Information Systems xxx (xxxx) xxx

Contents lists available at ScienceDirect

Information Systems

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

# Formalising and animating multiple instances in BPMN collaborations

Flavio Corradini, Chiara Muzi, Barbara Re *, Lorenzo Rossi, Francesco Tiezzi

*School of Science and Technology, University of Camerino, Camerino, Italy*

ARTICLE INFO

*Article history:*
Received 4 February 2019
Received in revised form 27 June 2019
Accepted 21 October 2019
Available online xxxx
Recommended by Gottfried Vossen

*Keywords:*
BPMN 2.0
Multiple instances
Data
Formal semantics
Animation

ABSTRACT

The increasing adoption of modelling methods contributes to a better understanding of the flow of processes, from the internal behaviour of a single organisation to a wider perspective where several organisations exchange messages. In this regard, BPMN collaborations provide a suitable modelling abstraction. Even if this is a widely accepted notation, only a limited effort has been expended in formalising its semantics, especially for what it concerns the interplay among control features, data handling and exchange of messages in scenarios requiring multiple instances of interacting participants. In this paper, we face the problem of providing a formal semantics for BPMN collaborations including elements dealing with multiple instances, i.e., multi-instance pools and sequential/parallel multi-instance tasks. For an accurate account of these features, it is necessary to consider the data perspective of collaboration models, thus supporting data objects, data collections and data stores, and different execution modalities of tasks concerning atomicity and concurrency. Beyond defining a novel formalisation, we also provide a BPMN collaboration animator tool, named MIDA, faithfully implementing the formal semantics. MIDA can also support designers in debugging multi-instance collaboration models.



## 1. Introduction

Nowadays, modelling is recognised as an important practice in supporting the continuous improvement of collaborative information systems. This demands for a clear understanding of interactions and data exchanges among participants. To ensure proper carrying out of such interactions, the participants should be provided with enough information about the messages they must or may send in a given context. This is particularly important when multiple instances of interacting participants are involved. In this regard, BPMN [1] collaboration diagrams result to be an effective way to reflect how multiple participants cooperate to reach a shared goal.

Even if widely accepted, a major drawback of BPMN is related to the complexity of the semi-formal definition of its meta-model and the possible misunderstanding of its execution semantics defined by means of natural text description, sometimes containing misleading information [2]. This becomes a more prominent issue as we consider BPMN supporting tools, such as animators, simulators and enactment tools, whose implementation of the execution semantics may not be compliant with the standard and be different from each other, thus undermining models portability and tools effectiveness.

To overcome these issues, several formalisations have been proposed, mainly focussing on the control flow perspective (e.g., [3–10]). Less attention has been paid to provide a formal semantics capturing the interplay between control features, message exchanges, and data. These perspectives are strongly related, especially when multi-instance participants have to interact. In fact, to achieve successful collaboration interactions, it is required to deliver the messages arriving at the receiver side to the appropriate instances. As messages are used to exchange data between participants, the BPMN standard fosters the use of the content of the messages themselves to correlate them with the corresponding instances. Thus, the data perspective plays a crucial role when considering multi-instance collaborations. Despite this, no formal semantics that considers all together these key aspects of BPMN collaboration models has been yet proposed in the literature.

In this work, we aim at filling this gap by answering the following research questions:

**RQ1:** What is the precise semantics of multi-instance BPMN collaborations?

**RQ2:** Can supporting tools assist designers to spot erroneous behaviours related to multiple instantiation and data handling in BPMN collaborations?

To answer RQ1, we provide an **operational semantics of BPMN collaboration models including multi-instance elements, while taking into account the data perspective.** In particular, besides multi-instance pools, we support multi-instance tasks with different execution modalities, resulting from the

* Corresponding author.
*E-mail addresses:* flavio.corradini@unicam.it (F. Corradini), chiara.muzi@unicam.it (C. Muzi), barbara.re@unicam.it (B. Re), lorenzo.rossi@unicam.it (L. Rossi), francesco.tiezzi@unicam.it (F. Tiezzi).

https://doi.org/10.1016/j.is.2019.101459
0306-4379/

combination of parameters concerning atomicity, concurrency and sequentialisation/parallelisation of task instances. Moreover, we include all kinds of data elements provided by BPMN: *data objects* and *data collections*, which are local to process instances, and *data stores*, persistently storing data shared among different instances. Besides being useful per se, as it provides a precise understanding of the ambiguous and loose points of the standard, a main benefit of this formalisation is that it paves the way for the development of tools supporting model analysis.

To answer RQ2, we go beyond the mere formalisation, by developing an **animator tool that faithfully implements the proposed formal semantics and visualises the execution of multi-instance collaborations**. It is indeed well recognised that process animators play an important role in enhancing the understanding of business processes behaviour [11] and that, to this aim, the faithful correspondence with the semantics is essential [12], although it is not always supported [13]. Our tool, named MIDA, supports model designers in achieving a priori knowledge of collaborations behaviour, in terms of executed activities, exchanged messages, and evolution of data values for each active instance. This allows designers to **debug** their collaboration models. In this way, they can detect, and hence prevent, undesired executions, where e.g., a control flow is blocked or an erroneous interaction arises. Designers can deduce the cause beyond them by checking the evolution of tokens distribution and of involved data. MIDA animation features result helpful both in educational contexts, for explaining the behaviour of BPMN elements, and in practical modelling activities.

This work is based on the paper "Animating Multiple Instances in BPMN Collaborations: From Formal Semantics to Tool Support" [14], published in the proceedings of the 16th International Conference on Business Process Management. Besides describing our approach in greater detail, this article extends the scope of the original conference paper as following reported.

1. We have enriched the proposed formalisation to include: sequential and parallel multi-instance tasks; different execution modalities for tasks; and data object collections and data stores. We have hence extended, and also streamlined, the formal definitions of our syntax and semantics.
2. We have extended the MIDA tool to support the novel elements introduced in the formalisation, and improved its usability.
3. We have considered a new running example, incrementally enriched to better illustrate the effectiveness and potentialities of our formalisation and tool.

The rest of the paper is organised as follows. Section 2 provides the motivations underlying the work, and presents the running example. Section 3 introduces the formal framework at the basis of our approach. Section 4 shows our formalisation at work on both typical and tricky multi-instance interaction scenarios. Section 5 illustrates how the formal concepts have been practically realised in the MIDA 2.0 tool. Section 6 compares our work with the related ones. Finally, Section 7 closes the paper with lessons learned and opportunities for future work.

## 2. The interplay between multiple instances, messages and data objects in BPMN collaborations

To deal with multiple instances in BPMN collaboration models, it is necessary to take into account the data flow. Indeed, the dynamic creation of process *instances* can be triggered by the arrival of *messages*, which contain data. Within a process instance, data can be accessed from *data objects, data collections and data stores* (here, and in the following, we use the term *data elements* to refer to all of them together), and drives the instance execution. Values of data elements can be used to fill the content of outgoing messages and, vice versa, the content of incoming messages can be stored in data elements. We clarify below the interplay between such concepts. To this aim, we introduce a BPMN collaboration model, used as a running example throughout the paper, concerning the preparation of a cake.

The example in Fig. 1 illustrates the collaboration between a *Pastry Chef* and his *Assistants* in the preparation of a three-layer cake with decorations, as requested by a *Customer*. The example is used throughout the paper to illustrate in detail the characteristics of the proposed approach.

The collaboration model combines the activities of the involved participants as following. The *Customer* provides details about the desired cake and checks the final result. The *Pastry Chef* coordinates the activities of the *Assistants*, combines the layers to assemble the cake, and delivers it to the *Customer*. Finally, the *Assistant* is the one in charge to prepare the layers of the cake. Since more than one *Assistant* is involved, each of them is modelled as a process instance of a multi-instance pool (in our example we have three *Assistants*, one for each layer of the cake). The collaboration is started by the *Customer*, who sends to the *Pastry Chef* a cake request including the information about the cake layers. This information, specified as a data input in the *Customer* process, concerns a description of the three layers of the cake, i.e., top, middle and bottom, and for each layer specifies the colour of the icing, i.e., brown, blue and pink. The request, initially stored in the *Layers Info* data object, is checked and then rearranged by the *Pastry Chef* in the *Layers Plan* data collection. Then, the *Pastry Chef* activates the *Assistants* by assigning a layer (i.e., an item of the *Layers Plan* collection) to each of them, via a parallel multi-instance task with loop cardinality set to three, according to the number of involved *Assistants* for the cake preparation. In this way, we will have three *Assistants* working in parallel: one on the bottom layer, one on the middle layer, and one on the top layer. Each *Assistant* immediately starts to prepare the assigned layer of the cake, and then waits to receive from the *Pastry Chef* the corresponding decorations to be applied. The decorations are indeed provided to the *Pastry Chef* by the *Customer* at a later time, and each kind of decoration must be properly associated to a specific cake layer. As soon as an *Assistant* has applied the received decorations, he sends back his decorated layer to the *Pastry Chef*. When all three layers are received by the *Pastry Chef* (via a sequential multi-instance task), the *Pastry Chef* combines the layers and provides the resulting cake to the *Customer*. The *Customer* checks the received cake: if it meets the expectation, both in term of layers assembly and in the combination of layers with decorations, he celebrates; otherwise he will be disappointed.

In this scenario, data support is crucial to precisely render the message exchanges between participants, especially because multiple instances of the *Assistant* process are created. In fact, messages coming into this pool might start a new process instance, or be routed to existing instances already underway. Messages and process instances must contain enough information to determine, when a message arrives at a pool, if a new process instance is needed or, if not, which existing instance will handle it. To this aim, BPMN makes use of the concept of *correlation*: it is up to each single message to provide the information that permits to associate the message with the appropriate (possibly new) instance. This is achieved by embedding values, called *correlation data*, in the content of the message itself. As reported in the standard, *"Correlation is used to associate a particular Message [...] between two particular Process instances. BPMN allows using existing Message data for correlation purposes [...] rather than requiring the introduction of technical correlation data"* [1, Sec. 8.3.2]. In particular, in our formalisation, we rely on pattern-matching to enable the correlation of exchanged messages. Considering our

Please cite this article as: F. Corradini, C. Muzi, B. Re et al., Formalising and animating multiple instances in BPMN collaborations, Information Systems (2019) 101459, https://doi.org/10.1016/j.is.2019.101459.

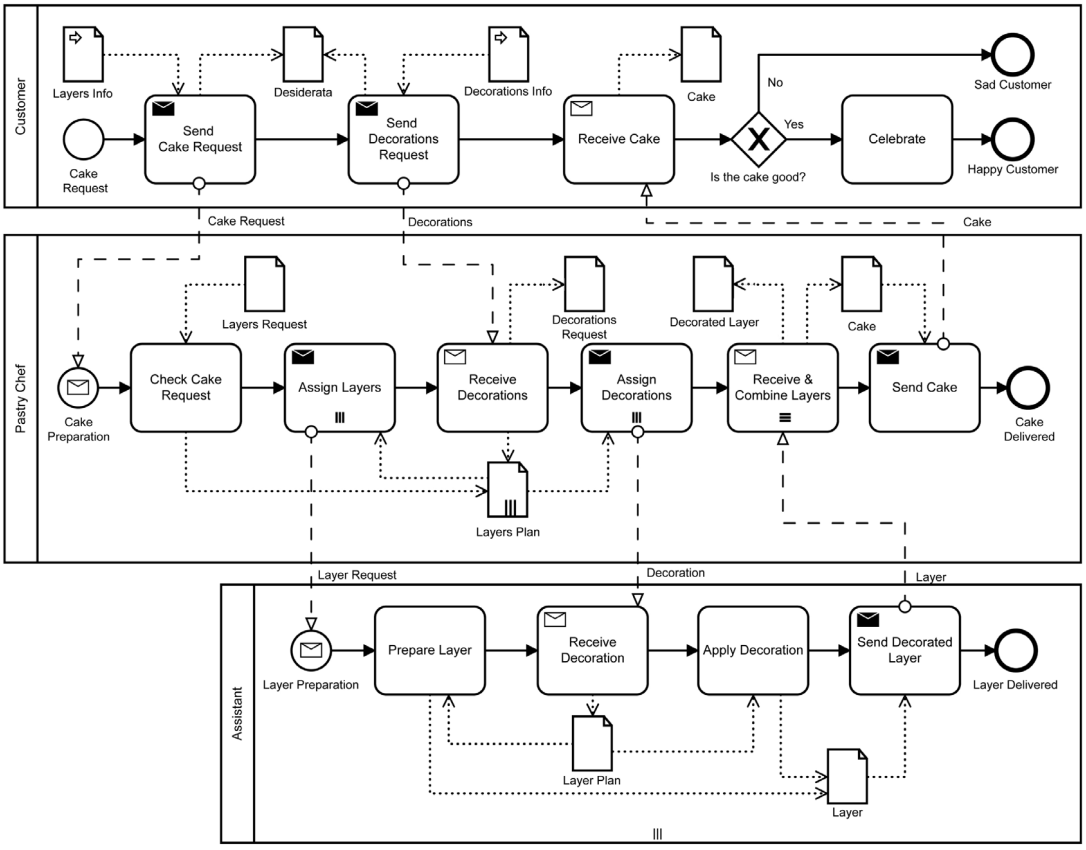



**Fig. 1.** Cake preparation collaboration model.

example, every time the *Pastry Chef* sends a kind of decorations to an *Assistant*, the message must contain information (in our case the layer position of the decorations) to be correlated to the correct process instance of the *Assistant* multi-instance pool. In this way, the decorations designed to be placed in the bottom layer will be properly delivered to the *Assistant* instance working on the bottom layer, the decorations for the middle layer will be delivered to the *Assistant* working on the middle layer, and so on.

According to the BPMN standard, data elements do not have any direct effect on the sequence flow or message flow of processes, since tokens do not flow along data associations [1, p. 221]. However, this statement is questionable. Indeed, on the one hand, the information stored in data elements can be used to drive the execution of process instances, as they can be referred in the conditional expressions of XOR split gateways to take decisions about which branch should be taken. On the other hand, data elements can be connected in input to tasks. In particular, the standard states that *"the Data Objects as inputs into the Tasks act as an additional constraint for the performance of those Tasks. The performers [...] cannot start the Task without the appropriate input"* [1, p. 183]. In both cases, a data element has an implicit indirect effect on the execution, since it can drive the decision taken by a XOR split gateway or acts as a guard condition on a task. In our running example, for instance, according to the value of the *Desiderata* data object, the conditional expression *Is the cake good?* is evaluated and a branch of the XOR split gateway is chosen. In particular, the task *Celebrate* can be executed only if the received cake fits with the desiderata of the *Customer* stored in the *Desiderata* data object. As another example, the task *Send Cake* in the *Pastry Chef* pool can be executed only if the fields of the *Cake* data object are filled.

Concerning the content of data elements, the standard left underspecified its structure, in order to keep the notation independent from the kind of data structure required from time to time. We consider here a generic record structure, assuming that a data object/store is just a list of fields, characterised by a name and the corresponding value. Instead, data collections are thought of as special data objects consisting of lists of elements that, in their own turn, are structured as list of fields. Fig. 2 reports the structure of the data elements used in our running example. Messages are structured as tuples of values; the latter can be manipulated and inserted into data elements fields via assignments performed by tasks.

Guards, assignments, and structure of data elements and messages are not explicitly reported in the graphical representation of the BPMN model, but are defined as attributes of the involved BPMN elements. We provide information on their definition and functioning in Section 3, and show how MIDA 2.0 users can specify them in Section 5.

## 3. A formal account of multi-instance collaborations

In this section we formalise the semantics of BPMN collaborations. We focus on those BPMN elements, informally presented in the previous section, that are strictly needed to deal with multiple instantiation of collaborations, namely multi-instance pools, message exchanges, multi-instance tasks (both in sequence and in parallel), data objects, data collections and data stores. Additionally, in order to define meaningful collaborations, we also consider some core BPMN elements (e.g., gateways and events).

To simplify the formal treatment of the semantics, we resort to a textual representation of BPMN models, which is more manageable for writing operational rules than the graphical notation. Notice that we do not propose an alternative modelling notation, but we just define a Backus–Naur Form (BNF) syntax of BPMN model structures.

Please cite this article as: F. Corradini, C. Muzi, B. Re et al., Formalising and animating multiple instances in BPMN collaborations, Information Systems (2019) 101459, https://doi.org/10.1016/j.is.2019.101459.

| | |
|---|---|
| LayersInfo {top, middle, bottom} | Desiderata {top, middle, bottom, cake} |
| DecorationsInfo {top, middle, bottom} | Cake {cake} |
| LayersRequest {top, middle, bottom} | DecorationsRequest {top, middle, bottom} |
| LayersPlan {position, layerColor, decorationColor} | |
| DecoratedLayer {layer} | Cake {cake, numLayers} |
| LayerPlan {position, layerColor, decorationColor} | Layer {layer, position, status} |

**Fig. 2.** Structures of data elements of the cake preparation example.

$$
\begin{array}{lcl}
C & ::= & \mathsf{pool}(\mathsf{p}, P) \mid \mathsf{miPool}(\mathsf{p}, P, \mathsf{max}) \mid C \parallel C \\
P & ::= & \mathsf{start}(\mathsf{e}, \mathsf{e}') \mid \mathsf{startRcv}(\mathsf{m}:\tilde{\mathsf{t}}, \mathsf{e}) \mid \mathsf{end}(\mathsf{e}) \mid \mathsf{endSnd}(\mathsf{e}, \mathsf{m}:\tilde{\mathsf{exp}}) \\
 & \mid & \mathsf{terminate}(\mathsf{e}) \mid \mathsf{eventBased}(\mathsf{e}, (\mathsf{m}_1:\tilde{\mathsf{t}}_1, \mathsf{e}_1), \ldots, (\mathsf{m}_h:\tilde{\mathsf{t}}_h, \mathsf{e}_h)) \\
 & \mid & \mathsf{andSplit}(\mathsf{e}, E) \mid \mathsf{xorSplit}(\mathsf{e}, G) \mid \mathsf{andJoin}(E, \mathsf{e}) \mid \mathsf{xorJoin}(E, \mathsf{e}) \\
 & \mid & \mathsf{interRcv}(\mathsf{e}, \mathsf{m}:\tilde{\mathsf{t}}, \mathsf{e}') \mid \mathsf{interSnd}(\mathsf{e}, \mathsf{m}:\tilde{\mathsf{exp}}, \mathsf{e}') \\
 & \mid & T \mid \mathsf{mipTask}(\mathsf{e}, \mathsf{exp}, T, \mathsf{c}, \mathsf{exp}', \mathsf{e}') \mid \mathsf{misTask}(\mathsf{e}, \mathsf{exp}, T, \mathsf{c}, \mathsf{exp}', \mathsf{e}') \\
 & \mid & P \parallel P \\
T & ::= & \mathsf{task}(\mathsf{e}, \mathsf{n}, M, \mathsf{exp}, A, \mathsf{e}') \mid \mathsf{taskRcv}(\mathsf{e}, \mathsf{n}, M, \mathsf{exp}, A, \mathsf{m}:\tilde{\mathsf{t}}, \mathsf{e}') \\
 & \mid & \mathsf{taskSnd}(\mathsf{e}, \mathsf{n}, M, \mathsf{exp}, A, \mathsf{m}:\tilde{\mathsf{exp}}, \mathsf{e}') \\
M & ::= & \mathsf{a} \mid N \\
N & ::= & \mathsf{na\_c} \mid \mathsf{na\_nc} \\
A & ::= & \epsilon \mid \mathsf{do.f} := \mathsf{exp} \mid \mathsf{ds.f} := \mathsf{exp} \mid \mathsf{get(do)} \mid \mathsf{push(do)} \mid \mathsf{A}, \mathsf{A}
\end{array}
$$

**Fig. 3.** BNF syntax of BPMN collaboration structures.

### 3.1. Textual notation of BPMN collaborations

We report in Fig. 3 the BNF syntax defining the textual notation of BPMN collaboration models. This syntax only describes the structure of models. Notably, even if our syntax would allow to write collaborations that cannot be expressed in BPMN, we consider here only those terms of the syntax that can be derived from BPMN models.

