Bus Inf Syst Eng 58(1):19–29 (2016)
DOI 10.1007/s12599-015-0416-y

RESEARCH PAPER

Mixed-Paradigm Process Modeling with Intertwined State Spaces

Johannes De Smedt · Jochen De Weerdt · Jan Vanthienen · Geert Poels

Received: 28 February 2015 / Accepted: 7 September 2015 / Published online: 2 December 2015

Abstract Business process modeling often deals with the trade-off between comprehensibility and flexibility. Many languages have been proposed to support different paradigms to tackle these characteristics. Well-known procedural, token-based languages such as Petri nets, BPMN, EPC, etc. have been used and extended to incorporate more flexible use cases, however the declarative workflow paradigm, most notably represented by the Declare framework, is still widely accepted for modeling flexible processes. A real trade-off exists between the readable, rather inflexible procedural models, and the highly-expressive but cognitively demanding declarative models containing a lot of implicit behavior. This paper investigates in detail the scenarios in which combining both approaches is useful, it provides a scoring table for Declare constructs to capture their intricacies and similarities compared to procedural ones, and offers a step-wise approach to construct mixed-paradigm models. Such models are especially useful in the case of environments with different layers of flexibility and go beyond using atomic subprocesses modeled according to either paradigm. The paper combines Petri nets and Declare to express the findings.

Keywords Business process modeling · Mixed-paradigm process modeling · Petri nets · Declare

Accepted after two revisions by the editors of the special issue.

Electronic supplementary material The online version of this article (doi:10.1007/s12599-015-0416-y) contains supplementary material, which is available to authorized users.

J. De Smedt (✉) · Prof. J. De Weerdt · Prof. J. Vanthienen
KU Leuven Faculty of Economics and Business, Leuven
Institute for Research on Information Systems, Naamsestraat 69,
3000 Louvain, Belgium
e-mail: Johannes.DeSmedt@kuleuven.be

Prof. J. De Weerdt
e-mail: Jochen.DeWeerdt@kuleuven.be

Prof. J. Vanthienen
e-mail: Jan.Vanthienen@kuleuven.be

Prof. G. Poels
UGent Management Information Systems Research Group,
Ghent University Faculty of Economics and Business
Administration, Tweekerkenstraat 2, 9000 Ghent, Belgium
e-mail: Geert.Poels@ugent.be

1 Introduction

Business Process Modeling (BPM) (Dumas et al. 2013) has become a powerful approach for managers to capture and analyze their workflows. To be effective, process models need to be both expressive and understandable. To achieve these goals, numerous languages have been proposed, each adding a certain aspect to the BPM language and tool sphere. There are two main control-flow paradigms, extensively discussed in Goedertier et al. (2013), that deal with the trade-off between comprehensibility and flexibility, the procedural and declarative paradigms. The former is characterized by the use of explicit activity flows to express the activity paths through a process model, while the latter is typified by a focus on curtailing behavior with activity-level rules rather than specifying entire activity paths, thus leaving many options for possible enactment. On the one hand, procedural models are regarded as rigid, but comprehensible, as they present the reader with what is possible in the process in a rather deterministic way. Declarative models on the other hand, leave much unspecified and therefore are harder to read, as the activity sequences allowed by the model remain implicit until they

become visible during execution. Each paradigm also has solutions to leverage its issues with flexibility and comprehensibility. For example, in procedural process models one can loosen the typically explicit paths around a certain activity, or declarative models one can overly restrain a process path to obtain a stricter workflow.

This paper investigates the possibilities of combining constructs of both paradigms in an intertwined model, supported by the semantics of both languages. More specifically, the authors seek to combine Petri nets with Declare, which are both well-supported languages in their paradigm. Mixed forms have already been discussed in Pesic et al. (2007) and Westergaard and Slaats (2013), mainly focusing on execution, while this paper rather focuses on the modeling effort itself. Since many real-life processes are not completely flexible, nor completely fixed, the setup of mixing both paradigms offers business process modelers many applications. The contributions are as follows. We identify in which scenarios such models are useful and what benefits they offer in a tutorial-like style. Also, we scrutinize the overlap and interplay of mixed-models' semantics and syntax with a scoring table for Declare constructs. Constraints that obtain a higher score are more difficult to represent with Petri net-based constructs and thus are a greater need of a mixed model. Finally, we propose a step-wise approach for modeling mixed-paradigm models for future users, taking into account the different characteristics of both models. Accordingly, this paper tries to address the following issues raised in van der Aalst (2013):

- The paper proposes a step-wise approach to model mixed-paradigm models, addressing use case *Design Model* (DesM).
- In case models from different systems, expressed by different model types are merged, the paper can be helpful to support the *Merge Models* (MerM) and *Compose Model* (CompM) use cases.
- The paper addresses issues that arise when mixing different model types, addressing the use case *Enact Model* (EnM).

The remainder of this paper is structured as follows. Section 2 reviews the state-of-the-art and represents the different approaches graphically. Next, the models' syntax and semantics are discussed. An example and use case for combining both paradigms is given in Sect. 4, after which Sect. 5 compares model constructs and different characteristics that are of high importance when mixing different paradigms. Finally, Sect. 6 provides a step-wise mixed-modeling approach and is followed by the conclusion which outlines future work.

2 Related Work

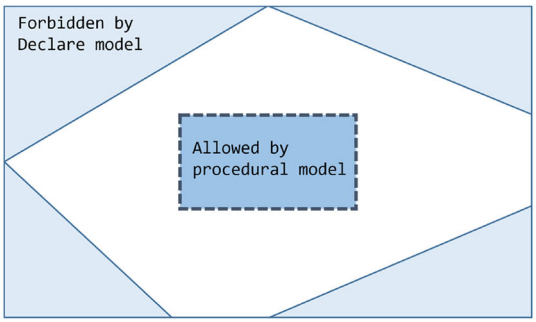
2.1 Procedural BPM

Process modeling has gained ground as a methodology to represent activities in a directed graph-like manner in order to capture and discuss business flows such as an ordering process, a customer journey, etc. (Rosemann et al. 2006). For this purpose, many languages have been proposed, most notably business process model and notation (BPMN) (White 2004), Petri nets (Murata 1989), event-driven process chains (EPC) (van der Aalst 1999), and yet another workflow language (YAWL) (van der Aalst and Ter Hofstede 2005). BPMN and EPC are often used in a business context and have been enhanced with numerous constructs supporting, e.g., message flows and ad-hoc processes. YAWL can be seen as an extension and effort to improve the stripped down Petri net execution semantics. Due to the simple, yet effective way Petri nets can capture flows and concurrency, they are widely used in numerous application domains. Their properties are well-studied and as such they remain very popular with researchers as well. Furthermore, the analysis techniques such as state space generation and soundness checks (van der Aalst 2002) make them the preferred language to which BPMN models and EPCs are translated to in order to provide firm execution semantics and model checking (Dijkman et al. 2008; van der Aalst 1999).

