will hold and until that moment ϕ_1 holds (with ϕ_1 and ϕ_2 LTL_f formulas).

Consider, for example, the *response* constraint $\Box(a \rightarrow \Diamond b)$. This constraint indicates that if *a occurs*, *b* must eventually *follow*. Therefore, this constraint is satisfied

Table 1: Semantics for some Declare templates

Template	LTL$_f$ **semantics**	**Activation**
responded existence	$\Diamond A \rightarrow \Diamond B$	A •——— B
response	$\Box(A \rightarrow \Diamond B)$	A •——→ B
alternate response	$\Box(A \rightarrow \bigcirc(\neg A \sqcup B))$	A •⇐= B
chain response	$\Box(A \rightarrow \bigcirc B)$	A •⟺ B
precedence	$(\neg B \sqcup A) \vee \Box(\neg B)$	A →• B
alternate precedence	$(\neg B \sqcup A) \vee \Box(\neg B) \wedge$ $\Box(B \rightarrow \bigcirc((\neg B \sqcup A) \vee \Box(\neg B)))$	A ⇒• B
chain precedence	$\Box(\bigcirc B \rightarrow A)$	A ⟹• B

for traces such as $\mathbf{t}_1 = \langle a, a, b, c \rangle$, $\mathbf{t}_2 = \langle b, b, c, d \rangle$, and $\mathbf{t}_3 = \langle a, b, c, b \rangle$, but not for $\mathbf{t}_4 = \langle a, b, a, c \rangle$ because, in this case, the second occurrence of a is not followed by an occurrence of b. A constraint can define more than one activity for each parameter specified in its template. In this case, we say that the parameters *branch out* and, in the graphical representation, they are replaced by multiple arcs to all branched activities. In the LTL$_f$ semantics, the parameters are replaced by a disjunction of branching activities. For example, the LTL$_f$ semantics of the response template with two branches on the target parameter is $\Box(A \rightarrow \Diamond(B \vee C))$.

Example 1. Consider the Declare model that represents a fragment of a purchase order process, as shown in Figure 1. The process is as follows:
- a payment cannot be done until the order is closed (*precedence* constraint);
- whenever a payment is done, then a receipt or an invoice must be produced (*branching response* constraint).
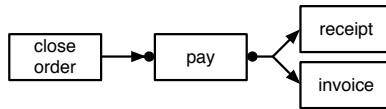

Fig. 1: Example of a Declare model

Like any declarative model, this model should be interpreted according to an "open-world" semantics: It is possible to send a receipt or an invoice without paying beforehand and, also, to close an order without eventually paying. In addition, an activity in the model can be executed several times. Closing an order several times has no effect on the process execution, whereas it is possible to pay several times (this is the case, for example, of installments) and, also, to send invoices and receipts several times.

Besides Declare, other declarative process modeling notations include Condition Response Graphs (DCR) graphs [8] and GSM [9]. DCR graphs rely on binary relations between tasks (as in Declare) but employ a smaller set of five core relations and support decomposition (nesting). GSM differs from Declare and DCR in that it does not rely on binary relations. Instead it relies on three core concepts: guard, stage and milestone. A stage is a phase in the execution of a process where a certain number of tasks (or other stages) may occur in any order and any number of times (similar to *ad hoc* activities in BPMN). The opening of a stage is subject to one of its guards (event-condition rules) becoming true. The stage is closed when one of its milestones is achieved (i.e., becomes true). A milestone is also defined by means of an event-condition rule. The guards of a stage may refer to data associated to the process and/or to the status of other stages or milestones (e.g., whether a given stage is currently "open" or "closed"). Tasks are modeled as atomic stages and may have their own guards and milestone(s). Several concepts proposed in the GSM notation have made their way into the CMMN standard [11].

Initially, declarative process modeling notations were proposed as alternatives to imperative ones. Recent research though has put into evidence synergies between these two approaches [15, 18]. Accordingly, hybrid process modeling notations have been proposed. In particular, [21] proposes to extend Colored Petri nets with the possibility of linking transitions to Declare constraints (in the same model). The notion of *transition enablement* is extended to handle declarative links between transitions. Meanwhile, [10, 6] combine imperative and declarative styles in the context of automated discovery of process models from event logs. Specifically, the approach in [10] discovers hierarchical models with sub-processes that can be either imperative (Petri nets) or declarative (Declare constraints). Meanwhile, the method in [6] discovers a process model with two types of arcs: imperative (sequence flows) and declarative (Declare constraints).

Another contribution that bridges declarative and imperative process modeling styles is [16], which proposes a translation from Declare to Petri nets. The idea is to first produce a Finite State Machine (FSM) from a Declare model using standard techniques for mapping regular expressions to automata. The FSM is then mapped into a sequential Petri net and methods related to "theory of regions" [3] are used to rewrite the Petri net so that parallelism is explicitly captured. The resulting Petri net can then be mapped into other imperative process modeling notations (e.g., BPMN). A drawback of this approach is that the resulting Petri net is large and complex relative to the initial declarative model. This drawback is illustrated in [16] where a Declare model with 4 tasks and 7 constraints leads to an FSM with 10 states and 24 arcs. This FSM in turn leads to a Petri net with 25 transitions, 13 places and over 40 arcs. The issue at stake is that a given constraint may be satisfied by a large number of distinct possible execution paths. Capturing these paths in an imperative style leads to significant amounts of task duplication (e.g., transitions with duplicate labels in the case of Petri nets).