In the proposed grammar, the non-terminal symbols $C$, $P$, $T$, $M$, $N$ and $A$ represent *Collaboration Structures*, *Process Structures*, *Task Structures*, *Task Execution Modalities*, *Non-Atomic Execution Modalities*, and *Data Assignments*, respectively. The terminal symbols, denoted by the sansserif font, are the typical elements of a BPMN model, i.e., pools, events, tasks and gateways. The syntax of these elements is based on the following disjoint sets: the set $\mathbb{P}$ of *pool names* (ranged over by $\mathsf{p}$, $\mathsf{p}'$, ...); the set $\mathbb{E}$ of *sequence edges* (ranged over by $\mathsf{e}$, $\mathsf{e}'$, $\mathsf{e}_i$, ...) with $E \in 2^{\mathbb{E}}$ ranging over *sets of edges*; the set $\mathbb{M}$ of *message names* (ranged over by $\mathsf{m}$, $\mathsf{m}'$, $\mathsf{m}_i$, ...); the set $\mathbb{T}$ of *task names* (ranged over by $\mathsf{n}$, $\mathsf{n}'$, ...); the set $\mathbb{C}$ of *counter names* (ranged over by $\mathsf{c}$, $\mathsf{c}'$, ...); the set $\mathbb{D}$ of *data object names* (ranged over by $\mathsf{do}$, $\mathsf{do}'$, ...); the set of *data store names* (ranged over by $\mathsf{ds}$, $\mathsf{ds}'$, ...); the set of *data field names* (ranged over by $\mathsf{f}$, $\mathsf{f}'$, ...); the set $\mathbb{F}$ of *data fields* (ranged over by $\mathsf{do.f}$, $\mathsf{ds.f}'$, ...); and the set $\mathbb{V}$ of *values* (ranged over by $\mathsf{v}$, $\mathsf{v}'$, ...). We also use a set $\mathbb{EXP}$ of *expressions* (ranged over by $\mathsf{exp}$, $\mathsf{exp}'$, ...), whose precise syntax is deliberately not specified; we just assume that expressions contain, at least, values $\mathsf{v}$, data object fields $\mathsf{do.f}$ and data store fields $\mathsf{ds.f}$. Notation $\tilde{\cdot}$ denotes tuples; e.g., $\tilde{\mathsf{exp}}$ stands for a tuple of expressions $\langle \mathsf{exp}_1, \ldots, \mathsf{exp}_n \rangle$.

Intuitively, a BPMN collaboration model is rendered in our syntax as a collection of (single-instance and multi-instance) pools, each one specifying a process. Formally, a collaboration $C$ is a composition, by means of the operator $\parallel$, of pools either of the form $\mathsf{pool}(\mathsf{p}, P)$ (for single-instance pools) or $\mathsf{miPool}(\mathsf{p}, P, \mathsf{max})$ (for multi-instance pools), where $\mathsf{p}$ is the name that uniquely identifies the pool, $P$ is the enclosed process, and $\mathsf{max}$ is the maximum number of instances that can be activated in case of a multi-instance pool. Similarly, a process $P$ is a composition of process elements by means of the operator $\parallel$.

The correspondence between the graphical notation of BPMN and the textual representation used here is straightforward, except for the terms $\mathsf{mipTask}$ and $\mathsf{misTask}$ where $\mathsf{mip}$ and $\mathsf{mis}$ stand for *multiple-instance parallel* and *multiple-instance sequential*, respectively. We exemplify this correspondence by means of our running example in Fig. 4, where for the sake of readability we omit the definition of those guard expressions that simply check the initialisation of data fields (we have only kept $\mathsf{exp}_1$ as an example). For a more detailed account of the one-to-one correspondence the interested reader can refer to Appendix A. In the textual representation there is some information (content of messages, receiving templates, data element assignments, etc.) that is not reported in the graphical notation. In fact, for the sake of understandability, according to the BPMN standard these technical details of collaborations are not part of the graphical representation, but they are part of the low-level XML characterisation of the model. This information is explicitly reported in our textual representation as it is needed to properly define the execution semantics of the collaboration models.

Please cite this article as: F. Corradini, C. Muzi, B. Re et al., Formalising and animating multiple instances in BPMN collaborations, Information Systems (2019) 101459, https://doi.org/10.1016/j.is.2019.101459.

*Overall cake preparation collaboration scenario:*
$\mathsf{pool}(\mathsf{p}_c, P_c) \parallel \mathsf{pool}(\mathsf{p}_p, P_p) \parallel \mathsf{miPool}(\mathsf{p}_a, P_a, 3)$

*Customer process* :

$P_c = \mathsf{start}(\mathsf{e}_1, \mathsf{e}_2) \parallel \mathsf{taskSnd}(\mathsf{e}_2, \mathsf{SendCakeRequest}, \mathsf{a}, \mathsf{exp}_1, A_1, \mathsf{CakeRequest}:\tilde{\mathsf{exp}}_2, \mathsf{e}_3) \parallel$
$\mathsf{taskSnd}(\mathsf{e}_3, \mathsf{SendDecorationsRequest}, \mathsf{a}, \mathsf{exp}_3, A_2, \mathsf{Decorations}:\tilde{\mathsf{exp}}_4, \mathsf{e}_4) \parallel$
$\mathsf{taskRcv}(\mathsf{e}_4, \mathsf{ReceiveCake}, \mathsf{a}, \mathsf{true}, \epsilon, \mathsf{Cake}:\tilde{\mathsf{t}}_1, \mathsf{e}_5) \parallel$
$\mathsf{xorSplit}(\mathsf{e}_5, \{(\mathsf{e}_6, \mathsf{Cake.cake} \neq \mathsf{Desiderata.cake}), (\mathsf{e}_7, \mathsf{Cake.cake} = \mathsf{Desiderata.cake})\}) \parallel$
$\mathsf{end}(\mathsf{e}_6) \parallel \mathsf{task}(\mathsf{e}_7, \mathsf{Celebrate}, \mathsf{a}, \mathsf{true}, \epsilon, \mathsf{e}_8) \parallel \mathsf{end}(\mathsf{e}_8)$

*Templates*, *expressions*, *assignments* :

$\mathsf{exp}_1 =$ LayersInfo.top $\neq$ null and LayersInfo.middle $\neq$ null and LayersInfo.bottom $\neq$ null
$A_1 =$ Desiderata.top := LayersInfo.top, Desiderata.middle := LayersInfo.middle,
Desiderata.bottom := LayersInfo.bottom
$\tilde{\mathsf{exp}}_2 = \langle$LayersInfo.top, LayersInfo.middle, LayersInfo.bottom$\rangle$
$A_2 =$ Desiderata.cake := Desiderata.top + '&'+DecorationsInfo.top + 'on'+Desiderata.middle
+'&'+DecorationsInfo.middle + 'on'+Desiderata.bottom + '&'+DecorationsInfo.bottom
$\tilde{\mathsf{exp}}_4 = \langle$DecorationsInfo.top, DecorationsInfo.middle, DecorationsInfo.bottom$\rangle$
$\tilde{\mathsf{t}}_1 = \langle$?Cake.cake$\rangle$

*Pastry Chef process* :

$P_p = \mathsf{startRcv}(\mathsf{CakeRequest}:\tilde{\mathsf{t}}_{21}, \mathsf{e}_{21}) \parallel \mathsf{task}(\mathsf{e}_{21}, \mathsf{CheckCakeRequest}, \mathsf{a}, \mathsf{exp}_{21}, A_{21}, \mathsf{e}_{22}) \parallel$
$\mathsf{mipTask}(\mathsf{e}_{22}, 3, \mathsf{taskSnd}(\mathsf{e}'_{22}, \mathsf{AssignLayers}, \mathsf{a}, \mathsf{exp}_{22}, A_{22}, \mathsf{LayerRequest}:\tilde{\mathsf{exp}}_{23}, \mathsf{e}'_{23}), \mathsf{c}_1, \mathit{false}, \mathsf{e}_{23}) \parallel$
$\mathsf{taskRcv}(\mathsf{e}_{23}, \mathsf{ReceiveDecorations}, \mathsf{a}, \mathsf{exp}_{24}, A_{23}, \mathsf{Decorations}:\tilde{\mathsf{t}}_{22}, \mathsf{e}_{24}) \parallel$
$\mathsf{mipTask}(\mathsf{e}_{24}, 3, \mathsf{taskSnd}(\mathsf{e}'_{24}, \mathsf{AssignDecorations}, \mathsf{a}, \mathsf{exp}_{25}, A_{24}, \mathsf{Decoration}:\tilde{\mathsf{exp}}_{26}, \mathsf{e}'_{25}), \mathsf{c}_2, \mathit{false}, \mathsf{e}_{25}) \parallel$
$\mathsf{misTask}(\mathsf{e}_{25}, 3, \mathsf{taskRcv}(\mathsf{e}''_{25}, \mathsf{Receive\&CombineLayers}, \mathsf{a}, \mathsf{exp}_{27}, A_{25}, \mathsf{Layer}:\tilde{\mathsf{t}}_{23}, \mathsf{e}'_{26}), \mathsf{c}_3, \mathit{false}, \mathsf{e}_{26}) \parallel$
$\mathsf{taskSnd}(\mathsf{e}_{26}, \mathsf{SendCake}, \mathsf{a}, \mathsf{exp}_{28}, \epsilon, \mathsf{m}:\tilde{\mathsf{exp}}_{29}, \mathsf{e}_{27}) \parallel \mathsf{end}(\mathsf{e}_{27})$

*Templates*, *expressions*, *assignments* :

$\tilde{\mathsf{t}}_{21} = \langle$?LayersRequest.top, ?LayersRequest.middle, ?LayersRequest.bottom$\rangle$
$A_{21} =$ LayersPlan.position := 'bottom', LayersPlan.layerColor := LayersRequest.bottom, push(LayersPlan),
LayersPlan.position := 'middle', LayersPlan.layerColor := LayersRequest.middle, push(LayersPlan),
LayersPlan.position := 'top', LayersPlan.layerColor := LayersRequest.top, push(LayersPlan)
$A_{22} =$ get(LayersPlan)
$\tilde{\mathsf{exp}}_{23} = \langle$LayersPlan.layerColor, LayersPlan.position$\rangle$
$A_{23} =$ LayersPlan.position := 'bottom', LayersPlan.decorationColor := DecorationsRequest.bottom, push(LayersPlan),
LayersPlan.position := 'middle', LayersPlan.decorationColor := DecorationsRequest.middle, push(LayersPlan),
LayersPlan.position := 'top', LayersPlan.decorationColor := DecorationsRequest.top, push(LayersPlan)
$\tilde{\mathsf{t}}_{22} = \langle$?DecorationsRequest.top, ?DecorationsRequest.middle, ?DecorationsRequest.bottom$\rangle$
$A_{24} =$ get(LayersPlan)
$\tilde{\mathsf{exp}}_{26} = \langle$LayersPlan.position, LayersPlan.decorationColor$\rangle$
$A_{25} =$ Cake.cake := addLayer(DecoratedLayer.layer, DecoratedLayer.position), Cake.numLayers := Cake.numLayers + 1
$\tilde{\mathsf{t}}_{23} = \langle$?DecoratedLayer.layer, ?DecoratedLayer.position$\rangle$
$\mathsf{exp}_{28} =$ Cake.numLayers = 3
$\tilde{\mathsf{exp}}_{29} = \langle$Cake.cake$\rangle$

*Assistant process* :

$P_a = \mathsf{startRcv}(\mathsf{LayerRequest}:\tilde{\mathsf{t}}_{31}, \mathsf{e}_{31}) \parallel \mathsf{task}(\mathsf{e}_{31}, \mathsf{PrepareLayer}, \mathsf{a}, \mathsf{exp}_{31}, A_{31}, \mathsf{e}_{32}) \parallel$
$\mathsf{taskRcv}(\mathsf{e}_{32}, \mathsf{ReceiveDecoration}, \mathsf{a}, \mathsf{true}, \epsilon, \mathsf{Decoration}:\tilde{\mathsf{t}}_{32}, \mathsf{e}_{33}) \parallel$
$\mathsf{task}(\mathsf{e}_{33}, \mathsf{ApplyDecoration}, \mathsf{a}, \mathsf{exp}_{32}, A_{32}, \mathsf{e}_{34}) \parallel$
$\mathsf{taskSnd}(\mathsf{e}_{34}, \mathsf{SendDecoratedLayer}, \mathsf{a}, \mathsf{exp}_{33}, \epsilon, \mathsf{Layer}:\tilde{\mathsf{exp}}_{34}, \mathsf{e}_{35}) \parallel \mathsf{end}(\mathsf{e}_{35})$

*Templates*, *expressions*, *assignments* :

$\tilde{\mathsf{t}}_{31} = \langle$?LayerPlan.layerColor, ?LayerPlan.position$\rangle$
$A_{31} =$ Layer.status := 'prepared'
$\tilde{\mathsf{t}}_{32} = \langle$LayerPlan.position, ?LayerPlan.decorationColor$\rangle$
$\mathsf{exp}_{32} =$ Layer.status = 'prepared'
$A_{32} =$ Layer.status := 'decorated', Layer.position := LayerPlan.position,
Layer.layer := LayerPlan.layerColor + '&'+LayerPlan.decorationColor
$\mathsf{exp}_{33} =$ Layer.status = 'decorated'
$\tilde{\mathsf{exp}}_{34} = \langle$Layer.layer, Layer.position$\rangle$

**Fig. 4.** Textual representation of the running example.

Moreover, to support a compositional approach, in the textual notation each sequence/message edge in the graphical notation is split in two parts: the part outgoing from the source element and the part incoming into the target element; the two parts are correlated by the unique edge name.

We do not provide a direct syntactic representation of *data elements*, i.e., data objects, data collections and data stores. The evolution of their state during the model execution is a semantic concern (described later in this section). Thus, syntactically, only the connections between data elements and the other process elements are relevant. They are rendered by references within *expressions*, used to check when a task is ready to start (graphically, the task has an incoming data association from the data element), to update the values stored in a data field (graphically, the task has an outgoing data association to the data element), and to drive the decision of a XOR split gateway. The BPMN standard is quite loose in specifying what is the actual structure of data elements. We assume here a generic record structure for data objects and data stores, so that a data object/store is just a list of fields, characterised by a name and the corresponding value. Specifically, the field named f of the data object named do (resp. the data store named ds) is accessed via the usual notation do.f (resp. ds.f). A data collection instead is a special data object consisting of a list of elements that, in their own turn, are structured as list of fields. The head element of a data collection do can be retrieved by means of get(do); as effect of the execution of this action, the fields of the retrieved element can be accessed as usual by means of do.f. To add an element in a data collection do, first the fields of the new element are filled with values via assignments of the form do.f := exp, then the element with the filled fields is inserted in the tail of the data collection by means of push(do).

Since we explicitly consider data, messages are characterised not only by labels, but also by the values that they may carry.

Please cite this article as: F. Corradini, C. Muzi, B. Re et al., Formalising and animating multiple instances in BPMN collaborations, Information Systems (2019) 101459, https://doi.org/10.1016/j.is.2019.101459.

Therefore, a sending action specifies a list of expressions whose evaluation will return a tuple of values to be sent, while a receiving action specifies a template to select matching messages and possibly assign values to data fields. Formally, a *message* is a pair $\mathsf{m}:\tilde{\mathsf{v}}$, where $\mathsf{m}$ is the (unique) message name (i.e., the label of the message edge) and $\tilde{\mathsf{v}}$ is a tuple of values representing the payload of the message. Sending actions have as argument a pair of the form $\mathsf{m}:\tilde{\mathsf{exp}}$. Receiving actions have as argument a pair of the form $\mathsf{m}:\tilde{\mathsf{t}}$, where $\tilde{\mathsf{t}}$ denotes a *template*, that is a sequence of expressions and formal fields used as pattern to select messages received by the pool. Formal fields are data object/store fields identified by the ?-tag (e.g., ?do.f or ?ds.f) and are used to bind fields to values. Data elements are associated to a task by means of a conditional expression, which is a guard enabling the task execution, and a list of *assignments* $A$, each of which assigns the value of an expression to a data field or retrieves/inserts information in a data collection. When there is no data element as input to a task, the guard is simply true, while if there is no data element in output to a task the list of assignments is empty ($\epsilon$).

The XOR split gateway specifies *guard conditions* in its outgoing edges, used to decide which edge to activate according to the values of data objects. This is formally rendered as a function $G : \mathbb{E} \to \mathbb{EXP}$ mapping edges to conditional expressions. Notably, we assume that the set $\mathbb{EXP}$ of expressions includes the distinguished expression default referring to the *default sequence edge* outgoing from the gateway (it is assigned to at most one edge). When convenient, we will deal with function $G$ as a set of pairs $(\mathsf{e}, \mathsf{exp})$.

Finally, our formalisation supports the possibility of specifying the execution modality of tasks. This information is crucial when the data perspective and multi-instance tasks are taken into account. In case of a task with atomic execution (modality a), the evaluation of its enabling guard, the possible sending/receiving of a message, and the data object assignments, are performed atomically. This semantics fits well in many scenarios, like e.g., when a task acts on a data element representing a paper document managed by a human actor that cannot be accessed concurrently by other actors involved in the collaboration. However, there are also some situations where a non-atomic access is more suitable, e.g., when data elements represent shared digital documents. In the non-atomic case it is also important to indicate if the instances of a task can be executed concurrently (modality na_mc) or not (modality na_nc). Actually, the BPMN standard is intentionally loose on these points, in order to allow the use of the modelling language in different contexts of use. To more effectively support designers, we allow them to specify for each task the corresponding execution modality. This enables the identification of concurrency issues in those data accesses where they can actually arise and, at same time, it allows to not take into account such issues when in the reality they cannot occur. The role of task execution modalities is particularly crucial in those cases where tasks act in parallel and access the same data elements. Parallel execution of tasks can produce in these cases different effects. This depends on the execution order of the internal steps of tasks, i.e. guard checks, message sending/receiving, and data element assignments. Let us consider, for instance, a simple scenario with two parallel tasks, each of which makes an assignment producing a violation of the guard of the other task. If the two tasks are atomic, the execution of one of them will be deadlocked, while in the non-atomic case such deadlock can be avoided if both tasks perform the guard checks before making the assignments. Concurrent and non-concurrent non-atomic modalities play an active role mainly when the involved tasks are multi-instance.