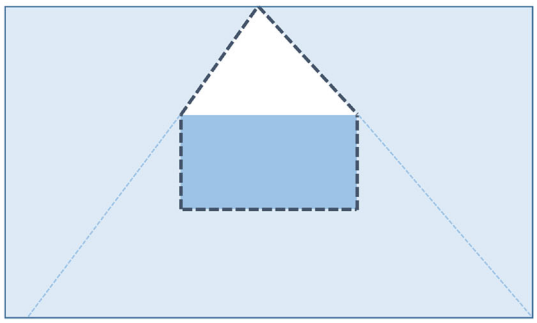
2.2 Declarative and Flexible BPM

Flexibility has numerous forms, such as flexibility by design, deviation, underspecification, and change, which are described in Schonenberg et al. (2008). Flexible process models tend to make use of these concepts, mostly in certain parts of the model, e.g., pockets of flexibility (Sadiq et al. 2001) and worklets (Adams et al. 2006). These are approaches for enabling procedural models to include flexible behavior by postponing and underspecifying execution decisions until run-time.

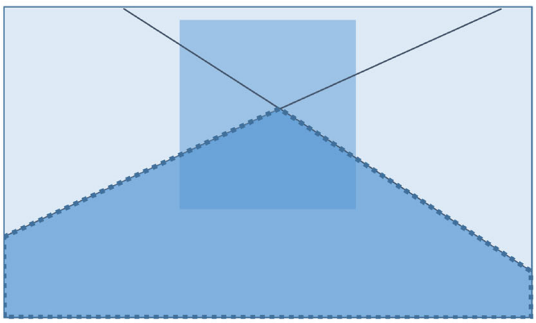
The major difference between procedural and declarative modeling is the way in which one approaches the model: either a specification of what has to happen (procedurally) is made, leaving no room for non-modeled behavior, compared to specifying what can happen, where everything that is not prohibited is possible (declaratively). Hence, declarative models leave more room for non-modeled behavior and thus are regarded as allowing more flexibility in the process execution. The event- and rule-driven Declare framework (Pesic and van der Aalst 2006; Pesic et al. 2007) has gained attraction amongst researchers as a completely flexible solution. Declare is based on rule



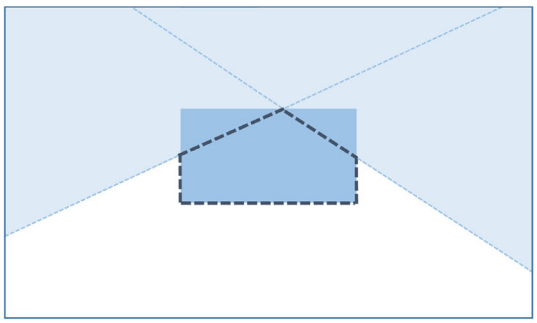
(a) The behavior allowed by the procedural model is depicted as the dark square, the behavior allowed by the declarative model as a trapezoid.



(b) This figure shows a procedural model which is relaxed on one side where the behavior is restricted only by the declarative model.



(c) The model is a pentagon using both model paradigms to account for the different levels of flexibility.



(d) This figures shows a procedural model which is even further restricted by declarative constraints.

Fig. 1 *Three layers* indicating all the possible behavior of the activities and flow constructs contained in a model. The *dotted line* represents the outcome of a combination of declarative and procedural constructs in (**b**), (**c**), and (**d**). A color version of all figure is available online via http://link.springer.com. **a** The behavior allowed by the procedural model is depicted as the *dark square*, the behavior allowed by the declarative model as a *trapezoid*. **b** This figure shows a procedural model which is relaxed on one side where the behavior is restricted only by the declarative model. **c** The model is a *pentagon* using both model paradigms to account for the different levels of flexibility. **d** This figures shows a procedural model which is even further restricted by declarative constraints

templates, which are developed by the means of linear temporal logic (LTL). Declare has become a widely spread language and modeling suite for rule-based modeling. Other well-known languages include the guard-stage-milestone models (Hull et al. 2011) and dynamic condition response graphs (Hildebrandt et al. 2012).

The differences and characteristics of how modelers and users apply both paradigms, has been researched extensively by Reijers et al. (2013), Haisjackl et al. (2014), and Fahland et al. (2009). The outcomes suggest that, overall, it is very difficult to read Declare models due to the invisible execution of accepting and non-accepting behavior, the lack of clear sequences in the beginning and ending, the subtleties of the constraints, and especially the complex interaction of the different templates. The overall suggestion is to model very sequential information with procedural languages, and to model flexible processes with declarative languages instead, in analogy to procedural and declarative programming.

The different types of behavior of the two paradigms can be depicted as in Fig. 1. To the left, Fig. 1a shows the traditional representation of both paradigms as in Pesic et al. (2007). Usually, Declare is referred to as the model type that constraints behavior by activity-level rules, leaving options open for more flexible specification and execution. Procedural models are depicted as very rigid process flows, containing only strictly regulated and delineated behavior.

2.3 Mixed Forms and Conversion

Modeling languages incorporating both paradigms also exist, but still focus rather on separate subworkflows, modeled with either procedural or declarative constructs, in

order to keep the state spaces and execution semantics of these subworkflows separated. This has been proposed for, e.g., YAWL and Declare (van der Aalst et al. 2009). This approach is similar to pockets of flexibility. Thus, flexibility is introduced into some parts of the process in a hierarchical way. This is depicted in Fig. 1b. Typically, a certain part of the model is loosened, e.g., the procedural model loosens a certain part which is now constrained by activity-level rules. The opposite is also possible, in that a very flexible main process contains some fixed sequences which can be easily captured by, e.g., a small Petri net fragment.