3 BPMN-D

This section introduces the BPMN-D notation and gives it a semantics by means of a translation from BPMN-D to plain BPMN, for which different formal semantics have been specified in previous work [7].

3.1 Overview

BPMN-D is an extension of BPMN partly inspired by BPMN-Q [2] – a language previously proposed to capture queries over collections of BPMN models. Like BPMN-Q, BPMN-D is a conservative (additive) extension of BPMN, implying that any BPMN model is also a BPMN-D model. Fig. 2 shows an example of a BPMN-D model. In particular, a BPMN-D model may have *start* and *end event nodes*, with the same semantics as in standard BPMN. For instance, in Fig. 2, there is one start event and two end events. Similarly, *behavioral XOR split/join* represents (exclusive) alternative behaviors that are allowed during the process execution, following the standard BPMN semantics of *deferred choice* (i.e., choice freely taken by the resources responsible for the process execution). As shown in Fig. 2, the graphical representation for a XOR gateway is the same as in BPMN. In this paper, we only discuss XOR gateways as they are sufficient to demonstrate the extensions proposed in BPMN-D and the translation from Declare to BPMN-D. In the remainder of the paper, we denote by Σ the set of all tasks that can be performed in a given business context.
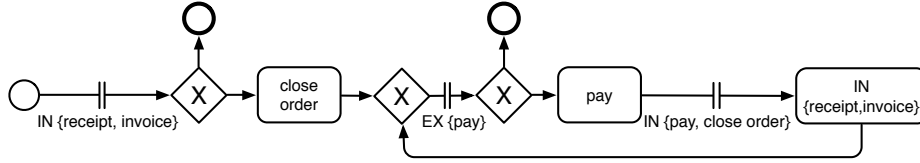

Fig. 2: Example of a BPMN-D model

BPMN-D extends only two constructs in BPMN, namely activity nodes and sequence flows connectors. An *activity node* represents a task in the process, and is represented as a labeled, rounded rectangle. As in standard BPMN, this in turn corresponds to an execution step inside the process. Differently from BPMN, though, a BPMN-D activity node can be labeled not only with a single task name $t \in \Sigma$, but with a set T of multiple tasks, such that T is nonempty and does not coincide with Σ. When the label denotes a single task t, the semantics coincides with that of BPMN: the activity node is executed whenever t is performed. When the label is instead a set T of tasks, the activity node is considered to be executed whenever a task t is executed, such that either $t \in T$ (*inclusive task*), or $t \notin T$ (*exclusive task*). To distinguish between these two cases, a set-labeled BPMN-D task is also annotated with a property IN or EX, so as to indicate whether the task is inclusive or exclusive. For example, in Fig. 2, the *pay*-labeled activity node indicates that a payment must be done, whereas the set-labeled activity node IN($\{receipt, invoice\}$) indicates that one task between *receipt* or *invoice* must be performed. In addition to these three types of activity nodes, we consider also the case in which the process expects participants to do "something", that is, to engage in an execution step by freely choosing a task from the global set Σ. In this case, we assume that the activity node is just labeled with label ANY. The different BPMN-D activity nodes are summarized in Table 2.

Notation	Name	Semantics
$\boxed{t}$	Atomic task	As in BPMN: perform t
$\boxed{\begin{smallmatrix}\text{IN}\\ \{t_1,\ldots,t_n\}\end{smallmatrix}}$	Inclusive task	Perform a task among $t_1,\ldots,t_n$
$\boxed{\begin{smallmatrix}\text{EX}\\ \{t_1,\ldots,t_n\}\end{smallmatrix}}$	Exclusive task	Perform a task different from $t_1,\ldots,t_n$
$\boxed{\text{ANY}}$	Any task	Perform any task from those available in the business context

Table 2: Overview of BPMN-D activity nodes

A *flow connector* is a binary, directed relation between nodes in the process. It indicates an ordering relationship between the connected nodes, and implicitly also the state of the process when the process has traversed the source node but has still to traverse the destination node. Differently from BPMN, in BPMN-D sequence-flow connectors do not only represent a direct ordering relationship (stating that the destination node comes next to the source one), but also a "loose" ordering relationship, which indicates that the destination node will be traversed *after* the source one, but that other BPMN-D tasks can be performed in between. First of all, BPMN-D supports the ordinary BPMN sequence flow, adopting its semantics and notation (a solid arrow from the source to the destination node). Loose flow connectors are instead visually depicted as an interrupted solid arrow, and their specific semantics is defined by labeling and annotating the connector with additional information, similarly to the case of BPMN-D activity nodes. As summarized in Table 3, three loose connectors are supported. The first two, namely *inclusive flow* and *exclusive flow*, label the flow connectors with a set of T of tasks, and indicate that while moving from the source to the destination node along the flow connector, 0 or more repetitions of tasks respectively from or not in T can be executed. To distinguish between the two cases, the set T is also annotated with a property IN or EX, so as to indicate whether the flow connector is inclusive or exclusive. For example, in Fig. 2, the first flow connector indicates that when the process starts, 0 or more repetitions of tasks *receipt* and *invoice* may occur while moving to the first decision point. In addition, we consider also the case in which while moving from a node to another node, 0 or more repeatitions of any task may occur. In this case, we assume that the flow connector is just labeled with label ANY, and we consequently call it *any flow*.