### 3.2. Semantics of BPMN collaborations

The syntax presented so far represents the mere structure of processes and collaborations. To describe their semantics, we enrich the structural information with a notion of execution state, given by the marking of sequence edges with tokens [1, p. 27], the value of data elements, the status of tasks, and the exchanged messages. We call process configurations and collaboration configurations these stateful descriptions, which produce local and global effects, respectively, on the process and collaboration execution. The operational semantics at collaboration level is defined by means of a *labelled transition system* (LTS), whose definition relies on an auxiliary LTS on the behaviour of processes. We first present the process semantics and later the collaboration one.

A *process configuration* has the form $\langle P, \sigma_e, \sigma_{do}, \sigma_{dc}, \sigma_{ds}, \sigma_t, \sigma_c \rangle$, where:

- $P$ is a process structure;
- $\sigma_e : \mathbb{E} \to \mathbb{N}$ is a *sequence edge state function* specifying, for each sequence edge, the current number of tokens marking it ($\mathbb{N}$ is indeed the set of natural numbers);
- $\sigma_{do} : \mathbb{F} \to \mathbb{V}$ is a *data object state function* assigning values (possibly null) to data object fields;[1]
- $\sigma_{dc} : \mathbb{D} \to (\mathbb{F} \to \mathbb{V})^n$ is a *data collection state function* assigning to each data collection a tuple of data object state functions;
- $\sigma_{ds} : \mathbb{F} \to \mathbb{V}$ is a *data store state function* assigning values (possibly null) to data store fields; even if this state function has the same type of $\sigma_{do}$, we used two separate state functions because the information in data objects is treated differently from that in data stores, as this latter kind of data is permanent and shared among instances;
- $\sigma_t : \mathbb{T} \times \{a, s, r\} \to \mathbb{N}$ is a *task state function* used to keep track, for each non-atomic task, of the number of task instances in a given status, i.e., active ($a$), sending ($s$), and message received ($r$); the status of a task depends on its evolution (depicted in Fig. 5), where the inactive status formally corresponds to have zero instances for all other statuses;
- $\sigma_c : \mathbb{C} \to \mathbb{N}$ is a *counter state function* used to keep track, for each multi-instance task, of the number of time that the task has still to be executed.

For the sake of presentation, in the following we will use notation $\sigma_d$ to denote in a compact way the triple $(\sigma_{do}, \sigma_{dc}, \sigma_{ds})$ representing the state of all data elements. Thus, we will write, e.g., a process configuration as $\langle P, \sigma_e, \sigma_d, \sigma_t, \sigma_c \rangle$. We denote by $\sigma_e^0$ (resp. $\sigma_d^0, \sigma_t^0, \sigma_c^0$) the edge (resp. data element, task and counter) state where all edges are unmarked (resp. all data object/store fields are set to null, all data collections are empty, all tasks are inactive, and all counters are set to 0). Formally, $\sigma_e^0(\mathsf{e}) = 0$ $\forall \mathsf{e} \in \mathbb{E}$, $\sigma_{do}^0(\mathsf{do.f}) = \mathsf{null}$ $\forall\, \mathsf{do.f} \in \mathbb{F}$, $\sigma_{dc}^0(\mathsf{do}) = \epsilon$ $\forall \mathsf{do} \in \mathbb{D}$, $\sigma_{ds}^0(\mathsf{ds.f}) = \mathsf{null}$ $\forall\, \mathsf{ds.f} \in \mathbb{F}$, $\sigma_t^0(\mathsf{n}, a) = \sigma_t^0(\mathsf{n}, s) = \sigma_t^0(\mathsf{n}, r) = 0$ $\forall \mathsf{n} \in \mathbb{T}$, and $\sigma_c^0(\mathsf{c}) = 0$ $\forall \mathsf{c} \in \mathbb{C}$. The state obtained by updating in $\sigma_e$ the number of tokens of the edge e to n, written as $\sigma_e \cdot [\mathsf{e} \mapsto \mathsf{n}]$, is defined as follows: $(\sigma_e \cdot [\mathsf{e} \mapsto \mathsf{n}])(\mathsf{e}')$ returns

[1] It is worth noticing that in our semantics we associate concrete values to data object fields. The same applies to data stores and data collections. This perfectly fits with our purpose of *animating* the execution of collaboration models showing the evolution of the specified data. A different approach is *simulation*, where batches of executions are performed to collect statistics suitable to enable quantitative analysis of process models [15, p. 235]. This can be achieved, e.g., by ranging the values of data fields for selecting different branches in the presence of XOR gateways, by relying on probability distributions for specifying task durations to consider different interleavings, or by specifying resources allocation. Simulation is indeed out of the scope of this paper and left as future work (see Section 7).

Please cite this article as: F. Corradini, C. Muzi, B. Re et al., Formalising and animating multiple instances in BPMN collaborations, Information Systems (2019) 101459, https://doi.org/10.1016/j.is.2019.101459.

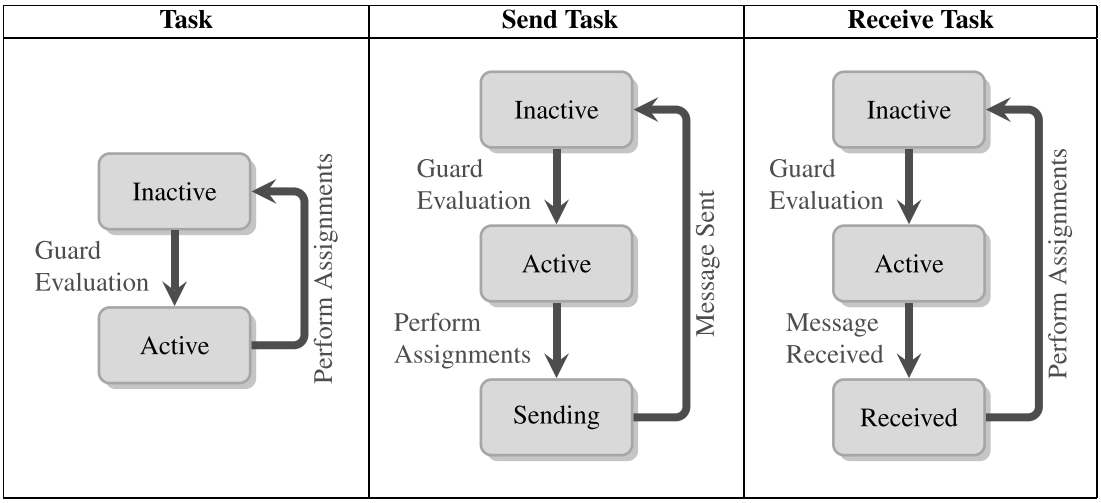



**Fig. 5.** Task status evolution.

n if $e' = e$, otherwise it returns $\sigma_e(e')$. The update of states $\sigma_{do}$, $\sigma_{dc}$, $\sigma_{ds}$, $\sigma_t$, and $\sigma_c$ are similarly defined. To simplify the definition of the operational rules, we introduce some auxiliary functions and relations to update states of process configurations (Fig. 6). Moreover, we will use function *edges*$(P)$ to get the set of all edges used in the process $P$, and function *in*$(T)$ (resp. *out*$(T)$) to get the edge incoming in (resp. outgoing from) the task $T$.

The auxiliary LTS on the behaviour of processes is a triple $\langle \mathcal{P}, \mathcal{L}, \rightarrow \rangle$ where: $\mathcal{P}$ is a set of process configurations; $\mathcal{L}$, ranged over by $\ell$, is a set of *labels*; and $\rightarrow \subseteq \mathcal{P} \times \mathcal{L} \times \mathcal{P}$ is a *transition relation*. We will write $\langle P, \sigma_e, \sigma_d, \sigma_t, \sigma_c \rangle \xrightarrow{\ell} \langle P, \sigma'_e, \sigma'_d, \sigma'_t, \sigma'_c \rangle$ to indicate that $(\langle P, \sigma_e, \sigma_d, \sigma_t, \sigma_c \rangle, \ell, \langle P, \sigma'_e, \sigma'_d, \sigma'_t, \sigma'_c \rangle) \in \rightarrow$, and say that 'the process in the configuration $\langle P, \sigma_e, \sigma_d, \sigma_t, \sigma_c \rangle$ can do a transition labelled by $\ell$ and evolve to the process configuration $\langle P, \sigma'_e, \sigma'_d, \sigma'_t, \sigma'_c \rangle$ in doing so'. To improve the readability of the operational rules, we apply the following simplifications on the notation of transitions.

**Notational simplifications.** We omit: *(i)* the states $\sigma_e$, $\sigma_d$, $\sigma_t$, $\sigma_c$ from the source configuration of transitions, since we use for them the same notation in all rules; *(ii)* the structure from the target configuration of transitions, since process execution only affects the current states and not the process structure; *(iii)* those states from the target configuration that are not affected by transitions. Thus, for example, a transition $\langle P, \sigma_e, \sigma_d, \sigma_t, \sigma_c \rangle \xrightarrow{\ell} \langle P, \sigma'_e, \sigma'_d, \sigma'_t, \sigma'_c \rangle$ will be written as $P \xrightarrow{\ell} \langle \sigma'_e \rangle$ when it simple affects the sequence edge state function.

The labels used by the process transition relation are generated by the following production rules:

$$\ell ::= \tau \mid !\mathsf{m}:\tilde{\mathsf{v}} \mid ?\,\mathsf{m}:\tilde{\mathsf{et}}, A \mid \mathit{new}\,\mathsf{m}:\tilde{\mathsf{et}} \qquad \tau ::= \epsilon \mid \mathit{kill}$$

The meaning of labels is as follows. Label $\tau$ denotes an action internal to the process, while $!\mathsf{m}:\tilde{\mathsf{v}}$ and $?\,\mathsf{m}:\tilde{\mathsf{et}}, A$ denote sending and receiving actions, respectively. Notation $\tilde{\mathsf{et}}$ denotes an evaluated template, that is a sequence of values and formal fields. Notably, the receiving label carries information about the data assignments $A$ to be executed after the message m is actually received. Label $\mathit{new}\,\mathsf{m}:\tilde{\mathsf{et}}$ denotes taking place of a receiving action that instantiates a new process instance (i.e., it corresponds to the occurrence of a start message event in a multi-instance pool). The meaning of internal actions is as follows: $\epsilon$ denotes an internal computation concerning the movement of tokens, while *kill* denotes taking place of the termination event.

The operational rules defining the transition relation of the process semantics are given by the inference rules in Figs. 7–11.

We now briefly comment on the rules in Fig. 7. Rule *P-Start* starts the execution of a process when it has been activated. To denote the enabled status of start events we have included in their syntactical definition an incoming (spurious) edge, named *enabling edge*. Thus, the process is activated when the enabling edge of a start event is marked. The effect of the rule is to increment the number of tokens in the edge outgoing from the start event and to decrease the marking of the enabling edge. Rule *P-End* instead is enabled when there is at least one token in the incoming edge of the end event, which is then simply consumed. Rule *P-Terminate* is similar, but it produces a *kill* label and forces the termination of the process instance by resetting the marking of edges and the status of tasks. Rule *P-StartRcv* starts the execution of a process by producing a label denoting the creation of a new instance and containing the information for consuming a received message at the collaboration layer (see rule *C-CreateMi* in Fig. 13). Rule *P-EndSnd* is enabled when there is at least a token in the incoming edge of the end event, which is then removed. Moreover, a send label is produced in order to deliver the produced message at the collaboration layer (see rule *C-DeliverMi* in Fig. 13). Rules *P-InterRcv* and *P-InterSnd* are enabled when there is at least a token in their incoming edge and move it to their outgoing edge, while producing a receive or a send label, respectively. Rule *P-AndSplit* is applied when there is at least one token in the incoming edge of an AND split gateway; as result of its application, the rule decrements the number of tokens in the incoming edge, and increments the tokens in each outgoing edge. Rule *P-XorSplit*$_1$ is applied when a token is available in the incoming edge of a XOR split gateway and a conditional expression of one of its outgoing edges is evaluated to *true*; the rule decrements the token in the incoming edge and increments the token in the selected outgoing edge. Notably, if more edges have their guards satisfied, one of them is non-deterministically chosen. Rule *P-XorSplit*$_2$ is applied when all guard expressions are evaluated to *false*; in this case the default edge is marked. Rule *P-AndJoin* decrements the tokens in each incoming edge and increments the number of tokens of the outgoing edge, when each incoming edge has at least one token. Rule *P-XorJoin* is activated every time there is a token in one of the incoming edges, which is then moved to the outgoing edge. Rule *P-EventG* is activated when there is a token in the incoming edge and there is a message $\mathsf{m}_j$ to be consumed, so that the application of the rule moves the token from the incoming edge to the outgoing edge corresponding to the received message. A label corresponding to the consumption of a message is observed.

Please cite this article as: F. Corradini, C. Muzi, B. Re et al., Formalising and animating multiple instances in BPMN collaborations, Information Systems (2019) 101459, https://doi.org/10.1016/j.is.2019.101459.

| | |
|---|---|
| $inc(\sigma_e, \mathsf{e}) = \sigma_e \cdot [\mathsf{e} \mapsto \sigma_e(\mathsf{e}) + 1]$ | increments by one the number of tokens marking the edge e in the state $\sigma_e$ |
| $dec(\sigma_e, \mathsf{e}) = \sigma_e \cdot [\mathsf{e} \mapsto \sigma_e(\mathsf{e}) - 1]$ | decrements by one the number of tokens marking the edge e in the state $\sigma_e$ |
| $inc(\sigma_e, E)$ (resp. $dec(\sigma_e, E)$) | increments (resp. decrements) by one the number of tokens marking all edges in $E$ in the state $\sigma_e$ |
| $reset(\sigma_e, \mathsf{e}) = \sigma_e \cdot [\mathsf{e} \mapsto 0]$ | sets to zero the number of tokens marking the edge e in the state $\sigma_e$ |
| $reset(\sigma_e, E)$ | resets all edges in $E$ in the state $\sigma_e$ |
| $reset(\sigma_e) = \sigma_e^0$ | resets all edges in the state $\sigma_e$ |
| $set(\sigma_e, \mathsf{e}, h) = \sigma_e \cdot [\mathsf{e} \mapsto h]$ | sets to $h$ the tokens marking the edge e in the state $\sigma_e$ |
| $eval(\mathsf{exp}, \sigma_d, \mathsf{v})$ | states that v is one of the possible values resulting from the evaluation of the expression exp on the data element state $\sigma_d$; this is a relation, because an expression may contain non-deterministic operators, and is not explicitly defined, since the syntax of expressions is deliberately not specified (we only assume that $eval(\mathsf{default}, \sigma_d, \mathsf{v})$ implies $\mathsf{v} = false$ for any $\sigma_d$) |
| $eval(\tilde{\mathsf{exp}}, \sigma_d, \tilde{\mathsf{v}})$ and $eval(\tilde{\mathsf{t}}, \sigma_d, \tilde{\mathsf{et}})$ | evaluate tuples of expressions and templates, resp. |
| $upd(\sigma_d, A, \sigma'_d)$ | states that $\sigma'_d$ is one of the possible states resulting from the update of $\sigma_d$ with assignment $A$ |
| $inc(\sigma_t, \mathsf{n}, \mathsf{S}) = \sigma_t \cdot [(\mathsf{n}, \mathsf{S}) \mapsto \sigma_t(\mathsf{n}, \mathsf{S}) + 1]$ | increments by one the number of instances of task n in the status $\mathsf{S} \in \{a, s, r\}$ in the state $\sigma_t$ |
| $dec(\sigma_t, \mathsf{n}, \mathsf{S}) = \sigma_t \cdot [(\mathsf{n}, \mathsf{S}) \mapsto \sigma_t(\mathsf{n}, \mathsf{S}) + 1]$ | decrements by one the number of instances of task n in the status $\mathsf{S} \in \{a, s, r\}$ in the state $\sigma_t$ |
| $reset(\sigma_t) = \sigma_t^0$ | sets to inactive the status of all tasks in the state $\sigma_t$ |
| $isInactive(\sigma_t, \mathsf{n}) = (\sigma_t(\mathsf{n}, a) = 0 \wedge \sigma_t(\mathsf{n}, s) = 0 \wedge \sigma_t(\mathsf{n}, r) = 0)$ | returns $true$ if the task n is in the inactive status in the state $\sigma_t$ |
| $set(\sigma_c, \mathsf{c}, h) = \sigma_c \cdot [\mathsf{c} \mapsto h]$ | sets to $h$ the value of the counter c in the state $\sigma_c$ |
| $reset(\sigma_c, \mathsf{c}) = \sigma_c \cdot [\mathsf{c} \mapsto 0]$ | resets the value of the counter c in the state $\sigma_c$ |
| $dec(\sigma_c, \mathsf{c}) = \sigma_c \cdot [\mathsf{c} \mapsto \sigma_e(\mathsf{c}) - 1]$ | decrements by one the value of c in the state $\sigma_e$ |

**Fig. 6.** BPMN process semantics: auxiliary functions and relations for updating states (the formal definitions here omitted for the sake of readability are reported in Appendix B).

Rules in Fig. 8 deal with task with atomic execution; we show how this requirement can be relaxed later in this section. Rule *P-Task*$_A$ deals with non-communicating tasks, possibly equipped with data objects. It is activated only when the guard expression exp is satisfied and there is a token in the incoming edge, which is then moved to the outgoing edge. The rule also updates the values of the data objects connected in output to the task by performing the assignments *A*. Rule *P-TaskRcv*$_A$ is similar, but it produces a label corresponding to the consumption of a message. In this case, however, the data updates are not executed, because they must be done only after the message is actually received; therefore, the assignments are passed by means of the label to the collaboration layer (see rule *C-ReceiveMi* in Fig. 13). Rule *P-TaskSnd*$_A$ sends a message, updates the data object and moves the incoming token to the outgoing edge. The produced send label is used to deliver the message at the collaboration layer (see rule *C-DeliverMi* in Fig. 13).