Execution semantics for truly intertwined state spaces exist as well. In Westergaard and Slaats (2013), execution semantics for Petri nets and Declare automata are presented. Intertwined state spaces can also be constructed by mixing converted Declare constraints expressed in Petri net constructs with other Petri nets, thus obtaining a mixed-model. In Fahland (2007), the possibility of converting a subset of DecSerFlow constraints, the predecessor of Declare, has been investigated. A full conversion is sought after in De Smedt et al. (2015), in which the full body of Declare templates is offered as a lexicon of Petri net constructs, extended with reset and inhibitor arcs (R/I-nets). The conversion of Declare constraints based on regular expressions has been researched in Prescher et al. (2014). By making use of synthesizing finite state machines into Petri nets with the theory of regions (Cortadella et al. 1998), Declare constraints can be converted to Petri nets. This technique is similar to enumerating all possible execution scenarios, as many duplicate activities are required to do so.

A process with mixed layers of flexibility which spread throughout the whole state space of the model cannot be captured by using solely subworkflows, as this setup requires the models to synchronize to a state before and after executing the subflow. For instance, an activity which can appear to be rather flexible, i.e., without a fixed place in a sequence, but which still affects a procedural part of the model cannot be modeled outside of its subworkflow. In a true mixed-paradigm approach with intertwined state spaces, process behavior is restricted by making use of the most appropriate combination of subsets of both models, thus combining modeling constructs that restrict the process behavior in some directions, but relax behavioral constraints in other directions. This is depicted in Fig. 1c, where a subset of both models constitutes the mixed-paradigm model. Still, one paradigm can dominate the other (e.g., the example in Fig. 6 where the declarative part clearly dominates the procedural part), however, they can also have equal influence on activities. Furthermore, not only flexibility can be achieved, but also an especially strict specification. In Fig. 1d, the Declare constraints cut into the procedural model, resulting in a less flexible model as sequence rules impose even further restrictions on the workflow.

3 Model Syntax and Semantics

The syntax of the mixed models used in this paper is based on the syntaxes of Petri nets and Declare. All activities are represented as transitions, connected by both Petri net places and arcs, and Declare arcs. The semantics for executing them are discussed below. For Petri nets, they are intertwined with syntax, for Declare they are not. Execution semantics, however, can aid users in understanding construct implications as they can immediately recreate a token game.

3.1 Declare Execution Semantics

In order to execute a Declare model, i.e., a set of declarative constraints, the constraints are converted to Büchi automata (Pesic 2008). Next, by taking the product of all separate automata (one for each constraint), a full

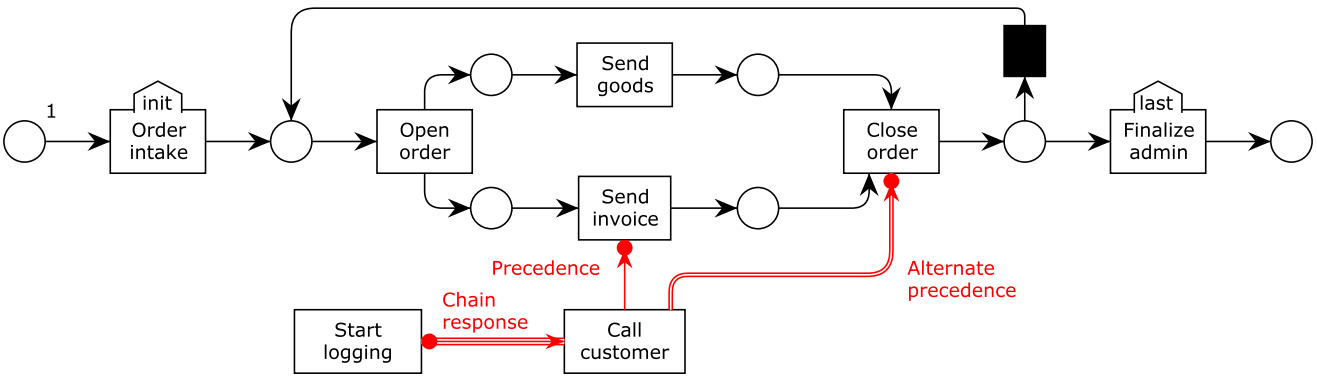


Fig. 2 A very straightforward AND-split and -join based process model represented in a mixture of Petri nets and Declare in standard notation

executable model is obtained, which can then be applied to detect satisfying, temporal, and permanently violated states when replaying words over them (Maggi et al. 2012).

In more recent work, a shift is made towards expressing Declare constraints by means of regular expressions (as opposed to LTL formula) (Di Ciccio and Mecella 2013; Westergaard et al. 2013). Both works deem LTL unfit to express finite traces and hence redefine Declare in finite state machines. A full overview of all constraints can be found in Table 2 (available online via http://link.springer.com).

3.2 Petri Nets with Reset and Inhibitor Arcs

Petri nets (Murata 1989) are a mathematical modeling language to describe distributed, concurrent systems. A weighted Petri net with reset and inhibitor arcs is a directed graph, expressed as a tuple, $PN = (P, T, F, R, I, W)$, with P a finite set of places (visually represented as circles), T a finite set of transitions (visually represented as boxes) with $P \cap T = \emptyset$, and $F \subseteq (P \times T) \cup (T \times P)$ the set of normal arcs (shown as arcs with a single arrow). Let $W : F \rightarrow \mathbb{N}$ determine a weighting function which associates a weight to each arc. Let $R : T \rightarrow \mathbb{P}(P)$ define the reset places (with $\mathbb{P}(P)$ the powerset of P) and $I : T \rightarrow \mathbb{P}(P)$ the inhibitor places for each transition, which also implicitly define the reset arcs (shown as an arc ending with double arrows) and inhibitor arcs (shown as an arc ending with a circle) respectively. The set of input nodes of a node $x \in P \cup T$ is denoted as $\bullet x = \{(y \in P \cup T | (y,x) \in F) \vee \quad (x \in T \wedge y \in R(x) \cup I(x))\}$, and the output nodes similarly as $x\bullet$.