3.2 BPMN-D Models

We now turn to the formal definition of BPMN-D model, substantiating the overview of the previous section. Given a set Σ of tasks, a BPMN-D model $\mathcal{M}$ is a tuple $\langle N, type_N, \ell_N, F, type_F, \ell_F \rangle$, where:

- N is a finite set of nodes, partitioned into activity nodes, event nodes, and gateways.
- $type_N$ is a total function from N to a finite set of *node types*; the following types are considered in this paper:
    - ATOMIC-TASK, IN-TASK, EX-TASK, and ANY-TASK for activity nodes (cf. Table 2);

Notation	Name	Semantics
A ➤ B	Sequence flow	As in BPMN: node B is traversed next to A
A IN {t₁,...,tₙ} ➤ B	Inclusive flow	B is traversed after A, with 0 or more repetitions of tasks from $t_1, \ldots, t_n$ in between
A EX {t₁,...,tₙ} ➤ B	Exclusive flow	B is traversed after A, with 0 or more repetitions of tasks different from $t_1, \ldots, t_n$ in between
A ANY ➤ B	Any flow	B is traversed after A, with 0 or more repetitions of tasks in between

Table 3: Overview of BPMN-D flow connectors

- START and END for event nodes;
- XOR-SPLIT and XOR-JOIN for gateways.

We consequently define:

- the set A of *activity nodes* as
  $\{n \mid n \in N \text{ and } type_N(n) \in \{\text{ATOMIC-TASK, IN-TASK, EX-TASK, ANY-TASK}\}\}$;
- the set E of *event nodes* as $\{n \mid n \in N \text{ and } type_N(n) \in \{\text{START, END}\}\}$;
- the set G of *gateways* as $\{n \mid n \in N \text{ and } type_N(n) \in \{\text{XOR-SPLIT, XOR-JOIN}\}\}$;

Notice that $N = A \uplus E \uplus G$.

- $\ell_N : A \longrightarrow 2^\Sigma$ is a function that assigns task names to activity nodes in N. To guarantee that each activity node is mapped to a set of tasks consistently with its specific type, ℓ_N must satisfy the following conditions:
  - for every $a \in A$ such that $type_N(a) = \text{ATOMIC-TASK}$, $|\ell_N(a)| = 1$;
  - for every $a \in A$ such that $type_N(a) \in \{\text{IN-TASK, EX-TASK}\}$, $\emptyset \subset \ell_N(a) \subset \Sigma$;
  - for every $a \in A$ such that $type_N(a) = \text{ANY-TASK}$, $\ell_N(a) = \emptyset$.
- $F \subseteq N \times N$ is a set of flow connectors that obeys to the following restrictions: *(i)* every start event node has no incoming sequence flow, and a single outgoing sequence flow; *(ii)* every end event node has a single incoming sequence flow, and no outgoing sequence flow; *(iii)* every activity node has a single incoming and a single outgoing sequence flow; *(iv)* every XOR split gateway has a single incoming and at least two outgoing sequence flows; *(v)* every XOR join gateway has a single outgoing and at least two incoming sequence flows.[4]
- $type_F$ is a total function from F to the finite set of *flow connector types* $\{\text{SEQ-FLOW, IN-FLOW, EX-FLOW, ANY-FLOW}\}$ (cf. Table 3).
- $\ell_F : F \longrightarrow 2^\Sigma$ is a function that assigns task names to flow connectors in F. To guarantee that each flow connector is mapped to a set of tasks consistently with its specific type, ℓ_F must satisfy the following conditions:
  - for every $f \in F$ such that $type_F(f) \in \{\text{SEQ-FLOW, ANY-TASK}\}$, $\ell_F(f) = \emptyset$;
  - for every $f \in F$ such that $type_F(f) \in \{\text{IN-TASK, EX-TASK}\}$, $\emptyset \subset \ell_F(f) \subset \Sigma$.

---

[4] Graphically, we sometimes collapse a XOR join, connected to a XOR split via a standard BPMN sequence flow, into a single XOR gateway acting simultaneously as split/join.
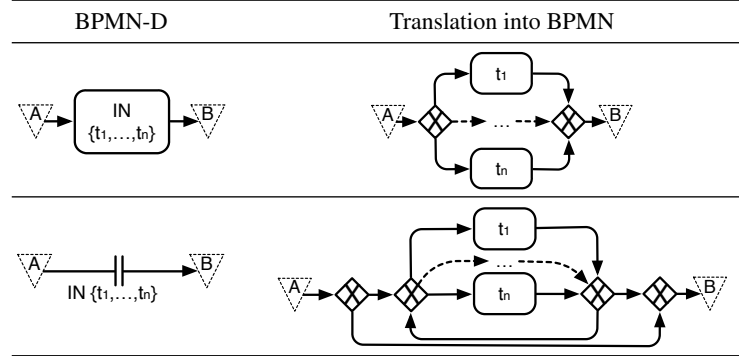
BPMN-D	Translation into BPMN

Table 4: Translation of the key BPMN-D elements into standard BPMN

3.3 Translating BPMN-D to Standard BPMN

Any BPMN-D diagram can be faithfully represented as a (trace-equivalent) corresponding standard BPMN diagram, at the price of conciseness. In this section, we discuss this translation, which has a twofold purpose: (i) it shows that, in principle, a BPMN-D process can be enacted on top of a standard BPMN engine; (ii) it provides an implicit execution semantics for BPMN-D in terms of standard BPMN.