Rules in Fig. 9 deals with task with non-atomic execution, with both concurrent and non-concurrent modality. According to the evolution of the task status, shown in Fig. 5, the execution of non-communicating tasks is split in two steps: task activation (rule *P-Task*$_{N1}$), dealing with the evaluation of the guard

Please cite this article as: F. Corradini, C. Muzi, B. Re et al., Formalising and animating multiple instances in BPMN collaborations, Information Systems (2019) 101459, https://doi.org/10.1016/j.is.2019.101459.

| | | |
|---|---|---|
| $\mathsf{start}(\mathsf{e},\mathsf{e}') \xrightarrow{\epsilon} \langle inc(dec(\sigma_e,\mathsf{e}),\mathsf{e}')\rangle$ | $\sigma_e(\mathsf{e}) > 0$ | $(P\text{-}Start)$ |
| $\mathsf{end}(\mathsf{e}) \xrightarrow{\epsilon} \langle dec(\sigma_e,\mathsf{e})\rangle$ | $\sigma_e(\mathsf{e}) > 0$ | $(P\text{-}End)$ |
| $\mathsf{terminate}(\mathsf{e}) \xrightarrow{kill} \langle reset(\sigma_e), reset(\sigma_t)\rangle$ | $\sigma_e(\mathsf{e}) > 0$ | $(P\text{-}Terminate)$ |
| $\mathsf{startRcv}(\mathsf{m}:\tilde{\mathsf{t}},\mathsf{e}) \xrightarrow{new\ \mathsf{m}:\tilde{\mathsf{et}}} \langle inc(\sigma_e,\mathsf{e})\rangle$ | $eval(\tilde{\mathsf{t}},\sigma_d,\tilde{\mathsf{et}})$ | $(P\text{-}StartRcv)$ |
| $\mathsf{endSnd}(\mathsf{e},\mathsf{m}:\tilde{\mathsf{exp}}) \xrightarrow{!\mathsf{m}:\tilde{\mathsf{v}}} \langle dec(\sigma_e,\mathsf{e})\rangle$ | $\sigma_e(\mathsf{e}) > 0,$ $eval(\tilde{\mathsf{exp}},\sigma_d,\tilde{\mathsf{v}})$ | $(P\text{-}EndSnd)$ |
| $\mathsf{interRcv}(\mathsf{e},\mathsf{m}:\tilde{\mathsf{t}},\mathsf{e}') \xrightarrow{?\mathsf{m}:\tilde{\mathsf{et}},\epsilon} \langle inc(dec(\sigma_e,\mathsf{e}),\mathsf{e}')\rangle$ | $\sigma_e(\mathsf{e}) > 0,$ $eval(\tilde{\mathsf{t}},\sigma_d,\tilde{\mathsf{et}})$ | $(P\text{-}InterRcv)$ |
| $\mathsf{interSnd}(\mathsf{e},\mathsf{m}:\tilde{\mathsf{exp}},\mathsf{e}') \xrightarrow{!\mathsf{m}:\tilde{\mathsf{v}}} \langle inc(dec(\sigma_e,\mathsf{e}),\mathsf{e}')\rangle$ | $\sigma_e(\mathsf{e}) > 0,$ $eval(\tilde{\mathsf{exp}},\sigma_d,\tilde{\mathsf{v}})$ | $(P\text{-}InterSnd)$ |
| $\mathsf{andSplit}(\mathsf{e},E) \xrightarrow{\epsilon} \langle inc(dec(\sigma_e,\mathsf{e}),E)\rangle$ | $\sigma_e(\mathsf{e}) > 0$ | $(P\text{-}AndSplit)$ |
| $\mathsf{xorSplit}(\mathsf{e},\{(\mathsf{e}',\mathsf{exp})\}\cup G) \xrightarrow{\epsilon} \langle inc(dec(\sigma_e,\mathsf{e}),\mathsf{e}')\rangle$ | $\sigma_e(\mathsf{e}) > 0,$ $eval(\mathsf{exp},\sigma_d,true)$ | $(P\text{-}XorSplit_1)$ |
| $\mathsf{xorSplit}(\mathsf{e},\{(\mathsf{e}',\mathsf{default})\}\cup G) \xrightarrow{\epsilon} \langle inc(dec(\sigma_e,\mathsf{e}),\mathsf{e}')\rangle$ | $\sigma_e(\mathsf{e}) > 0,$ $\forall(\mathsf{e}_j,\mathsf{exp}_j)\in G\ .$ $eval(\mathsf{exp}_j,\sigma_d,false)$ | $(P\text{-}XorSplit_2)$ |
| $\mathsf{andJoin}(E,\mathsf{e}) \xrightarrow{\epsilon} \langle inc(dec(\sigma_e,E),\mathsf{e})\rangle$ | $\forall\mathsf{e}'\in E\ .\ \sigma_e(\mathsf{e}') > 0$ | $(P\text{-}AndJoin)$ |
| $\mathsf{xorJoin}(\{\mathsf{e}\}\cup E,\mathsf{e}') \xrightarrow{\epsilon} \langle inc(dec(\sigma_e,\mathsf{e}),\mathsf{e}')\rangle$ | $\sigma_e(\mathsf{e}) > 0$ | $(P\text{-}XorJoin)$ |
| $\mathsf{eventBased}(\mathsf{e},(\mathsf{m}_1:\tilde{\mathsf{t}}_1,\mathsf{e}_1),\ldots,(\mathsf{m}_h:\tilde{\mathsf{t}}_h,\mathsf{e}_h)) \xrightarrow{?\mathsf{m}_j:\tilde{\mathsf{et}}_j,\epsilon} \langle inc(dec(\sigma_e,\mathsf{e}),\mathsf{e}_j)\rangle$ | $\sigma_e(\mathsf{e}) > 0,\ 1\leqslant j\leqslant h,$ $eval(\tilde{\mathsf{t}}_j,\sigma_d,\tilde{\mathsf{et}}_j)$ | $(P\text{-}EventG)$ |

**Fig. 7.** BPMN process semantics: events and gateways.

| | | |
|---|---|---|
| $\mathsf{task}(\mathsf{e},\mathsf{n},\mathsf{a},\mathsf{exp},A,\mathsf{e}') \xrightarrow{\epsilon} \langle inc(dec(\sigma_e,\mathsf{e}),\mathsf{e}'),\sigma'_d\rangle$ | $\sigma_e(\mathsf{e}) > 0,$ $eval(\mathsf{exp},\sigma_d,true),$ $upd(\sigma_d,A,\sigma'_d)$ | $(P\text{-}Task_A)$ |
| $\mathsf{taskRcv}(\mathsf{e},\mathsf{n},\mathsf{a},\mathsf{exp},A,\mathsf{m}:\tilde{\mathsf{t}},\mathsf{e}') \xrightarrow{?\mathsf{m}:\tilde{\mathsf{et}},A} \langle inc(dec(\sigma_e,\mathsf{e}),\mathsf{e}')\rangle$ | $\sigma_e(\mathsf{e}) > 0,$ $eval(\mathsf{exp},\sigma_d,true),$ $eval(\tilde{\mathsf{t}},\sigma_d,\tilde{\mathsf{et}})$ | $(P\text{-}TaskRcv_A)$ |
| $\mathsf{taskSnd}(\mathsf{e},\mathsf{n},\mathsf{a},\mathsf{exp},A,\mathsf{m}:\tilde{\mathsf{exp}},\mathsf{e}') \xrightarrow{!\mathsf{m}:\tilde{\mathsf{v}}} \langle inc(dec(\sigma_e,\mathsf{e}),\mathsf{e}'),\sigma'_d\rangle$ | $\sigma_e(\mathsf{e}) > 0,$ $eval(\mathsf{exp}',\sigma_d,true),$ $upd(\sigma_d,A,\sigma'_d),$ $eval(\tilde{\mathsf{exp}},\sigma_d,\tilde{\mathsf{v}})$ | $(P\text{-}TaskSnd_A)$ |

**Fig. 8.** BPMN process semantics: tasks with atomic execution.

and consumption of the token in the incoming edge, and task completion (rule *P-Task*$_{N2}$), dealing with the execution of the assignments and the insertion of the token in the outgoing edge. Notably, in case of non-concurrent execution, the task activation is performed only if the task is in the inactive status (i.e., there are no active instances). Similarly, the execution of receiving/sending

Please cite this article as: F. Corradini, C. Muzi, B. Re et al., Formalising and animating multiple instances in BPMN collaborations, Information Systems (2019) 101459, https://doi.org/10.1016/j.is.2019.101459.

$$\mathsf{task}(\mathsf{e},\mathsf{n},N,\mathsf{exp},A,\mathsf{e}') \xrightarrow{\epsilon} \langle dec(\sigma_e,\mathsf{e}), inc(\sigma_t,\mathsf{n},\mathsf{a})\rangle \quad \sigma_e(\mathsf{e})>0, eval(\mathsf{exp},\sigma_d,true), N=\mathsf{na\_nc} \Rightarrow isInactive(\sigma_t,\mathsf{n}) \quad (P\text{-}Task_{N1})$$

$$\mathsf{task}(\mathsf{e},\mathsf{n},N,\mathsf{exp},A,\mathsf{e}') \xrightarrow{\epsilon} \langle inc(\sigma_e,\mathsf{e}'),\sigma'_d,dec(\sigma_t,\mathsf{n},\mathsf{a})\rangle \quad \sigma_t(\mathsf{n},\mathsf{a})>0, upd(\sigma_d,A,\sigma'_d) \quad (P\text{-}Task_{N2})$$

$$\mathsf{taskRcv}(\mathsf{e},\mathsf{n},N,\mathsf{exp},A,\mathsf{m}:\tilde{\mathsf{t}},\mathsf{e}') \xrightarrow{\epsilon} \langle dec(\sigma_e,\mathsf{e}), inc(\sigma_t,\mathsf{n},\mathsf{a})\rangle \quad \sigma_e(\mathsf{e})>0, eval(\mathsf{exp},\sigma_d,true), N=\mathsf{na\_nc} \Rightarrow isInactive(\sigma_t,\mathsf{n}) \quad (P\text{-}TaskRcv_{N1})$$

$$\mathsf{taskRcv}(\mathsf{e},\mathsf{n},N,\mathsf{exp},A,\mathsf{m}:\tilde{\mathsf{t}},\mathsf{e}') \xrightarrow{?\mathsf{m}:\tilde{\mathsf{et}},\epsilon} \langle inc(dec(\sigma_t,\mathsf{n},\mathsf{a}),\mathsf{n},\mathsf{r})\rangle \quad \sigma_t(\mathsf{n},\mathsf{a})>0, eval(\tilde{\mathsf{t}},\sigma_d,\tilde{\mathsf{et}}) \quad (P\text{-}TaskRcv_{N2})$$

$$\mathsf{taskRcv}(\mathsf{e},\mathsf{n},N,\mathsf{exp},A,\mathsf{m}:\tilde{\mathsf{t}},\mathsf{e}') \xrightarrow{\epsilon} \langle inc(\sigma_e,\mathsf{e}'),\sigma'_d,dec(\sigma_t,\mathsf{n},\mathsf{r})\rangle \quad \sigma_t(\mathsf{n},\mathsf{r})>0, upd(\sigma_d,A,\sigma'_d) \quad (P\text{-}TaskRcv_{N3})$$

$$\mathsf{taskSnd}(\mathsf{e},\mathsf{n},N,\mathsf{exp},A,\mathsf{m}:\tilde{\mathsf{exp}},\mathsf{e}') \xrightarrow{\epsilon} \langle dec(\sigma_e,\mathsf{e}), inc(\sigma_t,\mathsf{n},\mathsf{a})\rangle \quad \sigma_e(\mathsf{e})>0, eval(\mathsf{exp},\sigma_d,true), N=\mathsf{na\_nc} \Rightarrow isInactive(\sigma_t,\mathsf{n}) \quad (P\text{-}TaskSnd_{N1})$$

$$\mathsf{taskSnd}(\mathsf{e},\mathsf{n},N,\mathsf{exp},A,\mathsf{m}:\tilde{\mathsf{exp}},\mathsf{e}') \xrightarrow{\epsilon} \langle \sigma'_d, inc(dec(\sigma_t,\mathsf{n},\mathsf{a}),\mathsf{n},\mathsf{s})\rangle \quad \sigma_t(\mathsf{n},\mathsf{a})>0, upd(\sigma_d,A,\sigma'_d) \quad (P\text{-}TaskSnd_{N2})$$

$$\mathsf{taskSnd}(\mathsf{e},\mathsf{n},N,\mathsf{exp},A,\mathsf{m}:\tilde{\mathsf{exp}},\mathsf{e}') \xrightarrow{!\mathsf{m}:\tilde{\mathsf{v}}} \langle inc(\sigma_e,\mathsf{e}'), dec(\sigma_t,\mathsf{n},\mathsf{s})\rangle \quad \sigma_t(\mathsf{n},\mathsf{s})>0, eval(\tilde{\mathsf{exp}},\sigma_d,\tilde{\mathsf{v}}) \quad (P\text{-}TaskSnd_{N3})$$

**Fig. 9.** BPMN process semantics: tasks with non-atomic execution.

tasks is split in three steps: task activation, receiving/sending of the message while the task is running, and task completion. Again, non-concurrent tasks are activated only if they are in the inactive status.

Rules in Fig. 10 deal with multi-instance tasks, both in parallel and in sequence. A parallel multi-instance task is activated when it is inactive (i.e., $\sigma_c(\mathsf{c}) = 0$) and has an incoming token. If the *loop cardinality* expression exp is evaluated to a natural number $h$ greater than 0 (rule *P-MipTask*$_1$), this value is assigned to the task counter c, and $h$ tokens are inserted in the incoming edge of the wrapped task $T$. Instead, if the loop cardinality is 0 (rule *P-MipTask*$_2$), no execution of $T$ is performed and the incoming token is moved directly to the outgoing edge. When the multi-instance task is active, the wrapped task can be executed according to the task rules previously described (rule *P-MipTask*$_3$). Finally, the multi-instance task completes (rule *P-MipTask*$_4$) when either all task instances have completed their execution (i.e., the number of tokens in the outgoing edge of $T$ is equal to the loop cardinality stored in the counter c) or the *completion condition* expression exp′ is evaluated to *true*. Rules for sequential multi-instance task are similar, thus we just discuss the key differences. When the multi-instance task is activated, only one token is inserted in the incoming edge of the wrapped task (rule *P-MisTask*$_1$). Then, when the wrapped task produces a token in its outgoing edge, indicating its termination, if the multi-instance task has not completed its execution then the wrapped task is reactivated and the counter decreased (rule *P-MisTask*$_4$).

The last group of rules, *P-Int*$_1$ and *P-Int*$_2$ in Fig. 11, deal with interleaving of process elements in a standard way, so that if an element of a process evolves then the whole process evolves accordingly.

Now, let us consider the semantics at collaboration level. A *collaboration configuration* has the form $\langle C, \sigma_i, \sigma_m, \sigma_{ds}\rangle$, where:

- $C$ is a collaboration structure;
- $\sigma_i : \mathbb{P} \to 2^{\mathbb{S}_{\sigma_e} \times \mathbb{S}_{\sigma_{do}} \times \mathbb{S}_{\sigma_{dc}} \times \mathbb{S}_{\sigma_t} \times \mathbb{S}_{\sigma_c}}$ is an *instance state function* mapping each pool name to a multiset of instance states, ranged over by $I$ and containing quintuples of the form $\langle \sigma_e, \sigma_{do}, \sigma_{dc}, \sigma_t, \sigma_c\rangle$ (where $\mathbb{S}_\sigma$ is the set of states of kind $\sigma$);
- $\sigma_m : \mathbb{M} \to 2^{\mathbb{V}^n}$ is a *message state function* that assigns to each message name m a multiset of value tuples representing the messages received along the message edge labelled by m;
- $\sigma_{ds}$ is a *data store state function*, defined as for process configurations.

Notice that our semantics has been defined according to a global perspective. Indeed, the overall state of a collaboration is collected by functions $\sigma_i$, $\sigma_m$ and $\sigma_{ds}$ of its configuration. On the other hand, the global semantics of a collaboration configuration is determined, in a compositional way, by the local semantics of the involved processes, which evolve independently from each other. The use of a global perspective simplifies *(i)* the technicalities required by the formal definition of the semantics, and *(ii)* the implementation of the animation of the overall collaboration execution. The compositional definition of the semantics, anyway, would allow to easily pass to a purely local perspective, where state functions are kept separate for each process.

To simplify the definition of the operational rules, we introduce in Fig. 12 some auxiliary functions to update states of collaboration configurations. To define the collaboration semantics, an auxiliary function is needed: $match(\tilde{\mathsf{et}}, \tilde{\mathsf{v}})$ is a partial function performing *pattern-matching* on structured data (like in [16]), thus determining if an evaluated template $\tilde{\mathsf{et}}$ matches a tuple of values $\tilde{\mathsf{v}}$. A successful matching returns a list of assignments $A$, updating the formal fields in the template; otherwise, the function is undefined. The formal definition of the pattern-matching function is reported in Appendix B.

Let us go back to our running example. The scenario in its initial state is rendered as the collaboration configuration

$$\langle(\mathsf{pool}(\mathsf{p}_c, P_c) \parallel \mathsf{pool}(\mathsf{p}_p, P_p) \parallel \mathsf{miPool}(\mathsf{p}_a, P_a, 3)), \sigma_i, \sigma_m, \sigma_{ds}\rangle$$

where: $\sigma_i(\mathsf{p}_c)=\{\langle \sigma_e, \sigma_{do}, \sigma_{dc}, \sigma_t, \sigma_c\rangle\}$ with $\sigma_e=\sigma_e^0 \cdot [\mathsf{e}_1 \mapsto 1]$ and $\sigma_{do}=\sigma_{do}^0 \cdot$[LayersInfo.top, LayersInfo.middle, LayersInfo.bottom $\mapsto$ *Blue*, *Pink*, *Brown*]; and $\sigma_i(\mathsf{p}_p) = \sigma_i(\mathsf{p}_a) = \varnothing$. Notice that the $\sigma_{do}$ function of the $\mathsf{p}_c$ instance is initialised with the content of the LayersInfo data input.