The state of a Petri net is called marking $M \in P \rightarrow \mathbb{N}$, indicating the number of tokens contained in each place. A transition t is said to be enabled, denoted as $M[t\rangle$, iff $M(p) > 0, \forall p \in \bullet t : [(p,t) \in F \quad \vee p \in R(t)] \wedge M(p) = 0, \forall p \in I(t)$. Firing an enabled transition results in a new marking M' so that $M'(p) = M(p) - (M(p)\text{iffp} \in \text{R(t)}, W(p,t)\ \text{iff(p,t)} \in \text{F}, 0\ \text{otherwise}) + (W(t,p)\text{iff(t,p)} \in \text{F}, 0\ \text{otherwise})$. That is, tokens are removed from input places according to arc weights. Places which act as reset places for a fired transition are emptied completely. Next, the token count of output places is incremented according to arc weights to obtain the new marking. We refer to Murata (1989) for more details.

3.3 Mixed-paradigm with intertwined state spaces

For combining both Declare and Petri nets, we use the conversion approach of De Smedt et al. (2015). This has the benefit of a pluggable approach in which there is no need for merging both state spaces as in Westergaard and Slaats (2013), as both models use the same language. For the R/I-net constructs in Table 2, we define for every letter in the Declare template/model alphabet one in the Petri net alphabet $\Sigma_{PN} = \Sigma_{Dec} \cup \{\lambda_{Invisible}\}$, with labeling function $\delta : T \rightarrow \Sigma_{PN}$.

In the templates, it is assumed that $T = \{t_{sink}, t_A(, t_B, t_C), t_{sink}\}, t_C = T \backslash \{t_A, t_B, t_{source}, t_{sink}\}$, $P = \{p_{source}, p_{sink}, p_1(, p_2, p_3, p_4)\}$, and $F = \{(p_{source}, t_{source}), (t_{sink}, p_{sink})\}$.

In order to synchronize Declare and Petri net models, it is also necessary to initialize and end the execution properly. For this purpose, it is important that there are dedicated source and sink activities t_{source}, t_{sink}, in the Petri net that match the activities involved in the *Init* and *Last* constraints. Tokens needed in the initial state of the Declare constraints are inserted by t_{source} (e.g. *Response(A,B)* is temporarily violated by default, enforced by a token in the input place of t_{sink}, and connected with a reset arc with B, see Table 2). For the sake of brevity, we assume the tokens are present in the places where they are required in the initial marking. Hence $M_0(p) = W(t_{source}, p)\ \forall p \in \bullet t, \forall t \in T$. t_{sink} is added in a way in which it can fire only once to keep track of the violation state of a Declare constraint. This might require introducing an extra (invisible) sink transition.

4 Running Example of a Mixed Model with Intertwined State Spaces

Consider the mixed-paradigm model in Fig. 2 which contains a procedural backbone which is supplemented with a flexible component containing activities *Call customer* and *Start logging*. The flexible part starting with activity *Start logging* can execute irrespective of the behavior modeled in the procedural backbone, but still influences the main process. The inclusion of *Chain response(Start logging, Call customer)* (after *Start logging*, *Call customer* has to happen next) disrupts the global model, as every activity but *Call customer* becomes disabled after firing *Start logging*. *Call customer* has to happen before *Send invoice* can ever occur (*Precedence*), and *Close order* can only fire again after a new occurrence of *Call customer* (*Alternate precedence*).

The combined use of procedural and declarative constructs results in an effective alternative solution, in-between solutions that would use declarative or procedural model constructs exclusively. By explicitly capturing the loop with Petri nets and reducing the amount of Declare constraints, readers and modelers can easily grasp the token game while a few verbose sequence rules (which can be found in Table 2) can explain the interplay of the flexible activities with the procedural net.

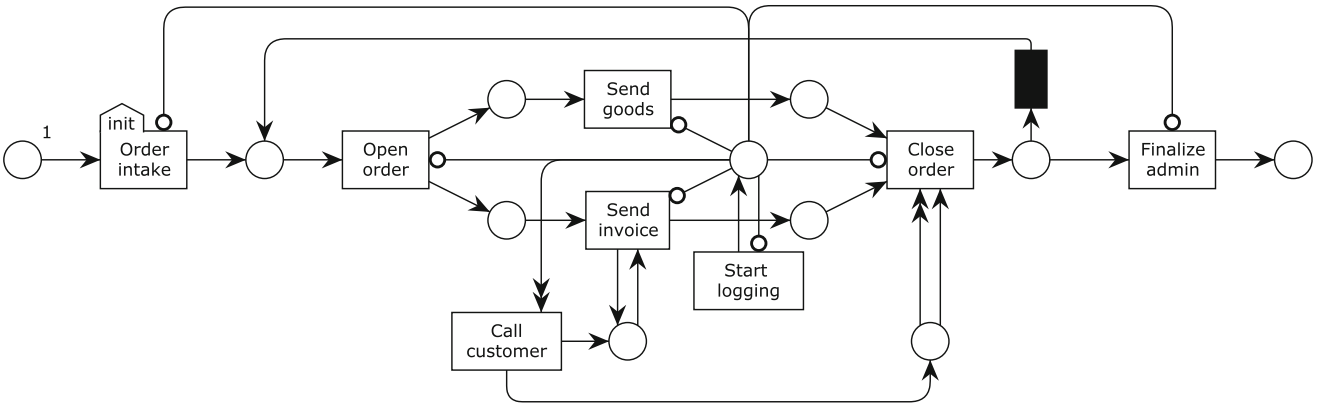
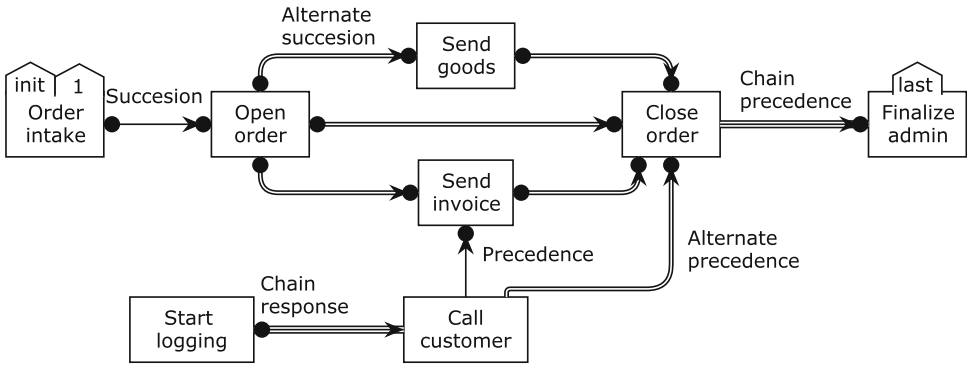


Fig. 3 The same model as in Fig. 2, but now solely in R/I-net constructs

Fig. 4 The same model as in Fig. 2, but now solely in Declare standard notation



Modeling the same scenario with a procedural language such as Petri nets, results in either a model with many duplicate activities, or reset and inhibitor constructs, as depicted in Fig. 3. Furthermore, capturing *Chain response* severely disrupts the main process which needs the incorporation of many (inhibitor) arcs which clutter up the model completely.