For the translation, we assume that the overall set of tasks Σ is fixed. In this respect, it is sufficient to discuss how elements annotated with IN or ANY have to be translated: each label of the kind EX(T) can be in fact equivalently re-expressed as IN($\Sigma \setminus T$). As shown in Table 4 (top row), an inclusive task with label $\{t_1, \ldots, t_n\}$ is translated into a deferred choice where one of the tasks t_1, ..., t_n is selected. An ANY-labeled task is translated in the same way, considering all tasks in Σ as possible alternatives. The translation of an inclusive path sequence flow with label $\{t_1, \ldots, t_n\}$ is also depicted in Table 4 (bottom row). In this case, two alternative behaviors are obtained: either the inclusive path sequence flow behaves as a normal sequence flow (thereby directly connecting node A to node B), or it allows the executors to repeatedly execute tasks $t_1, \ldots, t_n$ in between. As for tasks, also the case of a ANY-labeled sequence flow is handled in the same way, just considering all tasks in Σ as possible alternatives. Figure 3 shows the result obtained by applying this translation procedure to the BPMN-D model shown in Figure 2.

4 From Declare to BPMN-D

We now propose a translation mechanism that, given a declarative, constraint-based process model, produces a corresponding readable BPMN-D diagram that faithfully represents the original intended behaviors. As source language, we consider Linear Temporal Logic on finite traces (LTL$_f$), which is the logic underpinning the Declare notation. However, it is worth noting that our approach can be directly applied to the more expressive logic LDL$_f$ [5], which has been recently adopted to formalize and monitor Declare constraints and meta-constraints [4].
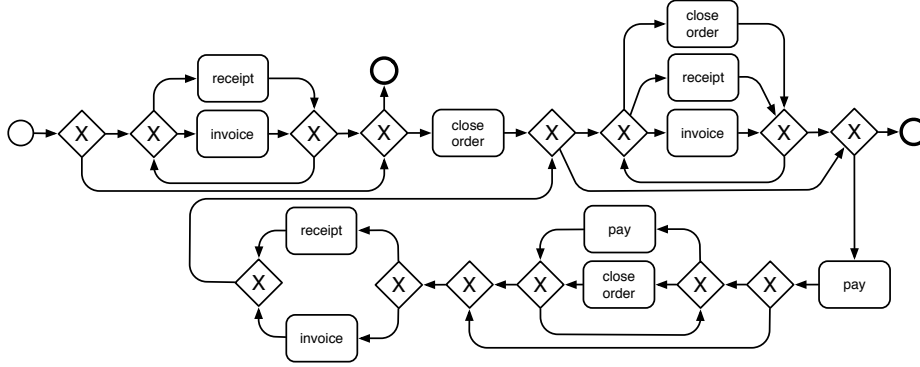
Fig. 3: Standard BPMN representation of the BPMN-D diagram of Figure 2

The algorithm proceeds in two phases. In the first phase, the given Declare (or LTL$_f$/LDL$_f$) specification is translated into a corresponding finite-state automaton that employs a form of declarative labels, and can therefore represent the given specification more compactly than for standard finite-state automata. In the second phase, this so-called "constraint automaton" is translated into a corresponding BPMN-D model. Both steps are such that the produced model accepts exactly the same traces as the input model, in turn guaranteeing that the BPMN-D model is a faithful, equivalent representation of the input Declare model.

In the remainder of this section, we provide a detailed account of these two phases.

4.1 From Declare to Constraint Automata

It is well-known that a Declare model can be transformed into a corresponding finite-state automaton that accepts exactly those traces that satisfy all the constraints present in the model [20, 4, 16]. A *finite-state automaton* (FSA) is a tuple $\langle \Sigma, S, I, F, \delta \rangle$, where: *(i)* Σ is the *alphabet* of symbols; *(ii)* S is a finite set of *states*; *(iii)* $I \subseteq S$ is the set of *initial states*; *(iv)* $F \subseteq S$ is the set of *final states*; *(v)* $\delta : S \times \Sigma \to 2^S$ is the *state transition function*, which maps each state and symbol to a set of successor states. Hereby, we assume that symbols represent atomic tasks, and consequently that the symbol alphabet is constituted by all atomic tasks that can be executed in the targeted domain. The *language* of an FSA A, written $\mathcal{L}(A)$, is the set of *finite traces* (i.e., words) over Σ that are accepted by A. The notions of deterministic finite-state automaton (DFA) and of minimal automaton are as usual.

Once the Declare model is translated into its corresponding FSA, the FSA can be determinized and minimized using standard techniques. This minimal DFA can be used to enact the Declare model [14]: at any time, the DFA states whether the process can be terminated or not, and indicates which tasks that can/must be executed next.

A drawback of this automata-based representation is that every transition of the automaton is labeled with a single, atomic task. This means that the same pair of states can be connected through several transitions, each associated to a different task. All

such transitions connect the same source state to the same destination state, and hence express different ways to achieve the same "effect" on the process. This closely resembles the notion of inclusive task in BPMN-D. To fully exploit this analogy, and make the automaton closer to BPMN-D, we introduce a variant of finite-state automata, called *constraint automata*. Differently from FSAs, constraint automata are finite-state automata whose transitions are associated to "declarative" labels, each of which acts as a *constraint* on the possible atomic tasks that can navigate the transition. In particular, a constraint automaton moves from one state to a successor state if the next symbol s to be analyzed *satisfies* the constraint C attached to the corresponding transition.