The labelled transition relation on collaboration configurations formalises the message exchange and the data update according to the process evolution. The LTS is a triple $\langle \mathcal{C}, \mathcal{L}_c, \to_c\rangle$ where: $\mathcal{C}$ is a set of collaboration configurations; $\mathcal{L}_c$, ranged over by $l$, is a set of labels; and $\to_c \subseteq \mathcal{C} \times \mathcal{L}_c \times \mathcal{C}$ is a transition relation. We apply the same readability simplifications we use for process

Please cite this article as: F. Corradini, C. Muzi, B. Re et al., Formalising and animating multiple instances in BPMN collaborations, Information Systems (2019) 101459, https://doi.org/10.1016/j.is.2019.101459.

$$\mathsf{mipTask}(\mathsf{e},\mathsf{exp},T,\mathsf{c},\mathsf{exp}',\mathsf{e}') \xrightarrow{\epsilon} \langle set(dec(\sigma_e,\mathsf{e}),in(T),h),set(\sigma_c,\mathsf{c},h)\rangle \quad \sigma_e(\mathsf{e})>0,\ \sigma_c(\mathsf{c})=0,\ eval(\mathsf{exp},\sigma_d,h)\ with\ h>0 \quad (P\text{-}MipTask_1)$$

$$\mathsf{mipTask}(\mathsf{e},\mathsf{exp},T,\mathsf{c},\mathsf{exp}',\mathsf{e}') \xrightarrow{\epsilon} \langle inc(dec(\sigma_e,\mathsf{e}),\mathsf{e}')\rangle \quad \sigma_e(\mathsf{e})>0,\ \sigma_c(\mathsf{c})=0,\ eval(\mathsf{exp},\sigma_d,0) \quad (P\text{-}MipTask_2)$$

$$\frac{\langle T,\sigma_e,\sigma_d,\sigma_t,\sigma_c\rangle \xrightarrow{\ell} \langle \sigma'_e,\sigma'_d,\sigma'_t,\sigma'_c\rangle}{\mathsf{mipTask}(\mathsf{e},\mathsf{exp},T,\mathsf{c},\mathsf{exp}',\mathsf{e}') \xrightarrow{\ell} \langle \sigma'_e,\ \sigma'_d,\sigma'_t,\sigma'_c\rangle} \quad (P\text{-}MipTask_3)$$

$$\mathsf{mipTask}(\mathsf{e},\mathsf{exp},T,\mathsf{c},\mathsf{exp}',\mathsf{e}') \xrightarrow{\epsilon} \langle inc(reset(\sigma_e,edges(T)),\mathsf{e}'),reset(\sigma_c,\mathsf{c})\rangle \quad \sigma_e(out(T))=\sigma_c(\mathsf{c}) \vee eval(\mathsf{exp}',\sigma_d,true) \quad (P\text{-}MipTask_4)$$

$$\mathsf{misTask}(\mathsf{e},\mathsf{exp},T,\mathsf{c},\mathsf{exp}',\mathsf{e}') \xrightarrow{\epsilon} \langle inc(dec(\sigma_e,\mathsf{e}),in(T)),set(\sigma_c,\mathsf{c},h)\rangle \quad \sigma_e(\mathsf{e})>0,\ \sigma_c(\mathsf{c})=0,\ eval(\mathsf{exp},\sigma_d,h)\ with\ h>0 \quad (P\text{-}MisTask_1)$$

$$\mathsf{misTask}(\mathsf{e},\mathsf{exp},T,\mathsf{c},\mathsf{exp}',\mathsf{e}'),\xrightarrow{\epsilon} \langle inc(dec(\sigma_e,\mathsf{e}),\mathsf{e}')\rangle \quad \sigma_e(\mathsf{e})>0,\ \sigma_c(\mathsf{c})=0,\ eval(\mathsf{exp},\sigma_d,0) \quad (P\text{-}MisTask_2)$$

$$\frac{\langle T,\sigma_e,\sigma_d,\sigma_t,\sigma_c\rangle \xrightarrow{\ell} \langle \sigma'_e,\ \sigma'_d,\sigma'_t,\sigma'_c\rangle}{\mathsf{misTask}(\mathsf{e},\mathsf{exp},T,\mathsf{c},\mathsf{exp}',\mathsf{e}') \xrightarrow{\ell} \langle \sigma'_e,\ \sigma'_d,\sigma'_t,\sigma'_c\rangle} \quad (P\text{-}MisTask_3)$$

$$\mathsf{misTask}(\mathsf{e},\mathsf{exp},T,\mathsf{c},\mathsf{exp}',\mathsf{e}') \xrightarrow{\epsilon} \langle inc(dec(\sigma_e,out(T)),in(T)),dec(\sigma_c,\mathsf{c})\rangle \quad \sigma_c(\mathsf{c})>1,\ \sigma_e(out(T))=1,\ eval(\mathsf{exp}',\sigma_d,false) \quad (P\text{-}MisTask_4)$$

$$\mathsf{misTask}(\mathsf{e},\mathsf{exp},T,\mathsf{c},\mathsf{exp}',\mathsf{e}') \xrightarrow{\epsilon} \langle inc(dec(\sigma_e,out(T)),\mathsf{e}'),reset(\sigma_c,\mathsf{c})\rangle \quad \sigma_e(out(T))=1,\ (\sigma_c(\mathsf{c})=1 \vee eval(\mathsf{exp}',\sigma_d,true)) \quad (P\text{-}MisTask_5)$$

**Fig. 10.** BPMN process semantics: parallel/sequential multi-instance tasks.

$$\frac{\langle P_1,\sigma_e,\sigma_d,\sigma_t,\sigma_c\rangle \xrightarrow{\ell} \langle \sigma'_e,\sigma'_d,\sigma'_t,\sigma'_c\rangle}{P_1 \parallel P_2 \xrightarrow{\ell} \langle \sigma'_e,\sigma'_d,\sigma'_t,\sigma'_c\rangle}\ (P\text{-}Int_1) \qquad \frac{\langle P_2,\sigma_e,\sigma_d,\sigma_t,\sigma_c\rangle \xrightarrow{\ell} \langle \sigma'_e,\sigma'_d,\sigma'_t,\sigma'_c\rangle}{P_1 \parallel P_2 \xrightarrow{\ell} \langle \sigma'_e,\sigma'_d,\sigma'_t,\sigma'_c\rangle}\ (P\text{-}Int_2)$$

**Fig. 11.** BPMN process semantics: interleaving.

| | |
|---|---|
| $addInst(\sigma_i,\mathsf{p},\sigma_e,\sigma_{do},\sigma_{dc},\sigma_t,\sigma_c) = \sigma_i \cdot [\mathsf{p} \mapsto \sigma_i(\mathsf{p}) + \{\langle \sigma_e,\sigma_{do},\sigma_{dc},\sigma_t,\sigma_c\rangle\}]$ | adds the new instance $\langle \sigma_e,\sigma_{do},\sigma_{dc},\sigma_t,\sigma_c\rangle$ to the multiset of instances of pool p in the state $\sigma_i$ |
| $updInst(\sigma_i,\mathsf{p},\sigma_e,\sigma_{do},\sigma_{dc},\sigma_t,\sigma_c,I) = \sigma_i \cdot [\mathsf{p} \mapsto \{\langle \sigma_e,\sigma_{do},\sigma_{dc},\sigma_t,\sigma_c\rangle\} + I]$ | replaces an existing instance of p in the state $\sigma_i$, leaving instances $I$ unaltered |
| $add(\sigma_m,\mathsf{m},\tilde{\mathsf{v}}) = \sigma_m \cdot [\mathsf{m} \mapsto \sigma_m(\mathsf{m}) + \{\tilde{\mathsf{v}}\}]$ | adds the value tuple $\tilde{\mathsf{v}}$ for the message name m in the state $\sigma_m$ |
| $rm(\sigma_m,\mathsf{m},\tilde{\mathsf{v}}) = \sigma_m \cdot [\mathsf{m} \mapsto \sigma_m(\mathsf{m}) - \{\tilde{\mathsf{v}}\}]$ | removes the message with value tuple $\tilde{\mathsf{v}}$ for the message name m from the state $\sigma_m$ |

**Fig. 12.** BPMN collaboration semantics: auxiliary functions for updating states (where operators $+$ and $-$ are the union and subtraction operations on multisets).

Please cite this article as: F. Corradini, C. Muzi, B. Re et al., Formalising and animating multiple instances in BPMN collaborations, Information Systems (2019) 101459, https://doi.org/10.1016/j.is.2019.101459.

$$\frac{\sigma_i(\mathsf{p}) = \varnothing \qquad \langle P, \sigma_e^0, \sigma_d^0, \sigma_t^0, \sigma_c^0\rangle \xrightarrow{new\ \mathsf{m}:\tilde{\mathsf{et}}} \langle \sigma'_e, \sigma'_d, \sigma'_t, \sigma'_c\rangle \quad \tilde{\mathsf{v}} \in \sigma_m(\mathsf{m}) \qquad match(\tilde{\mathsf{et}}, \tilde{\mathsf{v}}) = A \quad upd(\sigma'_d, A, \sigma''_{do}, \sigma''_{dc}, \sigma''_{ds})}{\mathsf{pool}(\mathsf{p}, P) \xrightarrow{new\ \mathsf{m}:\tilde{\mathsf{v}}} \langle addInst(\sigma_i, \mathsf{p}, \sigma'_e, \sigma''_{do}, \sigma''_{dc}, \sigma'_t, \sigma'_c), rm(\sigma_m, \mathsf{m}, \tilde{\mathsf{v}}), \sigma''_{ds}\rangle} \ (C\text{-}Create)$$

$$\frac{|\sigma_i(\mathsf{p})| < \mathsf{max} \qquad \langle P, \sigma_e^0, \sigma_d^0, \sigma_t^0, \sigma_c^0\rangle \xrightarrow{new\ \mathsf{m}:\tilde{\mathsf{et}}} \langle \sigma'_e, \sigma'_d, \sigma'_t, \sigma'_c\rangle \quad \tilde{\mathsf{v}} \in \sigma_m(\mathsf{m}) \qquad match(\tilde{\mathsf{et}}, \tilde{\mathsf{v}}) = A \quad upd(\sigma'_d, A, \sigma''_{do}, \sigma''_{dc}, \sigma''_{ds})}{\mathsf{miPool}(\mathsf{p}, P, \mathsf{max}) \xrightarrow{new\ \mathsf{m}:\tilde{\mathsf{v}}} \langle addInst(\sigma_i, \mathsf{p}, \sigma'_e, \sigma''_{do}, \sigma''_{dc}, \sigma'_t, \sigma'_c), rm(\sigma_m, \mathsf{m}, \tilde{\mathsf{v}}), \sigma''_{ds}\rangle} \ (C\text{-}CreateMi)$$

$$\frac{\sigma_i(\mathsf{p}) = \{\langle \sigma_e, \sigma_{do}, \sigma_{dc}, \sigma_t, \sigma_c\rangle\} + I \qquad \langle P, \sigma_e, \sigma_{do}, \sigma_{dc}, \sigma_{ds}, \sigma_t, \sigma_c\rangle \xrightarrow{\tau} \langle \sigma'_e, \sigma'_{do}, \sigma'_{dc}, \sigma'_{ds}, \sigma'_t, \sigma'_c\rangle}{\mathsf{miPool}(\mathsf{p}, P, \mathsf{max}) \xrightarrow{\tau} \langle updInst(\sigma_i, \mathsf{p}, \sigma'_e, \sigma'_{do}, \sigma'_{dc}, \sigma'_t, \sigma'_c, I), \sigma'_{ds}\rangle} \ (C\text{-}InternalMi)$$

$$\frac{\sigma_i(\mathsf{p}) = \{\langle \sigma_e, \sigma_{do}, \sigma_{dc}, \sigma_t, \sigma_c\rangle\} + I \qquad \langle P, \sigma_e, \sigma_{do}, \sigma_{dc}, \sigma_{ds}, \sigma_t, \sigma_c\rangle \xrightarrow{?\mathsf{m}:\tilde{\mathsf{et}}, A} \langle \sigma'_e, \sigma'_d, \sigma'_t, \sigma'_c\rangle \quad \tilde{\mathsf{v}} \in \sigma_m(\mathsf{m}) \qquad match(\tilde{\mathsf{et}}, \tilde{\mathsf{v}}) = A' \qquad upd(\sigma'_d, (A', A), \sigma''_{do}, \sigma''_{dc}, \sigma''_{ds})}{\mathsf{miPool}(\mathsf{p}, P, \mathsf{max}) \xrightarrow{?\mathsf{m}:\tilde{\mathsf{v}}} \langle updInst(\sigma_i, \mathsf{p}, \{\langle \sigma'_e, \sigma''_{do}, \sigma''_{dc}, \sigma'_t, \sigma'_c\rangle\} + I), rm(\sigma_m, \mathsf{m}, \tilde{\mathsf{v}}), \sigma''_{ds}\rangle} \ (C\text{-}ReceiveMi)$$

$$\frac{\sigma_i(\mathsf{p}) = \{\langle \sigma_e, \sigma_{do}, \sigma_{dc}, \sigma_t, \sigma_c\rangle\} + I \qquad \langle P, \sigma_e, \sigma_{do}, \sigma_{dc}, \sigma_{ds}, \sigma_t, \sigma_c\rangle \xrightarrow{!\mathsf{m}:\tilde{\mathsf{v}}} \langle \sigma'_e, \sigma'_{do}, \sigma'_{dc}, \sigma'_{ds}, \sigma'_t, \sigma'_c\rangle}{\mathsf{miPool}(\mathsf{p}, P, \mathsf{max}) \xrightarrow{!\mathsf{m}:\tilde{\mathsf{v}}} \langle updInst(\sigma_i, \mathsf{p}, \{\langle \sigma'_e, \sigma'_{do}, \sigma'_{dc}, \sigma'_t, \sigma'_c\rangle\} + I), add(\sigma_m, \mathsf{m}, \tilde{\mathsf{v}}), \sigma'_{ds}\rangle} \ (C\text{-}DeliverMi)$$

$$\frac{\langle C_1, \sigma_i, \sigma_m, \sigma_{ds}\rangle \xrightarrow{l} \langle \sigma'_i, \sigma'_m, \sigma'_{ds}\rangle}{C_1 \parallel C_2 \xrightarrow{l} \langle \sigma'_i, \sigma'_m, \sigma'_{ds}\rangle} \ (C\text{-}Int_1) \qquad \frac{\langle C_2, \sigma_i, \sigma_m, \sigma_{ds}\rangle \xrightarrow{l} \langle \sigma'_i, \sigma'_m, \sigma'_{ds}\rangle}{C_1 \parallel C_2 \xrightarrow{l} \langle \sigma'_i, \sigma'_m, \sigma'_{ds}\rangle} \ (C\text{-}Int_2)$$

**Fig. 13.** BPMN collaboration semantics.

configuration transitions. The labels used by the collaboration transition relation are generated by the following grammar:

$$l ::= \tau \quad | \quad !\mathsf{m}:\tilde{\mathsf{v}} \quad | \quad ?\mathsf{m}:\tilde{\mathsf{v}} \quad | \quad new\,\mathsf{m}:\tilde{\mathsf{v}}$$

Notably, internal and sending labels coincides with the same labels at the process level, while the receiving labels here just keep track of the received message.

The operational rules defining the transition relation of the collaboration semantics are given in Fig. 13; except for the rules used to create new process instances, we only report the rules for multi-instance pools, as the single-instance ones are similar (omitted rules are reported in Appendix B). We now briefly comment on the relevant points. The first two rules deal with instance creation. In the single instance case (rule *C-Create*), an instance is created only if no instance exists for the considered pool, and there is a matching message. As result, the assignments for the received data are performed, and the message is consumed. In the multi-instance case (rule *C-CreateMi*), the created instance is simply added to the multiset of existing instances of the pool. Anyway, the instance is created only if the maximum number of allowed instances is not exceeded. The next three rules allow a single pool, representing organisation p, to evolve according to the evolution of one of its process instances. In particular, if the process instance performs an internal action (rule *C-InternalMi*) or a receiving/delivery action (rules *C-ReceiveMi* or *C-DeliverMi*), the pool performs the corresponding action at collaboration layer. As for instance creation, rule *C-ReceiveMi* can be applied only if there is at least one matching message. Recall indeed that at process level the receiving labels just indicate the willingness of a process instance to consume a received message, regardless the actual presence of messages. The delivering of messages is based on the *correlation* mechanism: the correlation data are identified by the template fields that are not formal (i.e., those fields requiring specific matching values). Moreover, when a process performs a sending action, the message state function is updated in order to deliver the sent message to the receiving participant. Finally, rules *C-Int*$_1$ and *C-Int*$_2$ permit to interleave the execution of actions performed by pools of the same collaboration, so that if a part of a larger collaboration evolves, the whole collaboration evolves accordingly.

## 4. Formalisation at work on multi-instance interaction scenarios

In this section we show the capability of our formal approach to model multi-instance collaborations. In this kind of scenarios the use of a formalisation is particularly helpful. Indeed, the overall behaviour resulting from the interactions among multiple instances of different participants, and driven by local and shared data, is usually complex and convoluted.

The considered scenarios are introduced as variants of our running example, where different communication-related aspects are added with an increasing level of complexity. In particular, we consider: *(i)* communication between different multi-instance pools, where the initiator of the collaboration is single-instance; *(ii)* communication between multi-instance pools, where the initiator of the collaboration is multi-instance; and *(iii)* communication between instances of the same pool. For the sake of readability, we resort here to the graphical representation of the considered BPMN model examples. The complete specification of these models in our textual representation is available in

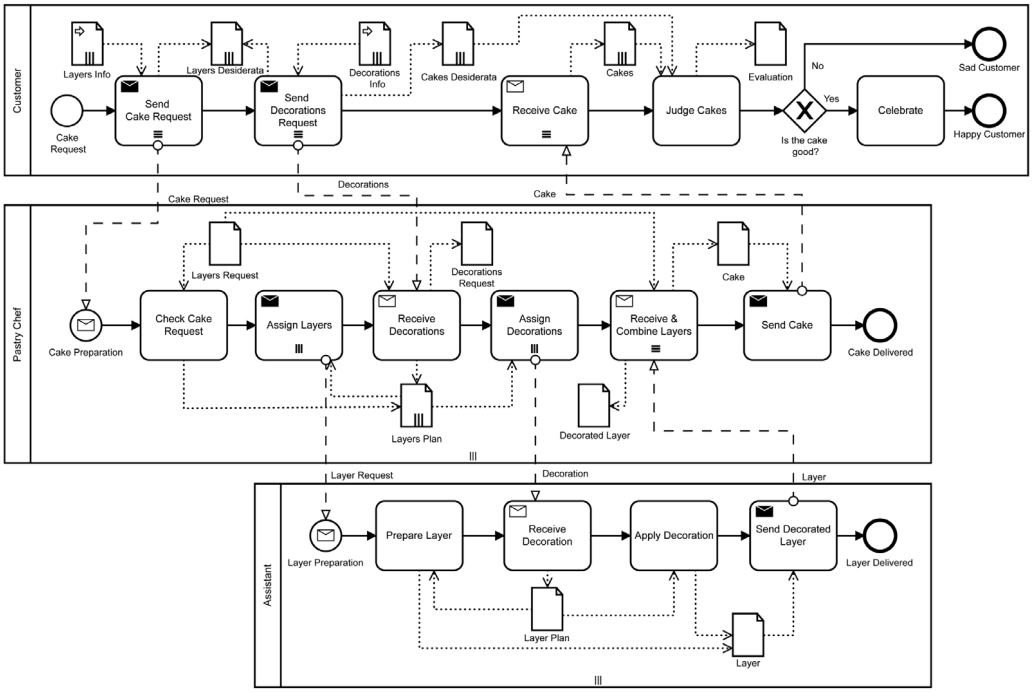



**Fig. 14.** Collaboration with two multi-instance pools.

Appendix C, while their *.bpmn* standard format definitions are available online.[2]

Let us first consider the scenario in Fig. 14, where the *Customer* pool initiates the collaboration, and the *Pastry Chef* and the *Assistant* multi-instance pools interact together. The collaboration model revises the running example in order to allow the *Customer* to request two different three-layer cakes. To this aim, we have modified the *Customer* by making *Send Cake Request*, as well the subsequent tasks, sequential multi-instance tasks with loop cardinality set to two according to the number of cakes to be produced. The data elements in the *Customer* pool have been revised accordingly, in order to provide information on the two cakes. Consequently, the *Pastry Chef* is now a multi-instance pool, with maximum number of instances set to two (we assume that only two pastry chefs are available on the pastry shop). Finally, we have added a new correlation field in the template of the *Receive & Combine Layers* task, so that each *Pastry Chef* instance is able to identify its own cake layer among those received from the *Assistant* pool. The revised example is presented in our syntax in Appendix C - Figs. C.26 and C.27. During the model execution, according to the proposed semantics (in particular, rule *P-StartRcv* at process level and rule *C-CreateMi* at collaboration level), the *Customer* triggers the creation of two *Pastry Chef* instances, which in their own turn trigger, by means of the same semantic rules, the creation of an *Assistant* instance for each cake layer to be produced (hence, we will have six *Assistant* instances in total). The template of the *Receive & Combine Layers* task is $\langle$? DecoratedLayer.layer, ? DecoratedLayer.position, LayersRequest.cakeID$\rangle$, where the additional correlation data LayersRequest.cakeID, which is a cake identifier, guarantees correct delivery and combination of decorated layers. Indeed, by rule *P-TaskRcv*$_A$, the template is evaluated with respect to the current data state $\sigma_d$ as $\langle$? DecoratedLayer.layer, ? DecoratedLayer.position$\rangle$, $v_{id}$, where $\sigma_d$(LayersRequest.cakeID) $= v_{id}$. The correlation between this template and the received message is formally expressed by the *match* function in rule *C-ReceiveMi*, which produces an assignment, instantiating the formal fields ? DecoratedLayer.layer and ? DecoratedLayer.position, used to update the state $\sigma_d$ by means of the *upd* relation. In this way, only the *Pastry Chef* instance working on the cake identified by $v_{id}$ can receive the *Layer* message containing the $v_{id}$ correlation data. Thus, the *Customer* will receive the two cakes properly combined, even if they have been produced concurrently by multiple chef and assistant instances.