Using Declare, it is hard to capture the procedural backbone in a straightforward and comprehensible way. To capture the same behavior as the loop does, one needs many *Alternate succession* constraints in which the loop remains hidden. Also, a *Chain precedence* constraint is required to model the XOR-split at the end of the loop. By providing readers solely with the standard constraint description, interpreting the model requires a significant amount of cognitive effort (Fig. 4).

In the end, a Declare model is not executable unless transformed into an automaton, displayed in Fig. 5. The flexible activities of Fig. 2 are indicated in red as well.

The state space is the same for all the models, and in the automaton, it is clearly visible how the state spaces are intertwined. The procedural behavior only needs a few state transitions, while the flexible behavior requires the inclusion of many of them, even though only three Declare constraints are used in the case of the mixed model in Fig. 2.

Observe that using a subworkflow for *Call customer* and *Start logging* is not possible. Since both activities affect the main workflow, one cannot simply model these activities in a concurrent subworkflow as, e.g., the impact of the *Chain response* is global and not restricted to both activities involved. Therefore, mixed-paradigm modeling attempts that only allow a combination of paradigms by making use of fully separated subprocesses modeled with one or another type of constructs, are not able to model the desired behavior appropriately.

5 Mixed-Paradigm Process Modeling: Constructs and Characteristics

Incorporating both modeling syntaxes and semantics into a single model requires carefully scrutinizing the different constructs and avoiding overlap as much as possible. In this section, a scoring mechanism for Declare constraints is presented according to different characteristics, which makes it possible to assess how straightforward it is to express them in R/I-net constructs, whether the constraint

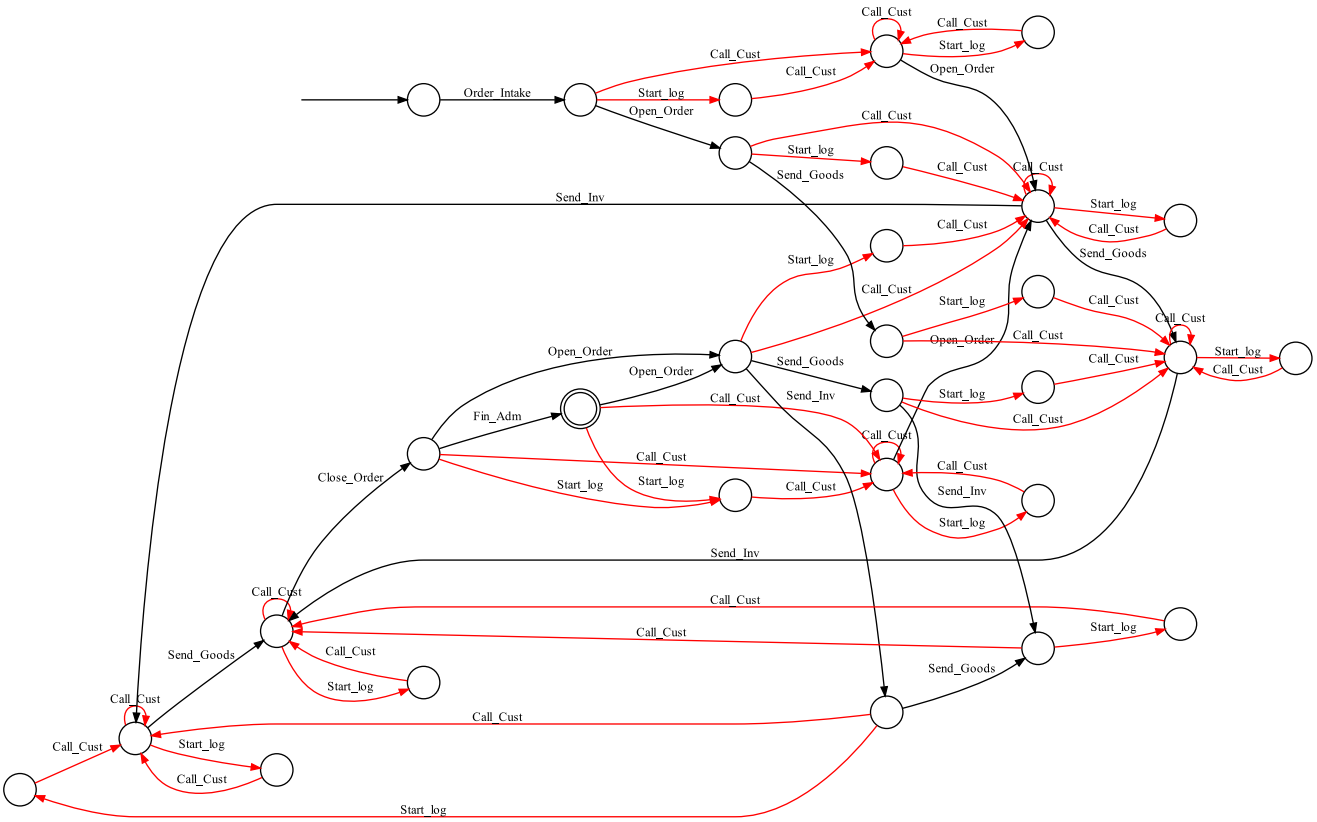


Fig. 5 The automaton for the Declare model with the flexible activity transitions in *red*

impacts global concurrency and global timing, and whether it inflicts hidden dependencies. These characteristics play an important role for merging procedural and declarative process models.