Given a task alphabet Σ, we consider the following *task constraints*, which are directly inspired from BPMN-D and consequently provide the basis for a natural translation of constraint automata to BPMN-D:

– t, with $t \in \Sigma$; an atomic task $t' \in \Sigma$ satisfies t iff $t = t'$.
– IN(T), with $\emptyset \subset T \subset \Sigma$; an atomic task $t \in \Sigma$ satisfies IN(T) iff $t \in T$.
– EX(T), with $\emptyset \subset T \subset \Sigma$; an atomic task $t \in \Sigma$ satisfies EX(T) iff $t \notin T$.
– ANY; every atomic task $t \in \Sigma$ satisfies ANY.

Intuitively, t represents the execution of an atomic task, IN(T) the execution of a task belonging to the set T of alternatives, EX(T) the execution of a task that does not fall inside the set T of forbidden tasks, and ANY the execution of some task. Like for BPMN-D, the following correspondences hold, consistently with the notion of satisfaction as defined above: *(i)* $t = $ IN($\{t\}$); *(ii)* IN(T) = EX($\Sigma \setminus T$).

In the following, we denote by $\mathcal{C}_\Sigma$ the set of all possible constraints that can be expressed over Σ. Technically, a *finite-state constraint automaton* (FCA) A_c is a tuple $\langle \Sigma, S_c, I_c, F_c, \delta_c, \ell_c \rangle$, where: *(i)* Σ is the (task) alphabet; *(ii)* S_c is a finite set of states; *(iii)* $I_c \subseteq S_c$ is the set of initial states; *(iv)* $F_c \subseteq S_c$ is the set of final states; *(v)* $\delta_c \subseteq S_c \times S_c$ is a transition relation between states; *(vi)* $\ell_c : \delta_c \to \mathcal{C}_\Sigma$ is a labeling function that, given a transition in δ_c, returns a task constraint over Σ.

Given a finite trace $\pi = \langle t_1, \ldots, t_n \rangle$ over Σ, we say that π is *accepted* by A_c if there exists a sequence of states $\langle s_1, \ldots, s_{n+1} \rangle$ over S_c such that: *(i)* $s_1 \in I_c$; *(ii)* $s_n \in F_c$; *(iii)* $\langle s_i, s_{i+1} \rangle \in \delta_c$ for $i \in \{1, n\}$; *(iv)* t_i satisfies constraint $\ell_c(\langle s_i, s_{i+1} \rangle)$ for $i \in \{1, n\}$. As usual, the *language* of a constraint automaton A_c, written $\mathcal{L}(A_c)$, is the set of *finite traces* over Σ that are accepted by A_c. We say that A_c is a *deterministic constraint automaton* (DCA) if $|I_c| = 1$ and, for every state $s \in S_c$ and every task $t \in \Sigma$, there exists *at most one* state $s' \in S_c$ such that $\langle s, s' \rangle \in \delta_c$ and t satisfies $\ell(\langle s, s' \rangle)$. If this is not the case, then A_c is a *nondeterministic constraint automaton* (NCA).

It is important to notice that in a constraint automaton, at most one transition can exist between a given pair of states. In fact, multiple simple transitions connecting the same pair of states can be compacted into a unique transition labeled with a constraint obtained from the combination of the original labels. We make this intuition systematic by introducing a translation mechanism that, given a standard finite-state automaton on Σ, produces a corresponding constraint automaton that employs the "most compact" constraints on arcs. In this context, "most compact" means that the constraint explicitly refers to the minimum number of tasks in Σ. For example, if $\Sigma = \{a, b, c\}$, we prefer EX($\{a\}$) over the equivalent constraint IN($\{b, c\}$).

Algorithm 1 Translation of a standard FSA to an equivalent FCA

1: **procedure** FSA2CA
2: **input** FSA $\langle \Sigma, S, I, F, \delta \rangle$
3: **output** FCA $\langle \Sigma, S_c, I_c, F_c, \delta_c, \ell_c \rangle$
4:     $S_c := S, I_c := I, F_c := F, \delta_c := \emptyset$
5:     **for all** $s_1, s_2 \in S$ **do**
6:         $T := \emptyset$
7:         **for all** $t \in \Sigma$ **do**
8:             **if** $s_2 \in \delta(s_1, t)$ **then** $T := T \cup \{t\}$
9:         **if** $T = \Sigma$ **then** $\delta_c := \delta_c \cup \langle s_1, s_2 \rangle, \ell_c(\langle s_1, s_2 \rangle) = \text{ANY}$
10:         **else if** $|T| \leq |\Sigma \setminus T|$ **then** $\delta_c := \delta_c \cup \langle s_1, s_2 \rangle, \ell_c(\langle s_1, s_2 \rangle) = \text{IN}(T)$
11:         **else if** $T \neq \emptyset$ **then** $\delta_c := \delta_c \cup \langle s_1, s_2 \rangle, \ell_c(\langle s_1, s_2 \rangle) = \text{EX}(\Sigma \setminus T)$
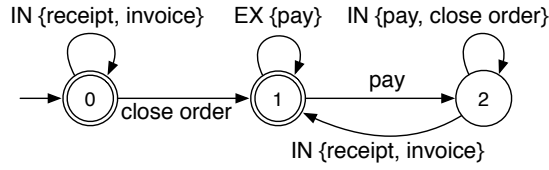

Fig. 4: Minimal DCA for the Declare model shown in Figure 1

The translation mechanism is defined in Algorithm 1, and it is straightforward to prove that it enjoys the following key properties:

Lemma 1. *For every* FSA *A: (i)* FSA2CA *is correct, i.e.,* $\mathcal{L}(A) = \mathcal{L}(\text{FSA2CA}(A))$; *(ii)* FSA2CA *preserves determinism, i.e., if A is a* DFA, *then* FSA2CA(A) *is a* DCA.