Let us extend the scenario in Fig. 14 by making all pools multi-instance. We obtain in this way the collaboration in Fig. 15 that is reported in our syntax in Appendix C - Fig. C.28. Now, we have that the minimum and maximum numbers of instances of the *Customer* pool are set to two, in order to consider two separate customers in the collaboration. Notably, the maximum number of instances is rendered in our textual notation as a parameter of the miPool term (i.e., miPool($p_c, P_c, 2$)); instead, the minimum number is used to initiate the instance state $\sigma_i$ for the *Customer* pool with two initial instances (i.e., $\sigma_i(p_c) = \{\langle \sigma'_e, \sigma'_{do}, \sigma^0_{dc}, \sigma^0_t, \sigma^0_c \rangle, \langle \sigma''_e, \sigma''_{do}, \sigma^0_{dc}, \sigma^0_t, \sigma^0_c \rangle\}$). The different cakes are described in the data objects *Layers Info* and *Decorations Info*, which are differently instantiated in the two *Customer* instances (i.e., $\sigma'_{do}$ and $\sigma''_{do}$ associate different values to the two data objects). The collaboration starts with the execution of the two instances of *Customer*. Notably, we can observe here how our formalisation permits the instantiation of different instances of the same pool using data inputs. More specifically, at the outset, $\sigma_i(p_c)$ returns two instances where the edge state functions correspond to $\sigma'_e = \sigma''_e = \sigma^0_e \cdot [e_1 \mapsto 1]$, and the data collection state functions include input data as following: $\sigma'_{do} = \sigma^0_{do} \cdot$ [LayersInfo.top $\mapsto$ Blue, LayersInfo.middle $\mapsto$ Pink, LayersInfo.low $\mapsto$ Brown, DecorationsInfo.top $\mapsto$ Blue, DecorationsInfo.middle $\mapsto$ Pink, DecorationsInfo.low $\mapsto$ Brown], and $\sigma''_{do}$ is defined similarly. During the model execution, according to rules *P-StartRcv* and *C-CreateMi*, each *Customer* instance triggers the creation of a *Pastry Chef* instance by filling

[2] The *.bpmn* files of the different variants of the BPMN models of the cake example introduced in this paper are available at: https://bitbucket.org/proslabteam/mida/src/master/assets/examples/Cake/.

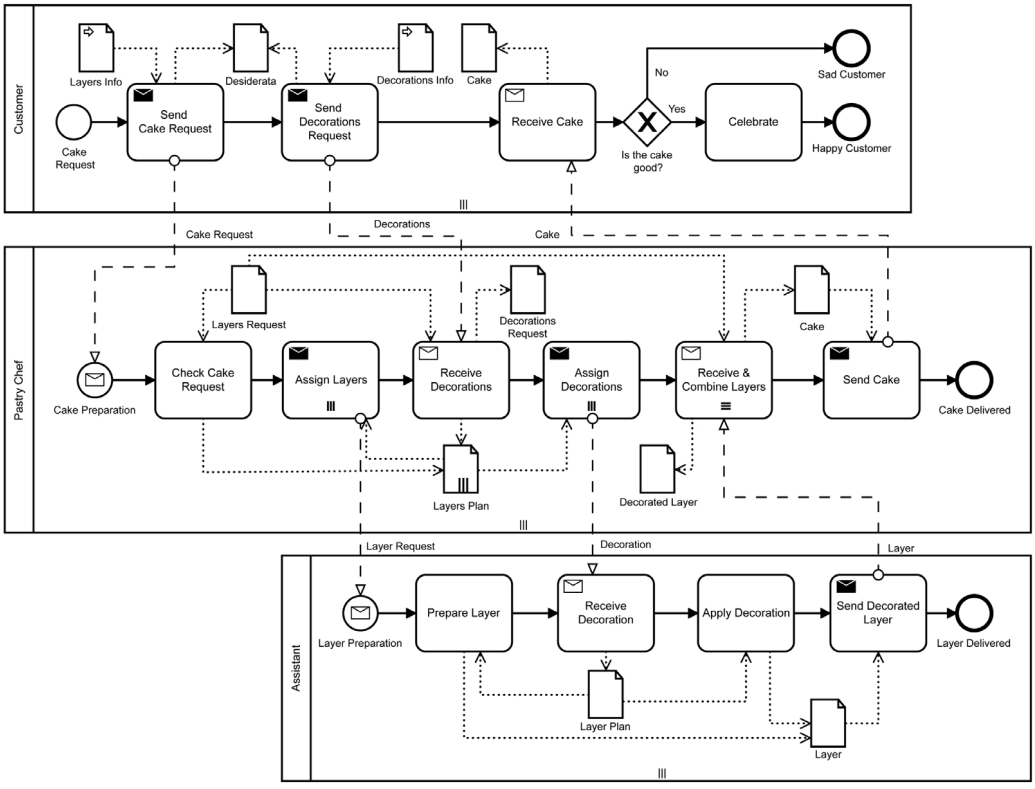



**Fig. 15.** Collaboration with three multi-instance pools.

the *CakeRequest* message with data from its data object state (i.e., $\sigma'_{do}$ or $\sigma''_{do}$). Then, as before, each *Pastry Chef* instance is able to coordinate the activity of a specific cake with three different instances of *Assistant*. The correlation data allows *(i)* each *Pastry Chef* to assign each type of decorations to the right *Assistant*; *(ii)* each *Assistant* to send back the decorated layer to the correct *Pastry Chef*; and *(iii)* each *Pastry Chef* to deliver the cake to the right *Customer*. This is supported again by the specification of correlation data and the use of the *match* function in rule *C-ReceiveMi*.

Finally, let us extend again the scenario by considering a situation where two customers, requesting different cakes, have to organise a party together and, hence, will celebrate only if both of them positively judge their cakes. This requires the two *Customer* instances to interact each other to coordinate on the decision about the party celebration. The resulting collaboration is depicted in Fig. 16 and presented in our syntax in Appendix C - Fig. C.29. To enable the interaction among instances of the same pool we resort to a data store, which acts as a shared memory allowing the instances to indirectly communicate. We recall indeed that in our formalisation the data store state function $\sigma_{ds}$ is not included in the state of process instances, but it is defined at collaboration level. Specifically, in our example we have added the data store *Judgements*, where each *Customer* instance inserts, via task *Judge Cake*, a judgement concerning the received cake. This is made possible by means of the *upd* function, executed by rule *P-Task*$_A$, that updates the value of the evaluation field of the data store by applying the assignment $\mathsf{Judgment.evaluation} := \mathsf{Judgment.evaluation} \wedge (\mathsf{Cake.cake} = \mathsf{Cake.desiderata})$. Then, the *Take Decision* is activated via the *P-Task*$_A$ rule only after both cake judgements have been inserted in the data store. This is achieved by means of the task guard $\mathsf{Judgment.counter} = 2$, where the field counter is increased by one each time an evaluation is inserted in the data store. In this way, the customer that has firstly received a cake must wait for the evaluation of the other customer. The final decision, stored in the data object *Decision*, is subsequently used by both *Customer* instances to evaluate the XOR gateway conditions.

## 5. The MIDA 2.0 animation tool

In this section, we present the version 2.0 of our BPMN animator tool MIDA (*Multiple Instances and Data Animator*) and provide details about its implementation and use. We firstly present the tool and, then, we show how MIDA 2.0 can effectively support designers in debugging their models. In the description we resort to our running example in Fig. 1 to illustrate the tool and its functionalities.

MIDA 2.0 is a web application written in JavaScript, accessible by users via a web browser without installing any plug-in or server backend. MIDA 2.0 has been realised by extending the *"bpmn.io Token Simulation"* plugin by Camunda [17]. The tool as well as source code, binaries, tutorial and example models are freely available at http://pros.unicam.it/mida. As shown in Fig. 17, the graphical interface of the tool consists of four main parts: *(i)* the canvas, where BPMN elements are composed to form a collaboration diagram; *(ii)* the palette, to insert elements in the diagram; *(iii)* the property panel including the Mida tab, to specify attributes of the BPMN elements contained in the diagram; and *(iv)* the data panel, to visualise values of data element fields. MIDA 2.0 permits to locally save models in the standard format *.bpmn* and, hence, to load models previously designed.

### 5.1. From MIDA to MIDA 2.0

MIDA 2.0 revises and extends the previous version of the tool presented in [18]. It now supports both modelling and animation

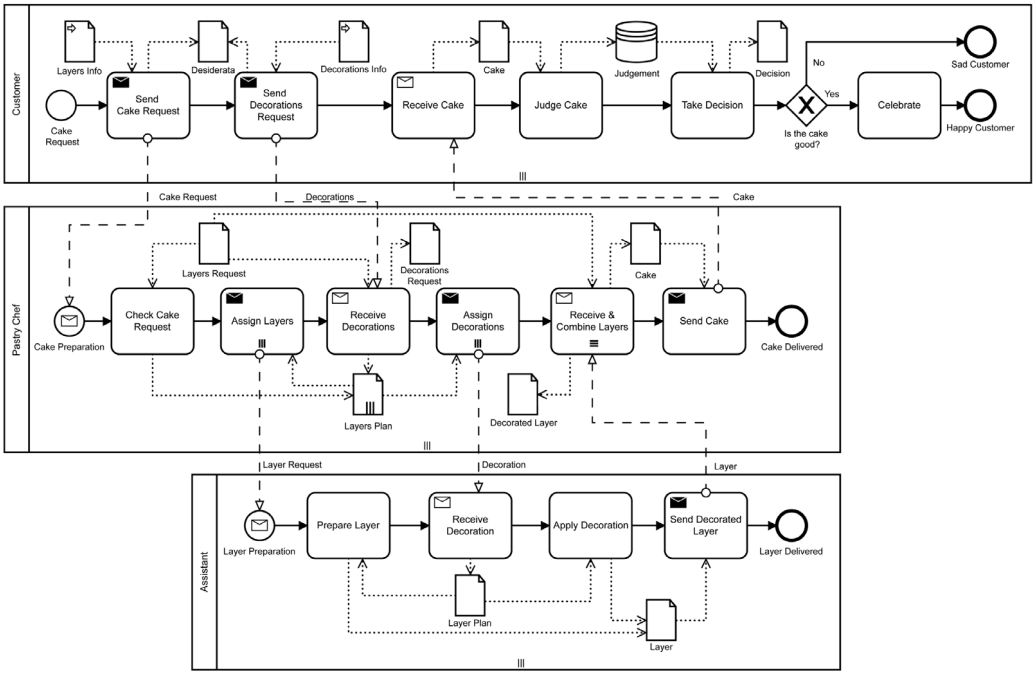



**Fig. 16.** Collaboration with interaction among instances of the same pool.

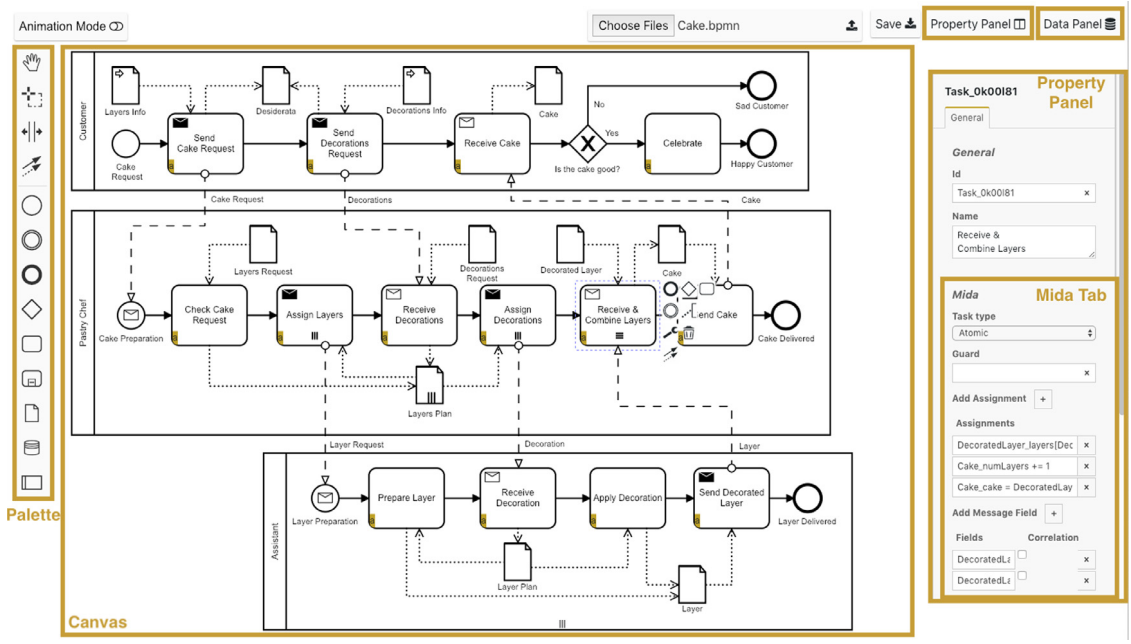



**Fig. 17.** MIDA 2.0 web interface.

of a wider set of BPMN elements included in the current state of the formalisation. The usability of MIDA 2.0 has been also improved. In particular, the introduction of the Mida tab in the property panel guides the designer to easily specify relevant information for multi-instance scenarios (e.g., guards and assignment of tasks, data and message fields, which data must be used for correlation, the maximum and minimum number of instances to be activated in a pool). Previously, all this information was unstructured and collected in description fields of the process elements. The animation and debugging has been also adapted to support the novel elements.

### 5.2. Modelling

The starting point to exploit the MIDA 2.0 functionalities is the modelling of a BPMN collaboration by means of the MIDA 2.0 modelling environment. Notably, the design goes beyond the graphical representation of the collaboration diagram. The property panel plays a key role when modelling collaborations with

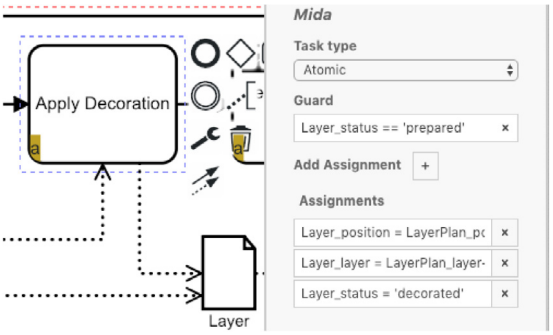



**Fig. 18.** Task guard and assignments.

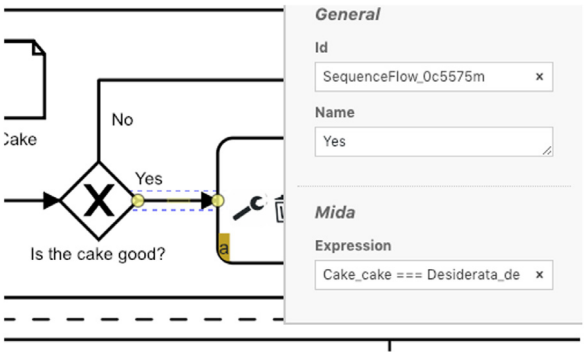



**Fig. 19.** XOR conditions.

MIDA 2.0, as it permits exploiting XML attributes of the *.bpmn* format to specify and save information about the BPMN elements. Different information needs to be provided depending on the considered BPMN element. In particular, information about multi-instance characteristics, data elements, and messages, which represent the specificities of our formal semantics, is introduced during the design of the model by means of the Mida tab.

Let us focus on the multi-instance characteristics. By selecting a pool element, the Mida tab provides two input fields named *Minimum* and *Maximum*. They allow to constrain the number of instances that will be executed for that pool. In our running example, the *Assistant* pool has *Minimum* = '1' and *Maximum* = '3', hence every time a *LayerRequest* message is received by the pool, it triggers the activation of one instance. Three instances at most will be activated for the *Assistant* pool. Differently, multi-instance tasks are defined by selecting the corresponding marker ($|||$ or $\equiv$) in the element context pad and by filling with expressions the *loopCardinality* and the *completionCondition* in the Mida tab. The evaluation of these expressions indicates, respectively, the number of task instances to be executed and the condition for an early termination of the multi-instance task. For example, the parallel multi-instance *AssignLayers* task in our running example has *loopCardinality* = '3' and *completionCondition* = '*false*', meaning that there will be exactly three parallel executions of the task.

Data elements are structured in fields that are specified in the Mida tab as variables names. They can be initialised (e.g., *LayersInfo.top* = '*blue*'), in order to specify data inputs, or left undefined (e.g., *Cake.cake*). Using the Mida tab, users can also select among a simple data object, input/output data object or data object collection. Notably, a data collection represents a list of data items that can be retrieved, modified and reinserted. As prescribed by the formal semantics, MIDA 2.0 provides dedicated functions to support such features.

According to the BPMN standard, the access to data is represented by associations between data elements and tasks that can predicate over field names. These associations can define preconditions for the execution of a task, expressed in MIDA 2.0 as a task *guard*. On the other hand, the effects of a task execution on a data element is instead specified by means of a list of *assignments*. In Fig. 18 we show the guard and the assignments related to the *ApplyDecoration* task introduced in the *Pastry Chef* pool.

Task behaviour can be chosen between *atomic*, *non-atomic concurrent*, or *non-atomic non-concurrent* by means of the Mida tab. A gold label at the right-bottom corner of the task shows the acronym of the execution modality (i.e., a, na_c, na_nc). In our running example all tasks are atomic.

Concerning sending and receiving tasks, the Mida tab allows to specify message fields representing, respectively, message expressions and templates. In particular, in receiving tasks, for each message field there is also the possibility to put a tick on a correlation box. Fields with a tick (e.g., the *LayerPlan.position* message field of *Assistant*) are used for pattern-matching.

Values stored in data elements can be also used by conditions associated to the outgoing sequence edges of XOR split gateways, in order to support decisions. Fig. 19 reports the conditional expression contained into the *Yes* branch of the XOR gateway in the *Customer* process of our running example. It checks if the received cake corresponds to the desiderata and consequently drives the process completion to a *happy* end.

### 5.3. Animation & debugging

The key characteristic of MIDA 2.0 is the animation of collaboration models, which enables models debugging. Anyway, like in software code debugging, the identification and fixing of bugs are still in charge of the human user.

By selecting the *Animation Mode* in the MIDA 2.0 interface, a *play* button will appear over each fireable start event. Every time this button is clicked, a new instance of the desired process is activated, accordingly with the multi-instance constraints specified in the modelling phase. Graphically, this corresponds to the creation of a new token labelled by a fresh instance identifier. Then, as shown in Fig. 20, the token starts to cross the model according to the operational rules induced by our formal semantics. The animation terminates once all tokens cannot move forward, since no semantic rule can be applied. From the Data Panel, users can monitor the evolution of the data state function $\sigma_d$ of each process instance, by observing the values associated by the function to data element fields, which are organised according to the process instances they belong to. Fig. 21 shows how data values change after the execution of the task *ReceiveDecoration* by means of the application of rules *P-TaskRcv*$_A$ and *C-ReceiveMi*, which updates the fields *position* and *decoration* of the *LayerPlan* data object with the values received via the *Decorations* message.