5.1 Construct-Based Similarities and Differences

Declare consists of many templates which have distinct features that require a large amount of Petri net constructs to mirror their behavior, as can be seen in Table 1. However, many other templates exist that can be straightforwardly represented with only a few Petri net constructs.m Therefore, these constraints can be easily interchanged in mixed-paradigm models to avoid using different syntaxes. The advantage of R/I-net constructs is that the syntax immediately yields execution semantics. Each constraint is thus scored for the amount of places (P) and occasionally transitions (T), arcs (A), reset arcs (R), and inhibitor arcs (I) that is needed to express them. Each construct is scored for 1 point.

5.2 Impact on a Global Concurrency Level

Constraints that can force activities, not directly related to them by other constraints, to be disabled impact global concurrency. Most notably, the *Chain* constraints exhibit this behavior, as they can stop any activity from executing until a certain other has fired. Not only does this require many constructs such as inhibitor arcs or prioritized Petri nets to model this in a procedural model, they also impact the execution semantics of, e.g., a Petri net mixed with a Declare model containing *Chain* constraint(s). This makes it harder to model and understand the behavior of such mixed-models. This is scored with 2 points in Table 1.

5.3 Impact on a Global Temporal Level

The concept of temporary violation is typical for rule-based approaches. It can be compared to a final marking in a Petri net. In the R/I-net, a dedicated sink transition t_{sink} is used to indicate the current violation status of the model (firing it leads to the accepting marking of a single token in p_{sink}). If the transition is enabled, no temporary violations are present (permanent violations cannot appear by default). Adding this explicit monitor helps users grasp the status of the net. Many constraints make use of this construct, as can be seen in Table 2. However, the concept of violation adds extra constructs and also requires a procedural model mixed with a Declare model to be able to also resolve the same temporary violation(s), which raises the efforts

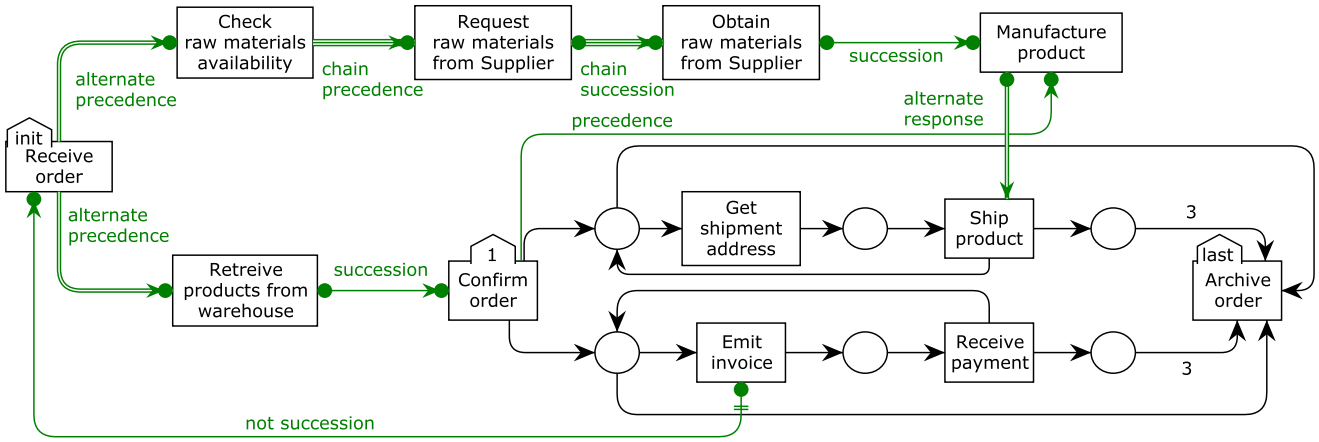


Fig. 6 A well-known fulfillment process model reworked according to the step-wise approach

needed to model correctly. Since this has a major impact, especially for synchronizing Declare with any other models in terms of temporal consistency, this is scored with 5 points in Table 1.

5.4 Permanently Disabling

Some Declare constraints require activities to become permanently disabled when they become satisfied. Most notably, *Absence, Exactly, Not succession, Not co-existence*, and *Exclusive choice* disable at least one activity for the rest of the execution. If this activity was still required to resolve any other constraints to an accepting state, the model ends up in a deadlock. This is often referred to in the literature as the 'hidden dependencies' (Haisjackl et al. 2014). In execution semantics, these dependencies are added by multiplying separate automata into one general executable automaton, as the sum of separate constraints does not prevent the model of ending up in such a state. This can result in, e.g., one *Exclusive choice* disabling many transitions permanently at once. Executing such models, thus, is extra precarious. Hence, constraints inflicting such behavior are hard to incorporate in two semantics at the same time, thus a high score of 10 is given to such constraints.

5.5 Overview

Taking into account all these different aspects of the constraints, a final score is assigned. The lower the score, the better. Constraint with a score below ten are easily pluggable into a procedural model. Between ten and twenty, considerable care must be taken. For constraint with a score above 20, it becomes very tedious to include them in a procedural model. E.g., the *Chain response* constraint requires one place, one Petri net arc, one reset arc, and inhibitor arcs connected to all other transitions in the net but one ($|T|-1$). Hence, it impacts global concurrency, as it can stop all activities in the net but one, and impacts global timing as it can be in a temporarily violated state. Hence, it receives a score of $(1+1+1+|T|-1)+2+5$. Since $|T|$ is included, using this constraint in bigger models with more transitions becomes more tedious.

As can be seen from the last column in Table 1, only a few constraints are considerably straightforward to model, comprehend, and use in a mixed-paradigm model:

- The simple and alternating ordered constraints are not impeded by the fact that they do not expose sophisticated behavior nor many constructs. This is especially true for *Precedence* constraints.
- Every constraint that impacts global concurrency or inflicts hidden dependencies cause severe synchronization problems. This includes, among others, the *Chain*, *Absence*, and *Existence* constraints.
- Although their principle is simple, *Not co-existence* and especially *Exclusive choice* are very hard to incorporate in a mixed-paradigm model.