The mechanism of transition compaction shown in Algorithm 1 can be either applied on-the-fly, during the construction of the automaton starting from the Declare model, or as a final post-processing step (using FSA2CA itself). Both strategies do not affect the computational complexity of the automaton construction, and do not interfere with determinization (cf. Lemma 1). Furthermore, the correctness of FSA2CA guarantees that, after this first phase, the input Declare model is transformed into a constraint automaton that accepts exactly the same behaviors.

Figure 4 represents the minimal DCA that corresponds to the Declare model shown in Figure 1, assuming $\Sigma = \{close\ order, pay, receipt, invoice\}$ as task alphabet.

4.2 From Constraint Automata to BPMN-D

The algorithm for translating a constraint automaton A_c into a corresponding BPMN-D specification $\mathcal{M}$ is described in this section. We assume that the input constraint automaton is deterministic, and thus the unique initial state of A_c corresponds to a single start event node in $\mathcal{M}$. Consistently with the fact that automata are centered around states, while BPMN (and hence also BPMN-D) is centered around tasks, the translation maps states of A_c into flow connectors of $\mathcal{M}$, and transitions of A_c into activity nodes of $\mathcal{M}$.

The full translation mechanism is provided in Algorithm 2. It is immediate to see that the translation is linear in the size of the input automaton (measured by considering the number of its states and transitions). Each state s of A_c is handled according to the following rules: (1) State s is mapped to a flow connector f_s that connects a dedicated xor-join $in(s)$ to a dedicated xor-split $out(s)$ (cf. lines 7-9 of Algorithm 2); $in(s)$ accounts for the incoming transitions in s, while $out(s)$ accounts for the outgoing transitions from s. (2) If s has a self-loop t, then the type and label of f_s are set according to the constraint c attached to t (cf. lines 10-13); this accounts for the fact that as long as tasks satisfying c are executed, the process continues to stay in state s, which in turn means that it is still flowing through f_s. If instead s has no self-loop, f_s is a simple sequence flow connector (cf. line 14). (3) If s is an input state, then the start event node of $\mathcal{M}$ is connected with a sequence flow to $in(s)$ (cf. lines 15-16); this models that when the process starts, it immediately flows through f_s. (4) If s is an output state, then $out(s)$ is connected with a sequence flow to an end event node in $\mathcal{M}$ (cf. lines 17-20); this models the fact that, while the process is flowing through f_s, the process executors can decide to terminate it.

Each transition $\langle s_1, s_2 \rangle$ of A_c that is not a self-loop (i.e., such that $s_1 \neq s_2$), is then simply managed by: (1) introducing a corresponding activity node in $\mathcal{M}$, whose type and label is determined according to the constraint attached to the transition (cf. lines 24-27); (2) connecting $out(s_1)$ with a sequence flow to the activity node, and the activity node with another sequence flow to $in(s_2)$, reconstructing the state transition triggered by the constraint from which the activity node is derived (cf. lines 28-30).

Obviously, the technique so presented may lead to introduce several "inconsistent" x-or split and join gateways with only one input and one output attached sequence flow. To compensate for this issue, $\mathcal{M}$ is finally post-processed by removing all such unnecessary gateways (cf. the REMOVE-UNNECESSARY-XOR procedure on line 31 of Algorithm 2). This is quite straightforward, hence its actual code is omitted.

By considering the language of a DCA, and by modularly applying the translation procedure from BPMN-D to BPMN of Section 3.3 to the BPMN-D fragments produced by the different components of Algorithm 2, we have that:

Lemma 2. *The* FCA2BPMND *procedure is correct: for every* DCA, FCA2BPMND(DCA) *produces a proper BPMN-D model (according to the definition of Sec. 3.2), which accepts all and only the traces in* $\mathcal{L}(\text{DCA})$.

We close this section by illustrating, in Figure 5, the result of the FCA2BPMND procedure the FCA of Figure 4.

4.3   The Whole Translation Procedure

By combining the contributions of Sections 4.1 and 4.2, we can finally set up the whole translation procedure DECLARE2BPMND, which transforms a Declare model into BPMN-D, as shown in Algorithm 3. The following key result witnesses the correctness of this transformation:

Theorem 1. DECLARE2BPMND *is correct: for every Declare model* $\mathcal{D}$, *the BPMN-D model produced by* DECLARE2BPMND *accepts all and only the traces accepted by* $\mathcal{D}$.