MIDA 2.0 can effectively support designers in identifying issues in their business processes. By pausing the animation, he/she can take a careful look at the flow of tokens and at the evolution of data representing the state functions $\sigma_e$ and $\sigma_d$, respectively, in order to immediately detect unwanted behaviours. Moreover, if a token remains blocked or violates conditions (e.g., guard conditions, XOR conditions, constraints about maximum number of instances), MIDA 2.0 highlights it in red, as shown in Fig. 22 where the violation comes from the presence of an undefined field in the *Layers Info* data input.

Considering our running example, receiving a *LayerRequest* message triggers the activation of a new process instance of *Assistant* (rules *P-StartRcv* in combination with *C-CreateMi*), while a *Decoration* message has to be routed to an already existing *Assistant* instance (rules *P-TaskRcv*$_A$ in combination with

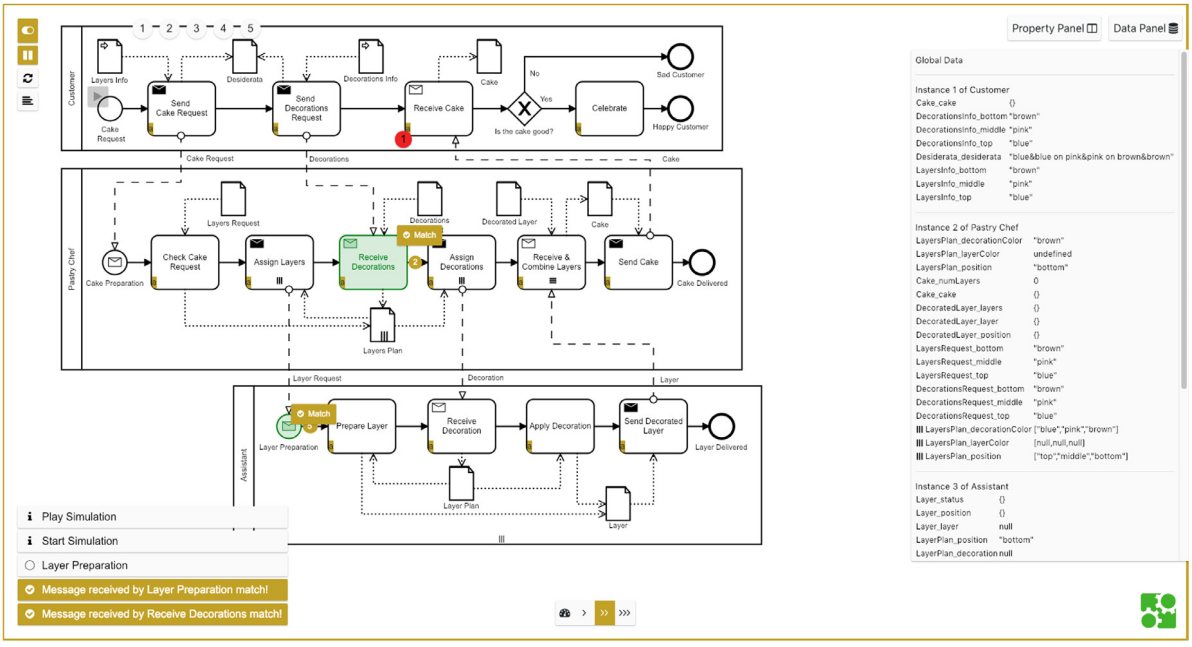



**Fig. 20.** MIDA 2.0 Animation.

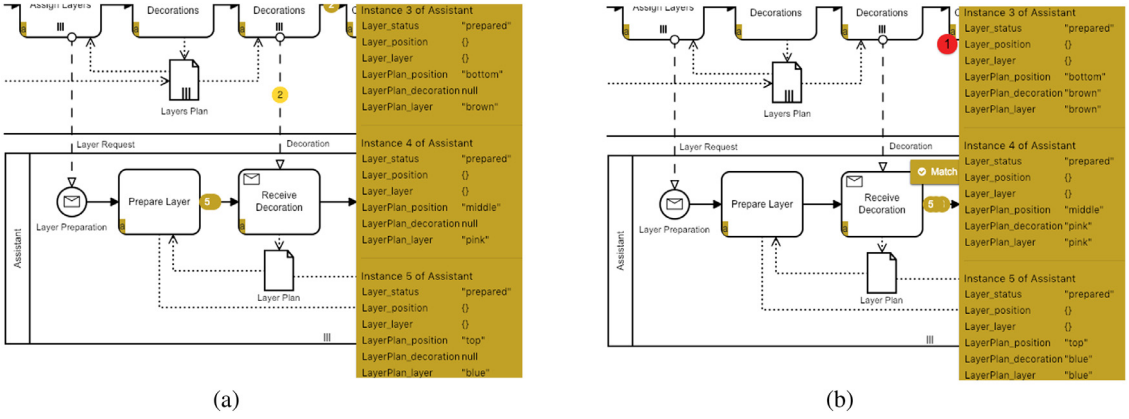



**Fig. 21.** Data Panel before (a) and after (b) the execution of task *Receive Decoration*.

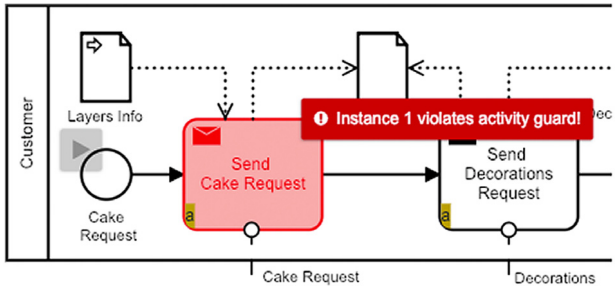



**Fig. 22.** Guard violation.

*C-ReceiveMi*). Hence, in the latter case, the message needs to be properly correlated to the right instance. Otherwise, the *Pastry Chef* risks to receive back cake layers with a wrong decoration. To ensure the correct correlation, the *LayerRequest* message contains a field representing the position of the cake layer. Each *Assistant* instance stores this information inside the *Layer* data object. Then, the *Assistant* uses the position value for checking

the pattern-matching with the value contained in the received *Decoration* messages. However, if the correlation check is not properly specified in the task *ReceiveDecoration* (e.g., no correlation data is provided), decorations are applied on layers randomly. This results with a cake different from the desired one. However, MIDA 2.0 allows to detect, and hence solve, this correlation issue. Similarly, malformed or unexpected messages may introduce deadlocks in the execution flow, which can be easily identified by looking for blocked tokens in the animation. For instance, in the running example a *Decoration* message without the *decorationColor* field would be never consumed by the task *Receive Decoration* of the *Assistant* pool, because the premise of rule *C-ReceiveMi* performing the *match* function check is not satisfied, thus making the rule not applicable. Finally, since our animation is based on data elements fields, also issues due to bad data handling can be detected using MIDA 2.0. Let us suppose that the sequential multi-instance receive task *Receive & Combine Layers* in the pool *Pastry Chef* has a wrong loop cardinality set to two (instead of three). Rule $P\text{-}MisTask_1$ activates the multiple instance task. Then, after the receiving of two *Layers* messages from the *Assistant*, rules $P\text{-}TaskRcv_A$ and $P\text{-}MisTask_3$ can be applied, and the *Pastry Chef* composes a cake with just two layers. Finally, rules $P\text{-}MisTask_3$ completes the execution of multiple-instance task passing the token to the *Send Cake* task. However, the rule $P\text{-}TaskSnd_A$ related to the task *Send Cake* cannot be applied, since the guard condition $Cake.numLayers = 3$ is violated. This results in a deadlock easily detectable by means of MIDA 2.0.

To sum up, the MIDA 2.0 tool can support designers in debugging their multi-instance collaboration models, as it permits to check the evolution of data, messages and processes marking while executing the models step-by-step.

## 6. Related work

In this section we discuss the most relevant attempts in formalising multiple instances and data for BPMN models. We then compare MIDA with other animation tools.

***On Formalising Multiple Instances and Data.*** Many works in the literature attempted to formalise the core features of BPMN. However, most of them (see, e.g., [3–10]) do not consider multiple instances and data, which are the focus of our work. Considering these features in BPMN collaborations, relevant works are [19–22]. Meyer et al. in [19] focus on process models where data objects are shared entities and the correlation mechanism is used to distinguish and refer data object instances. Use of data objects local to (multiple) instances, exchange of messages between participants, and correlation of messages are instead our focus. In [20], the authors describe a model-driven approach for BPMN to include the data perspective. Differently from us, they do not provide a formal semantics for BPMN multiple instances. Moreover, they do not use data in decision gateways. Moreover, Kheldoun et al. propose in [21] a formal semantics of BPMN covering features such as message-exchange, cancellation, multiple instantiation of sub-processes and exception handling, while taking into account data flow aspects. However, they do not consider multi-instance pools and do not address the correlation issue. Semantics of data objects and their use in decision gateways is instead proposed by El-Saber and Boronat in [22]. Differently from us, this formal treatment does not include collaborations and, hence, exchange of messages and multiple instances. Considering other modelling languages, YAWL [23] and high-level Petri nets [24] provide direct support for the multiple instance patterns. However, they lack support for handling data. In both cases, process instances are characterised by their identities, rather than by the values of their data, which are however necessary to correlate messages to running instances.

Regarding choreographies, relevant works are [25–27]. López et al. [25] study the choreography problem derived from the synchronisation of multiple instances necessary for the management of data dependencies. Knuplesch et al. [26] introduces a data-aware collaboration approach including formal correctness criteria. However, they define the data perspective using data-aware interaction nets, a proprietary notation, instead of the wider accepted BPMN. Improving data-awareness and data-related capabilities for choreographies is the goal of Hahn et al. [27]. They propose a way to unify the data flow across participants with the data flow inside a participant. The scope of data objects is global to the overall choreography, while we consider data objects with scope local to participant instances, as prescribed by the BPMN standard. Apart from the specific differences mentioned above, our work differs from the others for the focus on collaboration diagrams, rather than on choreographies. This allows us to specifically deal with multiple process instantiation and messages correlation.

Concerning the correlation mechanism, the BPMN standard and, hence, our work have been mainly inspired by works in the area of service-oriented computing (see the relationship between BPMN and WS-BPEL [28] in [1, Sec. 14.1.2]). In fact, when a service engages in multiple interactions, it is generally required to create an instance to concurrently serve each request, and correlate subsequent incoming messages to the created instances. Among the others, the COWS [16] formalism captures the basic aspects of service-oriented systems, and in particular service instantiation and message correlation à la WS-BPEL. From the formal point of view, correlation is realised by means of a pattern-matching function similar to that used in our formal semantics. Let us focus more on how correlation is dealt with in BPMN [1, Sec. 8.3.2]. The standard identifies two mechanisms to manage the correlation of messages with process instances. The first is a *key-based* mechanism that couples sender and receiver by means of the concept of correlation key. Any message, to be properly correlated, needs to carry values of a correlation key within its payload. Those values are initialised during the first interaction and then extracted, even partially, to correctly match the follow-up messages. The second is instead a *context-based* mechanism, as it depends on the process data (i.e., the content of data elements) associated to the process instances. This is a more expressive form of correlation with respect to key-based correlation, since this latter can only populate a correlation key implicitly from the values of the first message. Instead, in this case a correlation key can contain formal expressions dynamically evaluated at runtime using the process context, hence the correlation key can be automatically updated whenever the underlying data elements change. *"In that sense, changes in the Process context can alter the correlation condition"* [1, p. 75]. Similarly, in our formalisation we define correlation keys in the receiving elements to match the correct messages and to store their payload. In particular, in our case correlation keys are identified by the template fields that are not formal (i.e., without the ?-tag). Since non formal fields are either values or expressions, our approach based on pattern-matching is able to mime both the key-based and the context-based mechanism. However, BPMN considers a correlation key as *"a composite key out of one or many CorrelationProperties that essentially specify extraction Expressions atop Messages"* and, in the context-base case, *"atop the Process context"* [1, pp. 75–76]. Thus, correlation properties can be quite articulate expressions, especially when they have to be used to match messages with complex structures. Instead, in our case, messages have a simple tuple structure, i.e., they are ordered lists of values. As consequence, our templates are ordered lists as well, and the correlation mechanism, once the template expressions have been evaluated, simply performs a (field-by-field) pattern-matching check.

Please cite this article as: F. Corradini, C. Muzi, B. Re et al., Formalising and animating multiple instances in BPMN collaborations, Information Systems (2019) 101459, https://doi.org/10.1016/j.is.2019.101459.

Concerning data-awareness in process modelling, several works refer to the data-centric approach (see, e.g., the surveys [29–31]). This approach uses data elements as first class citizens and focusses on their life cycles (i.e., on the data flow) [32]. Differently, our approach focusses on BPMN as reference language, thus concentrating on activities and, more generally, on control flow. Data elements act as pre- and post-conditions for activity execution, and as main decision indicator at exclusive gateways. Moreover, our main interest is the study of the BPMN management of multiple instances that, even if it is affected by data, keeps the focus on the control flow perspective.

Finally, to the best of our knowledge, no work in the literature permits to specify different execution modalities (i.e., atomic, non-atomic concurrent, non-atomic non-concurrent) for tasks of BPMN models. This feature allows us to study, e.g., the impact of different settings of such modalities in BPMN models involving multi-instance tasks that access data.

***Business Process Animation.*** Relevant contributions about animation of business processes are proposed in literature and from modelling tool vendors. Differently from us, in these implementations the interplay between multiple instances, messages and data is not fully supported. Allweyer and Schweitzer [33] propose a tool for animating BPMN models that considers only processes, as it discards message exchanges, both semantically and graphically. In addition, gateway decisions are performed manually by users during the animation, instead of depending on data. Aysolmaz [34] proposes an animator for BPMN process models, decoupling the animation from the modelling that is not directly supported by the tool. Even though the graphic approach is notable, several modelling elements are not implemented in the tool, in particular data and multiple instances. The animator of the Signavio [35] modeller allows users to step through the process element-by-element and to focus completely on the process flow. However, it discards important elements, such as message flows and data objects. Hence, Signavio animates only non-collaborative processes, without data-driven decisions, which instead are key features of our approach. Finally, Visual Paradigm [36] provides an animator that supports also collaboration diagrams. This tool allows users to visualise the flow of messages and implements the semantics of receiving tasks and events, but it does not animate data evolution and multiple instances.

## 7. Concluding remarks

In this paper we provide a novel operational semantics clarifying the interplay between control features, data, message exchanges and multiple instances. Moreover, we propose MIDA, an animator tool, based on our formal semantics, that provides the visualisation of the behaviour of a collaboration by taking into account the data-based correlation of messages to process instances. We have shown, on our running example, that MIDA supports designers to spot erroneous behaviours. It is worth noticing that beyond the case study we use in the paper as running example, in the MIDA web page we also make available a collection of BPMN models referring to different scenarios ready to be animated (concerning, e.g., conference paper reviewing, travel booking, smart home heating system management, procedures for student internship and exam registration).[3] Each example is provided in different variants, to show to the user how MIDA can spot different execution issues.

[3] These BPMN models are available at: https://bitbucket.org/proslabteam/mida/src/36a18b195b5a/assets/examples/.

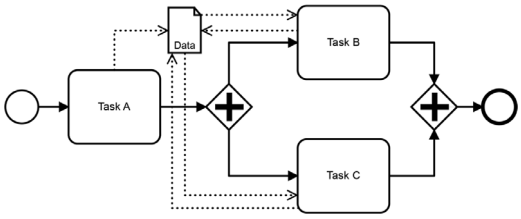



**Fig. 23.** Atomic vs non-atomic task execution.

We conclude the paper by discussing lessons learned, and the assumptions and limitations of our approach, also touching upon directions for future work.

***Lessons learned.*** The BPMN standard has the flavour of a framework rather than of a concrete language, because some aspects are not covered by it, but left to the designer [37]. For example, the standard left underspecified the internal structure of data objects: "*Data Object elements can optionally reference a DataState element [...] The definition of these states, e.g., possible values and any specific semantics are out of scope of this specification*" [1, p. 206]. The situation does not change if we refer to the internal structure of data stores and data collections. This gap left by the BPMN standard must be filled in order to concretely deal with data in our formalisation, and hence in the animation of BPMN collaboration models. To this aim, we consider a generic record structure for data elements. Similarly, the expression language operating on data is left unspecified by the standard. This is not an issue for the formalisation, but the expression language has to be instantiated in the concrete implementation of the animator. In MIDA 2.0, for the sake of simplicity, we resort to the expression language of JavaScript, as this is the language used for implementing the tool.

The BPMN standard also lacks of a clear description of the task execution modality. The paper contributes to fill this gap thanks to the capability of our semantics to manage different modalities of task execution taking into account atomicity and concurrency. We explain this feature by means of the process model in Fig. 23. It provides a minimal example whose execution consists in performing firstly *Task A*, then *Task B* and *Task C* in parallel, and when both complete, the process ends. All these tasks can access to the data object *Data* composed by three fields *a*, *b*, and *c*. Guards and assignments are specified as follows: *(i) Task A* has *true* as guard condition and performs the assignment $Data.a := 1$; *(ii) Task B* has $Data.a = 1$ as guard and performs $Data.b := 2$ and $Data.a := 0$ as assignments; and *(iii) Task C* has $Data.a = 1$ as guard and performs $Data.c := 5$ and $Data.a := 0$ as assignments. The execution of this process can produce different results depending on the execution modality setting of its tasks. In case all tasks are executed with the atomic modality, the assignment to the field *a* performed by the firstly executed task between *Task B* and *Task C* disables the other task (because it makes the task's guard become *false*). Therefore, regardless the order of execution of the parallel tasks, the execution of the overall process never reaches the end. In the non-atomic case, instead, different process executions can take place, as the task execution is now split in two steps: *(i)* guard evaluation, and *(ii)* assignments execution. Depending on how these steps of the execution of *Task B* and *Task C* interleave, the process may ends properly or not. Finally, in case the tasks could be activated more than once at the same time (e.g., in case they would be multi-instance tasks, or in case of an unsafe process), the overall behaviour would be also affected by the setting of the concurrent execution modality, which may allow or not the interleaved execution among instances of the same task.

Please cite this article as: F. Corradini, C. Muzi, B. Re et al., Formalising and animating multiple instances in BPMN collaborations, Information Systems (2019) 101459, https://doi.org/10.1016/j.is.2019.101459.

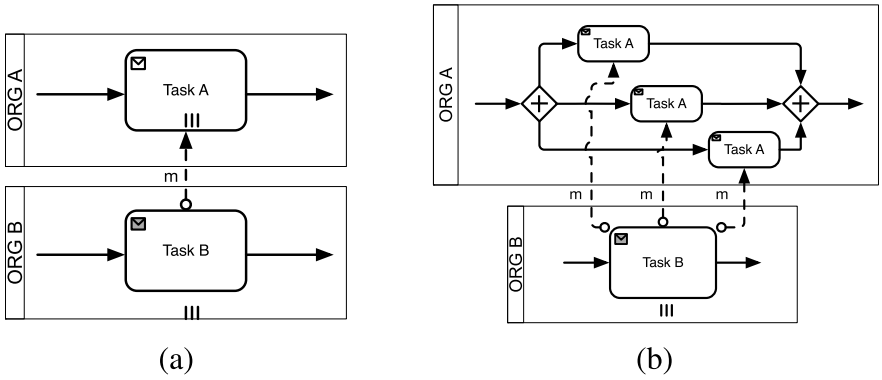



**Fig. 24.** Parallel multi-instance send tasks (a) and its macro expansion (b).

In addition, the lack of a formal semantics in the standard may lead to different interpretations of the tricky features of BPMN. In this work we aim at clarifying the interplay between multiple instances, messages and data objects. In particular, the standard provides an informal description of the mechanism used to correlate messages and process instances [1, p. 74], which we have formalised and implemented by following the solution adopted by the standard for executable business processes [28]. However, the BPMN standard does not provide any hint on how instances of the same process can communicate with each other. As discussed in Section 4, this may be particular useful in practical scenarios when instances of the same pool have to coordinate (in case, e.g., they need to achieve an agreement on a shared decision). To this aim, we have exploited a data store to share persistent information among the instances.