6 A Step-Wise Approach for Mixed-Paradigm Modeling with Intertwined State Spaces

6.1 Introduction

Based on the insights gathered from previous sections, we now propose a step-wise approach for modeling mixed-paradigm models. The insights relate directly to the different characteristics discussed in Sect. 5. The table can be used by mixed-paradigm modelers to assess the influence of certain constraints on the model and what the consequences of using them might entail. By applying the scores,

Table 1 Scorecard of Declare constraints for the number of R/I-net constructs needed, and semantic characteristics

	R/I-net constructs					Impact global concurrency (2)	Impact global timing (5)	Permanently disabling (10)	Interchangeability score
	P/T	A	R	I	Sum				
Init	1			|T|-1	|T|				|T|
Last			|P|		|P|		✓		|P|+5
Existence	1	2			3		✓		**8**
Absence	1	2			3			✓	13
Exactly	1	2		1	4		✓	✓	19
Response	1	1	1	1	4		✓		**9**
Precedence	1	3			4				**4**
Succession	2	4	1	1	8		✓		13
Alternate response	2	2	2	1	7		✓		12
Alternate precedence	1	2	1		4				**4**
Alternate succession	2	4		1	7		✓		12
Chain response	1	1	1	|T|-1	|T|+2	✓	✓		|T|+9
Chain precedence	1	|T|-1	1	1	|T|+2	✓			|T|+4
Chain succession	2	2	2	|T|	|T|+6	✓	✓		|T|+13
Responded Existence	2+1T	3	2	1	9		✓		14
Co-existence	3+1T	6	3	1	14		✓		19
Not succession	1	1		1	3			✓	13
Not chain succession	1		|T|-2	1	|T|	✓			|T|+2
Not co-existence	2	2		2	6			✓	16
Choice	1		2	1	4		✓		**9**
Exclusive choice	3	2	2	3	10		✓	✓	25

$|T|$ and $|P|$ stand for the number of transitions and places in the model respectively

The ticks indicate whether a certain property holds for the constraint

it now also becomes possible to objectively start measuring different mixed-paradigm solutions in terms of comprehensibility (in terms of the amount of model constructs), and the semantic difficulties that are introduced.

1. *Determine for each activity whether its behavior can be contained in a procedural workflow, or rather requires a looser setup with rules.* By indicating where in the process an activity can occur, it will reveal the extent to which it requires flexibility.
   - If the position of the activity is not fixed within the workflow, it is better to exclude it from the procedural model.
   - If the activity occurs a predefined number of times, Petri nets might be used, or a *Unary* Declare constraint. Otherwise, it may prove hard to use a token game around the activity, as an undesired amount of tokens might be pushed down the model, which could require adding silent transitions to model skipping steps.

2. *Determine which relationships are needed between the different model types.* In a mixed-model, there are 4 different types of relationships, given that activities are labeled 'Declarative' or 'Procedural' in step 1:
   - Declarative-Declarative,
   - Declarative-Procedural,
   - Procedural-Declarative,
   - and Procedural-Procedural.

   The second and third types constitute the real mixed cases. In the case of using them, it is advisory to consult Table 1 to check for characteristics towards violation and temporal issues. Generally, it is advised to avoid using binary Declare rules between activities solely present in the procedural part of a mixed-model. Although it is possible to do this, it is better to approach the procedural part from the outside to avoid internal anomalies such as deadlocks. Only the construction of the state space of a Petri net can show whether the resolution of, e.g., temporary violations is still possible. Hence, avoid constraints that have, e.g., a global impact on concurrency and timing. Also, hidden dependencies propagate through the procedural model. Therefore, these constraints are best used in isolation within a declarative model. Safe connections between

procedural and declarative parts are mainly *Precedence* relations, and any constraint that does not impact global timing and hidden dependencies.

3. *Synchronize beginning and end points of both model types if possible.* By using *Init* and *Last* constraints in combination with a Petri net source and sink transition, the models are intertwined in a proper way. The inclusion of separate sink activities might be required.
4. *Check whether it is necessary to use two types of language constructs.* According to the scores in Table 1, several constraints are easy to model in Petri nets with R/I-net constructs. Replacing them, while still referring to them with their Declare constraint name, avoids multiple modeling notations. Furthermore, R/I-net constructs yield executable syntax, hence making the construction of an automaton obsolete in many cases (not, however, where there are hidden dependencies or multiple violation states).

6.2 Reworking an Existing Example with the Approach

In this section, we show how to transform a procedurally modeled order fulfillment process (Dumas et al. 2013, page 77), and expand it with declarative constructs. Also, it is shown where gaps still exist between the two approaches.

The setup of the order fulfillment process, however, is interpreted slightly differently than in Dumas et al. (2013). In this scenario, which can be found in Fig. 6, multiple orders can be made and at least three product shipments and payments have to have happened before the archiving of an order. Furthermore, the requests for raw materials can now only be done directly after checking their stock level, and obtaining the materials always has to happen directly after requesting them.

1. In the original model, every activity is rather fixed within the sequence. Due to the unspecified amount of occurrences of *Receive order* and its successors it becomes more interesting to use declarative constructs, as they are better capable of mixing different strings of activities while maintaining a somewhat structured process. Hence, everything up to *Confirm order* is rather declarative, while the shipping and invoicing processes are kept procedural.
2. Some relationships, as indicated in Table 1, are easier to express in Declare. Most notably, the use of *Chain* relationships to indicate directly follows parity, and the use of *Alternate precedence* for an unspecified amount of occurrences of activities around *Receive order* are more convenient and avoid the model becoming convoluted. It is more tedious to express that *Archive order* needs at least three occurrences of *Ship product* and *Receive payment* in Declare, as it is harder to count in regular languages than in Petri nets (requirements such as $a^n b^n$). Also, it is clearer to do so by keeping track of the tallying with tokens. Finally, some Declare constraints are used to connect the material and invoicing and shipping parts.
3. Beginning and end points are synchronized through the *Init* and *Last* constraints. In this case, the *Last* constraint for *Archive order* has a global impact on the declarative part of the model as well, most notably on *Manufacture product*.
4. As can be seen in Table 1, the *Precedence* constraints can be expressed with R/I-net constructs. *Not succession*, however, requires special care in this case, as it has an impact on dependent activities both in the declarative part as well as in the procedural part due to propagation of dependency (disabling *Receive order* also disables all succeeding activities in a *Precedence* relationship).