Algorithm 2 Translation of an FCA to BPMN-D

1: **procedure** FCA2BPMND
2: **input** FCA $\langle \Sigma, S_c, I_c, F_c, \delta_c, \ell_c \rangle$
3: **output** BPMN-D model $\mathcal{M} = \langle N, type_N, \ell_N, F, type_F, \ell_F \rangle$
4:     **pick** fresh node se
5:     $F := \emptyset \; N := \{se\}, type_N(se) := \text{START}$
6:     **for all** $s \in S_c$ **do**
7:         **pick** fresh nodes $in(s)$ and $out(s)$
8:         $f_s := \langle in(s), out(s) \rangle, N := N \cup \{in(s), out(s)\}, F := F \cup \{f_s\}$
9:         $type_N(in(s)) := \text{XOR-JOIN}, type_N(out(s)) := \text{XOR-SPLIT}$
10:        **if** $\langle s, s \rangle \in \delta_c$ and $\ell_c(\langle s, s \rangle) = t$ **then** $type_F(f_s) := \text{IN-FLOW}, \ell_F(f_s) := \{t\}$
11:        **else if** $\langle s, s \rangle \in \delta_c$ and $\ell_c(\langle s, s \rangle) = \text{IN}(T)$ **then** $type_F(f_s) := \text{IN-FLOW}, \ell_F(f_s) := T$
12:        **else if** $\langle s, s \rangle \in \delta_c$ and $\ell_c(\langle s, s \rangle) = \text{EX}(T)$ **then** $type_F(f_s) := \text{EX-FLOW}, \ell_F(f_s) := T$
13:        **else if** $\langle s, s \rangle \in \delta_c$ and $\ell_c(\langle s, s \rangle) = \text{ANY}$ **then** $type_F(f_s) := \text{ANY-FLOW}, \ell_F(f_s) := \emptyset$
14:        **else** $type_F(f_s) := \text{SEQ-FLOW}, \ell_F(f_s) := \emptyset$ $\triangleright \langle s, s \rangle \notin \delta_c$
15:        **if** $s \in I_c$ **then**
16:           $F := F \cup \{\langle se, in(s) \rangle\}, type_F(\langle se, in(s) \rangle) := \text{SEQ-FLOW}, \ell_F(\langle se, in(s) \rangle) := \emptyset$
17:        **if** $s \in F_c$ **then**
18:           **pick** fresh node ee_s
19:           $N := N \cup \{ee_s\}, type_N(ee_s) := \text{END}, F := F \cup \{\langle out(s), ee_s \rangle\}$
20:           $type_F(\langle out(s), ee_s \rangle) := \text{SEQ-FLOW}, \ell_F(\langle out(s), ee_s \rangle) := \emptyset$
21:     **for all** $\langle s_1, s_2 \rangle \in \delta_c$ such that $s_1 \neq s_2$ **do**
22:         **pick** fresh node a
23:         $N := N \cup \{a\}$
24:         **if** $\ell_c(\langle s_1, s_2 \rangle) = t$ **then** $type_N(a) := \text{ATOMIC-TASK}, \ell_N(a) := \{t\}$
25:         **else if** $\ell_c(\langle s_1, s_2 \rangle) = \text{IN}(T)$ **then** $type_N(a) := \text{IN-TASK}, \ell_N(a) := T$
26:         **else if** $\ell_c(\langle s_1, s_2 \rangle) = \text{EX}(T)$ **then** $type_N(a) := \text{EX-TASK}, \ell_N(a) := T$
27:         **else** $type_N(a) := \text{ANY-TASK}, \ell_N(a) := \emptyset$ $\triangleright \ell_c(\langle s_1, s_2 \rangle) = \text{ANY}$
28:         $F := F \cup \{\langle out(s_1), a \rangle, \langle a, in(s_2) \rangle\}$
29:         $type_F(\langle out(s_1), a \rangle) := \text{SEQ-FLOW}, \ell_F(\langle out(s_1), a \rangle) := \emptyset$
30:         $type_F(\langle a, in(s_2) \rangle) := \text{SEQ-FLOW}, \ell_F(\langle a, in(s_2) \rangle) := \emptyset$
31:     $\text{REMOVE-UNNECESSARY-XOR}(\mathcal{M})$

Proof. First of all, by Lemma 1, since FSA2CA is applied on A after the determinization, also the produced constraint automaton is actually an FCA, and hence it can be correctly fed into FCA2BPMND. The correctness of line 5 is obtained from [4], and that of line 7 is obtained by applying Lemma 1 and 2.

5 Conclusion

We have provided elements to support a negative answer to the original question: "Are completely new notations needed to support hybrid process modeling?". The definition of BPMN-D as a conservative extension to BPMN shows that "open-world" modeling constructs can be embedded into existing imperative process modeling notations without fundamentally extending their semantics. Indeed, the proposed BPMN-D notation
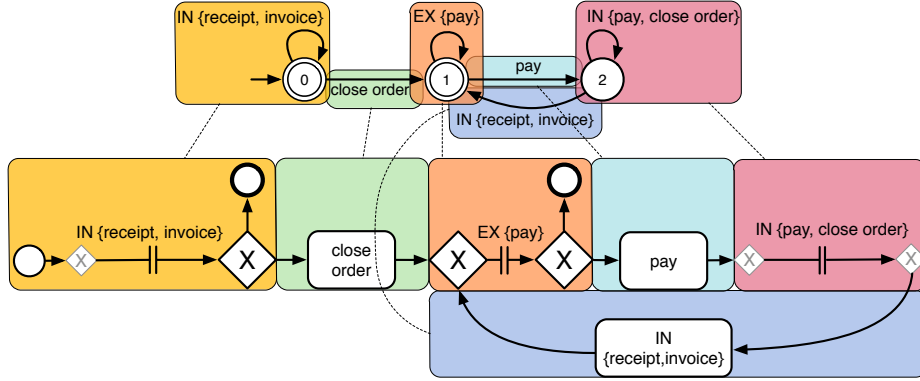
Fig. 5: Application of FCA2BPMND procedure to the FCA of Fig. 4; the small, light xor gateways are removed (cf. last line of the algorithm) leading to the diagram in Fig. 2

Algorithm 3 Translation of a Declare model to BPMN-D

1: **procedure** DECLARE2BPMND
2: **input** Declare model $\mathcal{D}$
3: **output** BPMN-D model $\mathcal{M}$
4: $\quad \Phi := \bigwedge_{c \text{ constraint of } \mathcal{D}} \text{LTL}_f(c)$ $\qquad$ ▷ Obtained from the LTL_f formalization of Declare
5: $\quad A := \text{LDL}_f 2\text{NFA}(\Phi)$ $\qquad$ ▷ A is an NFA produced using the technique in [4]
6: $\quad A := \text{MINIMIZE}(\text{DETERMINIZE}(A))$ $\qquad$ ▷ Standard automata operations
7: $\quad \mathcal{M} := \text{FCA2BPMND}(\text{FSA2CA}(A))$

---

is a macro-extension of BPMN. Moreover, we have shown that this notation can capture the range of constraints present in the Declare notation in an intuitive manner.

In its present form, the translation from Declare to BPMN-D generates process models with exclusive (XOR) gateways only, thus without parallelism. A direction for future work is to extend this translation with the ability to generate BPMN-D models with inclusive and parallel gateways. A possible approach is to adapt existing techniques from theory of regions [3], which extract parallelism in the context of Petri nets. A direct application of this approach can lead to unreadable process models as put into evidence in [16]. However, if we take constraint automata as a basis – as in our translation approach – it may be possible to adapt techniques from theory of regions to produce simpler constraint-annotated Petri nets that explicitly capture parallelism, and from there we could generate a BPMN-D process model.

References

1. van der Aalst, W., Pesic, M., Schonenberg, H.: Declarative workflows: Balancing between flexibility and support. Computer Science - Research and Development 23 (2009)

2. Awad, A., Sakr, S.: On efficient processing of BPMN-Q queries. Computers in Industry 63(9) (2012)
3. Carmona, J.: Projection approaches to process mining using region-based techniques. Data Min. Knowl. Discov. 24(1) (2012)
4. De Giacomo, G., De Masellis, R., Grasso, M., Maggi, F., Montali, M.: Monitoring business metaconstraints based on LTL & LDL for finite traces. In: 12th Int. Conf. on Business Process Management (BPM). LNCS, vol. 8659. Springer (2014)
5. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: 23rd Int. Joint Conf. on Artificial Intelligence (IJCAI). AAAI (2013)
6. De Smedt, J., De Weerdt, J., Vanthienen, J.: Multi-paradigm process mining: Retrieving better models by combining rules and sequences. In: Proc. of OTM Conferences. LNCS, vol. 8841. Springer (2014)
7. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Information & Software Technology 50(12), 1281–1294 (2008)
8. Hildebrandt, T.T., Mukkamala, R.R., Slaats, T.: Nested dynamic condition response graphs. In: 4th IPM Int. Conf. on Fundamentals of Software Engineering (FSEN). vol. 7141 (2011)
9. Hull, R., Damaggio, E., Masellis, R.D., Fournier, F., Gupta, M., Heath, F., Hobson, S., Linehan, M., Maradugu, S., Nigam, A., Noi Sukaviriya, P., Vaculín, R.: Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. In: 5th ACM Int. Conf. on Distributed Event-Based Systems (DEBS). ACM (2011)
10. Maggi, F.M., Slaats, T., Reijers, H.A.: The automated discovery of hybrid processes. In: 12th Int. Conf. on Business Process Management (BPM). LNCS, vol. 8659. Springer (2014)
11. Marin, M., Hull, R., Vaculín, R.: Data centric BPM and the emerging case management standard: A short survey. In: Proc. of BPM Workshops. Springer (2012)
12. Montali, M., Pesic, M., van der Aalst, W.M.P., Chesani, F., Mello, P., Storari, S.: Declarative Specification and Verification of Service Choreographies. ACM Tr. on the Web (2010)
13. Object Management Group: Business Process Modeling Notation Version 2.0. Tech. rep., Object Management Group Final Adopted Specification (2011)
14. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: Declare: Full support for loosely-structured processes. In: EDOC07 (2007)
15. Pichler, P., Weber, B., Zugal, S., Pinggera, J., Mendling, J., Reijers, H.: Imperative versus declarative process modeling languages: An empirical investigation. In: BPM Ws (2011)
16. Prescher, J., Di Ciccio, C., Mendling, J.: From declarative processes to imperative models. In: 4th Int. Symp. on Data-Driven Process Discovery and Analysis (SIMPDA). CEUR-WS.org (2014)
17. Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies. Springer (2012)
18. Reijers, H.A., Slaats, T., Stahl, C.: Declarative modeling-an academic dream or the future for BPM? In: 11th Int. Conf. on Business Process Management (BPM). LNCS, vol. 8094. Springer (2013)
19. Schonenberg, H., Mans, R., Russell, N., N.Mulyar, van der Aalst, W.: Towards a Taxonomy of Process Flexibility. In: Forum at the CAiSE'08 Conf. vol. 344. CEUR-WS.org (2008)
20. Westergaard, M.: Better algorithms for analyzing and enacting declarative workflow languages using LTL. In: 11th Int. Conf. on Business Process Management (BPM). LNCS, vol. 8094. Springer (2011)
21. Westergaard, M., Slaats, T.: Mixing paradigms for more comprehensible models. In: 11th Int. Conf. on Business Process Management (BPM). vol. 8094. Springer (2013)