In the end, using different syntaxes in mixed-paradigm models is also of interest as it can better indicate which parts of the model are procedural, and which ones are declarative.

7 Conclusion and Future Work

This paper explored the gap between procedural and declarative process modeling approaches, focusing on understandability, syntax, and execution semantics. More specifically, the authors looked at the possibilities that arise when combining both paradigms with intertwined state spaces. Overall, it is found that there is a trade-off between syntax that yields execution semantics, and verbose Declare constraints with many implications for execution. A scoring table for Declare constraints is presented, which can be used for objectively assessing the complexity of mixed-models, enabling the comparison of different mixed-paradigm solutions and guiding modelers when selecting appropriate constructs. Finally, a step-wise approach is proposed for mixed-paradigm modeling, for which an example is elaborated in which the trade-offs that exist are illustrated and made explicit.

Future work will entail integrating the insights into different process languages, such as BPMN, for constructing mixed-paradigm models with high readability and applications for business users. Furthermore, it will be investigated how this approach might simplify model collections and how to elaborate more extensive examples. Finally, tool support for transforming and reducing mixed-paradigm models will be pursued, based on the guidelines in this paper.

References

Adams M, Ter Hofstede AHM, Edmond D, van der Aalst WMP (2006) Worklets: a service-oriented implementation of dynamic flexibility in workflows. In: On the move to meaningful internet systems 2006: CoopIS, DOA, GADA, and ODBASE. Springer, pp 291–308

Cortadella J, Kishinevsky M, Lavagno L, Yakovlev A (1998) Deriving Petri nets from finite transition systems. Comput IEEE Trans 47(8):859–882

De Smedt J, vanden Broucke SKLM, De Weerdt J, Vanthienen J (2015) A full R/I-net construct lexicon for declare constraints. Research report KBI 1506

Di Ciccio C, Mecella M (2013) A two-step fast algorithm for the automated discovery of declarative workflows. In: Computational intelligence and data mining (CIDM), 2013 IEEE Symposium on IEEE, pp 135–142

Dijkman RM, Dumas M, Ouyang C (2008) Semantics and analysis of business process models in BPMN. Inf Softw Technol 50(12):1281–1294

Dumas M, La Rosa M, Mendling J, Reijers HA (2013) Fundamentals of business process management. Springer

Fahland D (2007) Towards analyzing declarative workflows. Auton Adapt Web Serv 7061:6

Fahland D, Lübke D, Mendling J, Reijers H, Weber B, Weidlich M, Zugal S (2009) Declarative versus imperative process modeling languages: the issue of understandability. In: Enterprise, business-process and information systems modeling. Springer, pp 353–366

Goedertier S, Vanthienen J, Caron F (2013) Declarative business process modelling: principles and modelling languages. Enterp Inf Syst pp 1–25 (ahead-of-print)

Haisjackl C, Barba I, Zugal S, Soffer P, Hadar I, Reichert M, Pinggera J, Weber B (2014) Understanding declare models: strategies, pitfalls, empirical results. Softw Syst Model, pp 1–28

Hildebrandt T, Mukkamala RR, Slaats T (2012) Nested dynamic condition response graphs. In: Fundamentals of software engineering. Springer, pp 343–350

Hull R, Damaggio E, Fournier F, Gupta M, Heath III FT, Hobson S, Linehan M, Maradugu S, Nigam A, Sukaviriya P et al (2011) Introducing the guard-stage-milestone approach for specifying business entity lifecycles. In: Web services and formal methods. Springer, pp 1–24

Maggi FM, Westergaard M, Montali M, van der Aalst WMP (2012) Runtime verification of LTL-based declarative process models. In: Runtime verification. Springer, pp 131–146

Murata T (1989) Petri nets: properties, analysis and applications. Proc IEEE 77(4):541–580

Pesic M (2008) Constraint-based workflow management systems: shifting control to users. PhD thesis, Technische Universiteit Eindhoven

Pesic M, Schonenberg H, van der Aalst WMP (2007) Declare: full support for loosely-structured processes. In: Enterprise distributed object computing conference, 2007. EDOC 2007. 11th IEEE International, IEEE, pp 287–298

Pesic M, van der Aalst WMP (2006) A declarative approach for flexible business processes management. In: Business process management workshops. Springer, pp 169–180

Prescher J, Di Ciccio C, Mendling J (2014) From declarative processes to imperative models. In: Proceedings of the 4th international symposium on data-driven process discovery and analysis (SIMPDA 2014), Milan, pp 162–173

Reijers HA, Slaats T, Stahl C (2013) Declarative modeling–an academic dream or the future for BPM? In: Business process management. Springer, pp 307–322

Rosemann M, Recker J, Indulska M, Green P (2006) A study of the evolution of the representational capabilities of process modeling grammars. In: Advanced information systems engineering. Springer, pp 447–461

Sadiq S, Sadiq W, Orlowska M (2001) Pockets of flexibility in workflow specification. In: Conceptual modelingER 2001. Springer, pp 513–526

Schonenberg H, Ronny M, Nick R, Nataliya M, van der Aalst WMP (2008) Towards a taxonomy of process flexibility. In: CAiSE forum, vol 344, pp 81–84

van der Aalst WMP (1999) Formalization and verification of event-driven process chains. Inf Softw Technol 41(10):639–650

van der Aalst WMP (2002) Making work flow: on the application of petri nets to business process management. In: Application and theory of petri nets 2002. Springer, pp 1–22

van der Aalst WMP (2013) A comprehensive survey. ISRN Software Engineering, Business process management

van der Aalst WMP, Adams M, Ter Hofstede AHM, Pesic M, Schonenberg H (2009) Flexibility as a service. In: Database systems for advanced applications. Springer, pp 319–333

van der Aalst WMP, Ter Hofstede AHM (2005) YAWL: yet another workflow language. Inf Syst 30(4):245–275

Westergaard M, Slaats T (2013) Mixing paradigms for more comprehensible models. In: Business process management. Springer, pp 283–290

Westergaard M, Stahl C, Reijers HA (2013) UnconstrainedMiner: efficient discovery of generalized declarative process models. Technical Report BPM-13-28, BPMcenter

White SA (2004) Introduction to BPMN. IBM Corporation, vol 2