Finally, even if from the semantic point of view it is a common practice to consider multi-instance tasks as macros, we have provided in this paper a direct characterisation of their semantics. Concerning the sequential case, we are aware that the multi-instance task can be simply dealt with as a macro: it corresponds to a task enclosed within a sort of 'for' loop. Indeed, this was the solution we adopted in [14]. However, to keep track of the number of executed instances it is necessary to add to the model a further data object, to be used as a counter, for each multi-instance task. Moreover, to provide a complete specification of this BPMN element, loop cardinality expression and completion condition have to be considered. Even if formally sound, the use of this macro alters the original model, increasing its complexity, and hence is not practical in supporting tools, like the MIDA 2.0 animator. This is why we have decided to introduce a syntactic term for sequential multi-instance tasks with its own semantic rules. The parallel case, instead, is more tricky. It is commonly considered as a macro as well: the parallel multi-instance task is thought of as a set of tasks between AND split and join gateways [6,23], assuming to know at design time the number of instances to be generated. However, this replacement is no longer admissible when this kind of element is used within multi-instance pools, thus requiring a direct definition of its formal semantics. In fact, consider for example the collaboration fragment in Fig. 24(a), where a multi-instance receiving task communicates with a multi-instance pool. Supposing to have three instances of *Task A*, by applying the mentioned macro replacement we would obtain the collaboration fragment in Fig. 24(b), which however is not semantically equivalent. Indeed, each instance of *ORG B* in Fig. 24(a) has a *Task B* that sends only one message, while in Fig. 24(b) each instance has a *Task B* sending three times the same messages, one for each copy of *Task A* in *ORG A*. This suggests that parallel multi-instance tasks are not simple macros, but they require their own direct formalisation, as we have done in our work.

***Assumptions and limitations.*** Our formal semantics focusses on the communication mechanisms of collaborative systems, where multiple participants cooperate and share information. Thus, we have left out those features of BPMN whose formal treatment is orthogonal to the addressed problem, such as timed events and error handling.

Moreover, to keep our formalisation more manageable, sub-processes are left out, despite they can be relevant for multi-instance collaborations. Introducing sub-processes in our formalisation cannot be done by including it as a mere macro. In fact, in general, simply flattening a process by replacing its sub-process elements by their expanded processes results in a model with different behaviour. This because a sub-process, for example, delimits the scope of the enclosed data objects and confines the effect of termination events. Therefore, it would be necessary to explicitly deal with the resulting multi-layer perspective, which adds complexity to the formal treatment. The formalisation would become even more complex if we consider multi-instance sub-processes, which would require an extension of the correlation mechanism.

Finally, in our work we use concrete values for data elements, since our aim is to support model *animation*. This data management approach is however not adequate to support model *simulation*, where many executions of the same model can be automatically produced by constraining the involved values.

***Future Work.*** We plan to continue our programme to effectively support modelling and animation of BPMN multi-instance collaborations, by overcoming the above limitations. In particular, we intend to extend the treatment of data perspective, by considering more sophisticated definitions of the internal structure of data elements (e.g., based on UML) with respect to the generic record structure considered here. In addition, we plan to investigate other expression languages operating on data (e.g., the language FEEL [38, Ch. 10]). Moreover, we plan to extend our work, both from the formal and practical perspective. To this aim, we intend to first define a symbolic formal semantics, like that in [39], and then to extend the MIDA implementation accordingly. The use of SMT solvers will be considered to deal with the constraints generated by the symbolic semantics. Finally, we plan to exploit the formal semantics, and its implementation, to enable the verification of properties using, e.g., model checking techniques.

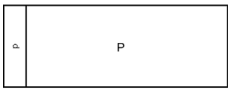
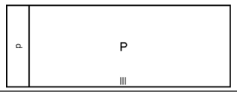
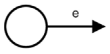
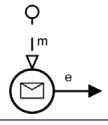
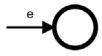
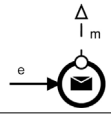
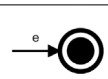
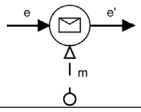
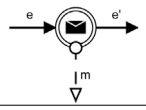
## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgment

This work has been partially supported by the PRIN projects "SEDUCE" n. 2017TWRCNB and "Fluidware" n. 2017KRC7KT.

**Table A.1**
Correspondence between graphical and textual notation: pools and events.

| Graphical Notation | Textual Notation |
|---|---|
|   | pool(p, $P$) |
|   | miPool(p, $P$, max) |
|   | start(e′, e) |
|   | startRcv(m : t̃, e) |
|   | end(e) |
|   | endSnd(e, m : ẽxp) |
|   | terminate(e) |
|   | interRcv(e, m : t̃, e′) |
|   | interSnd(e, m : ẽxp, e′) |

## Appendix A. One-to-one correspondence between the BPMN graphical notation and textual notation and our textual notation (most relevant cases).

See Tables A.1–A.3.

## Appendix B. Omitted formal definitions of auxiliary functions and relations

We report in this appendix the formal definitions of auxiliary functions and relations that, for the sake of readability, have been omitted in the body of the paper.

- Functions *inc* and *dec* on $\sigma_e$ extend in a natural ways to sets $E$ of edges. Specifically, *inc* is inductively defined as follows:
  - $inc(\sigma_e, \varnothing) = \sigma_e$
  - $inc(\sigma_e, \{\mathsf{e}\} \cup E) = inc(inc(\sigma_e, \mathsf{e}), E)$

  while *dec* is inductively defined as follows:
  - $dec(\sigma_e, \varnothing) = \sigma_e$
  - $dec(\sigma_e, \{\mathsf{e}\} \cup E) = dec(dec(\sigma_e, \mathsf{e}), E)$
- Function *reset* on $\sigma_e$ extends in a natural ways to sets of edges as follows:
  - $reset(\sigma_e, \varnothing) = \sigma_e$
  - $reset(\sigma_e, \{\mathsf{e}\} \cup E) = reset(reset(\sigma_e, \mathsf{e}), E)$
- Relation *upd* is inductively defined as follows, for any $\sigma_{do}$, $\sigma_{dc}$, $\sigma_{ds}$:
  - $upd(\sigma_{do}, \sigma_{dc}, \sigma_{ds}, \epsilon, \sigma_{do}, \sigma_{dc}, \sigma_{ds})$
  - $upd(\sigma_{do}, \sigma_{dc}, \sigma_{ds}, \mathsf{do.f} := \mathsf{exp}, \sigma_{do} \cdot [\mathsf{do.f} \mapsto \mathsf{v}], \sigma_{dc}, \sigma_{ds})$ with v such that $eval(\mathsf{exp}, \sigma_{do}, \sigma_{dc}, \sigma_{ds}, \mathsf{v})$
  - $upd(\sigma_{do}, \sigma_{dc}, \sigma_{ds}, \mathsf{ds.f} := \mathsf{exp}, \sigma_{do}, \sigma_{dc}, \sigma_{ds} \cdot [\mathsf{ds.f} \mapsto \mathsf{v}])$ with v such that $eval(\mathsf{exp}, \sigma_{do}, \sigma_{dc}, \sigma_{ds}, \mathsf{v})$
  - $upd(\sigma_{do}, \sigma_{dc}, \sigma_{ds}, \mathsf{get(do)}, \sigma_{do} \cdot \sigma^1_{do}, \sigma'_{dc}, \sigma_{ds})$ with $\sigma_{dc}(\mathsf{do}) = \langle \sigma^1_{do}, \sigma^2_{do}, \ldots, \sigma^n_{do} \rangle$ and $\sigma'_{dc}$ such that $\sigma'_{dc}(\mathsf{do}) =$

**Table A.2**
Correspondence between graphical and textual notation: gateways.

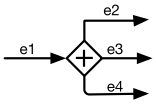
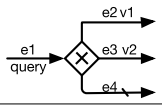
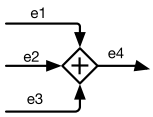
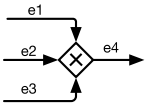
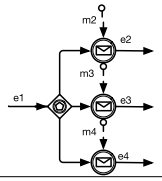
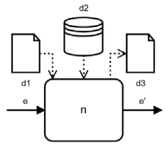
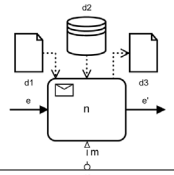
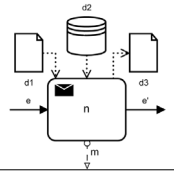
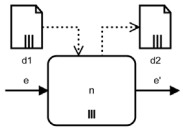
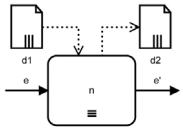
| Graphical Notation | Textual Notation |
|---|---|
|   | andSplit(e1, {e2, e3, e4}) |
|   | xorSplit(e1, {(e2, query = $\mathsf{v}_1$), (e3, query = $\mathsf{v}_2$), (e4, default)}) |
|   | andJoin({e1, e2, e3}, e4) |
|   | xorJoin({e1, e2, e3}, e4) |
|   | eventBased(e1, (m2 : t̃$_2$, e2), (m3 : t̃$_3$, e3), (m4 : t̃$_4$, e4)) |

**Table A.3**
Correspondence between graphical and textual notation: tasks.

| Graphical Notation | Textual Notation |
|---|---|
|  | $\mathsf{task}(\mathsf{e}, \mathsf{n}, N, \mathsf{exp}(\mathsf{d}_1, \mathsf{d}_2), (\mathsf{d}_3.\mathsf{f}_1 := \mathsf{exp}_1, \ldots, \mathsf{d}_3.\mathsf{f}_n := \mathsf{exp}_n), \mathsf{e}')$ |
|  | $\mathsf{taskRcv}(\mathsf{e}, \mathsf{n}, N, \mathsf{exp}(\mathsf{d}_1, \mathsf{d}_2), (\mathsf{d}_3.\mathsf{f}_1 := \mathsf{exp}_1, \ldots, \mathsf{d}_3.\mathsf{f}_n := \mathsf{exp}_n), \mathsf{m}:\tilde{\mathsf{t}}, \mathsf{e}')$ |
|  | $\mathsf{taskSnd}(\mathsf{e}, \mathsf{n}, N, \mathsf{exp}(\mathsf{d}_1, \mathsf{d}_2), (\mathsf{d}_3.\mathsf{f}_1 := \mathsf{exp}_1, \ldots, \mathsf{d}_3.\mathsf{f}_n := \mathsf{exp}_n), \mathsf{m}:\tilde{\mathsf{exp}}, \mathsf{e}')$ |
|  | $\mathsf{mipTask}(\mathsf{e}, \mathsf{exp}, \mathsf{task}(\mathsf{e}'', \mathsf{n}, N, \mathsf{notEmpty}(\mathsf{d}_1), (\mathsf{get}(\mathsf{d}_1), \mathsf{d}_2.\mathsf{f}_1 := \mathsf{exp}_1, \ldots, \mathsf{d}_2.\mathsf{f}_n := \mathsf{exp}_n, \mathsf{push}(\mathsf{d2})), \mathsf{e}'''), \mathsf{c}, \mathsf{exp}', \mathsf{e}')$ |
|  | $\mathsf{misTask}(\mathsf{e}, \mathsf{exp}, \mathsf{task}(\mathsf{e}'', \mathsf{n}, N, \mathsf{notEmpty}(\mathsf{d}_1), (\mathsf{get}(\mathsf{d}_1), \mathsf{d}_2.\mathsf{f}_1 := \mathsf{exp}_1, \ldots, \mathsf{d}_2.\mathsf{f}_n := \mathsf{exp}_n, \mathsf{push}(\mathsf{d2})), \mathsf{e}'''), \mathsf{c}, \mathsf{exp}', \mathsf{e}')$ |

$$\frac{\sigma_i(\mathsf{p}) = \{\langle \sigma_e, \sigma_{do}, \sigma_{dc}, \sigma_t, \sigma_c \rangle\} \quad \langle P, \sigma_e, \sigma_{do}, \sigma_{dc}, \sigma_{ds}, \sigma_t, \sigma_c \rangle \xrightarrow{\tau} \langle \sigma'_e, \sigma'_{do}, \sigma'_{dc}, \sigma'_{ds}, \sigma'_t, \sigma'_c \rangle}{\mathsf{pool}(\mathsf{p}, P) \xrightarrow{\tau} \langle updInst(\sigma_i, \mathsf{p}, \sigma'_e, \sigma'_{do}, \sigma'_{dc}, \sigma'_t, \sigma'_c, \varnothing), \sigma'_{ds} \rangle} \; (C\text{-}Internal)$$

$$\frac{\sigma_i(\mathsf{p}) = \{\langle \sigma_e, \sigma_{do}, \sigma_{dc}, \sigma_t, \sigma_c \rangle\} \quad \langle P, \sigma_e, \sigma_{do}, \sigma_{dc}, \sigma_{ds}, \sigma_t, \sigma_c \rangle \xrightarrow{?\mathsf{m}:\tilde{\mathsf{et}}, A} \langle \sigma'_e, \sigma'_d, \sigma'_t, \sigma'_c \rangle \quad \tilde{\mathsf{v}} \in \sigma_m(\mathsf{m}) \quad match(\tilde{\mathsf{et}}, \tilde{\mathsf{v}}) = A' \quad upd(\sigma'_d, (A', A), \sigma''_{do}, \sigma''_{dc}, \sigma''_{ds})}{\mathsf{pool}(\mathsf{p}, P) \xrightarrow{?\mathsf{m}:\tilde{\mathsf{v}}} \langle updInst(\sigma_i, \mathsf{p}, \{\langle \sigma'_e, \sigma''_{do}, \sigma''_{dc}, \sigma'_t, \sigma'_c \rangle\}), rm(\sigma_m, \mathsf{m}, \tilde{\mathsf{v}}), \sigma''_{ds} \rangle} \; (C\text{-}Receive)$$

$$\frac{\sigma_i(\mathsf{p}) = \{\langle \sigma_e, \sigma_{do}, \sigma_{dc}, \sigma_t, \sigma_c \rangle\} \quad \langle P, \sigma_e, \sigma_{do}, \sigma_{dc}, \sigma_{ds}, \sigma_t, \sigma_c \rangle \xrightarrow{!\mathsf{m}:\tilde{\mathsf{v}}} \langle \sigma'_e, \sigma'_{do}, \sigma'_{dc}, \sigma'_{ds}, \sigma'_t, \sigma'_c \rangle}{\mathsf{pool}(\mathsf{p}, P) \xrightarrow{!\mathsf{m}:\tilde{\mathsf{v}}} \langle updInst(\sigma_i, \mathsf{p}, \{\langle \sigma'_e, \sigma'_{do}, \sigma'_{dc}, \sigma'_t, \sigma'_c \rangle\}), add(\sigma_m, \mathsf{m}, \tilde{\mathsf{v}}), \sigma'_{ds} \rangle} \; (C\text{-}Deliver)$$

**Fig. B.25.** BPMN collaboration semantics (omitted rules).

$\langle \sigma^2_{do}, \ldots, \sigma^n_{do} \rangle$ and $\sigma'_{dc}(\mathsf{do}') = \sigma_{dc}(\mathsf{do}')$ with $\mathsf{do} \neq \mathsf{do}'$, where $\sigma_{do} \cdot \sigma'_{do} = \sigma_{do} \cdot [\mathsf{do}_1.\mathsf{f}_1 \mapsto \sigma'_{do}(\mathsf{do}_1.\mathsf{f}_1), \ldots, \mathsf{do}_n.\mathsf{f}_n \mapsto \sigma'_{do}(\mathsf{do}_n.\mathsf{f}_n)]$, $\forall \mathsf{do}_i.\mathsf{f}_i \in \mathbb{F}$ such that $\sigma'_{do}(\mathsf{do}_i.\mathsf{f}_i) \neq \mathsf{null}$

- $upd(\sigma_{do}, \sigma_{dc}, \sigma_{ds}, \mathsf{push}(\mathsf{do}), \sigma_{do}, \sigma'_{dc}, \sigma_{ds})$ with $\sigma_{dc}(\mathsf{do}) = \langle \sigma^1_{do}, \sigma^2_{do}, \ldots, \sigma^n_{do} \rangle$ and $\sigma'_{dc}$ such that $\sigma'_{dc}(\mathsf{do}) = \langle \sigma^1_{do}, \sigma^2_{do}, \ldots, \sigma^n_{do}, \sigma'_{do} \rangle$ and $\sigma'_{dc}(\mathsf{do}') = \sigma_{dc}(\mathsf{do}')$ for $\mathsf{do}' \neq \mathsf{do}$, and $\sigma'_{do}$ such that $\sigma'_{do}(\mathsf{do}''.\mathsf{f}) = \sigma_{do}(\mathsf{do}''.\mathsf{f})$ if $\mathsf{do}'' = \mathsf{do}$ and $\sigma'_{do}(\mathsf{do}''.\mathsf{f}) = \mathsf{null}$ if $\mathsf{do}'' \neq \mathsf{do}$
- $upd(\sigma_{do}, \sigma_{dc}, \sigma_{ds}, (A_1, A_2), \sigma''_{do}, \sigma''_{dc}, \sigma''_{ds})$ with $\sigma''_{do}, \sigma''_{dc}, \sigma''_{ds}$ such that $upd(\sigma'_{do}, \sigma'_{dc}, \sigma'_{ds}, A_2, \sigma''_{do}, \sigma''_{dc}, \sigma''_{ds})$ and $\sigma'_{do}, \sigma'_{dc}, \sigma'_{ds}$ such that $upd(\sigma_{do}, \sigma_{dc}, \sigma_{ds}, A_1, \sigma'_{do}, \sigma'_{dc}, \sigma'_{ds})$

- Function *match* is inductively defined as follows:
  - $match(\mathsf{v}, \mathsf{v}) = \epsilon$
  - $match(?\mathsf{do.f}, \mathsf{v}) = (\mathsf{do.f} := \mathsf{v})$
  - $match(?\mathsf{ds.f}, \mathsf{v}) = (\mathsf{ds.f} := \mathsf{v})$
  - $match((\mathsf{et}', \tilde{\mathsf{et}}), (\mathsf{v}', \tilde{\mathsf{v}})) = match(\mathsf{et}', \mathsf{v}'), match(\tilde{\mathsf{et}}, \tilde{\mathsf{v}})$
- The omitted rules of the collaboration semantics are reported in Fig. B.25.

Please cite this article as: F. Corradini, C. Muzi, B. Re et al., Formalising and animating multiple instances in BPMN collaborations, Information Systems (2019) 101459, https://doi.org/10.1016/j.is.2019.101459.

## Appendix C. Textual representation of the running example revisions

For sake of presentation, guards that check the initialisation of data fields are omitted in the following model specifications.

*Overall cake preparation collaboration scenario:*

$\mathsf{pool}(\mathsf{p}_c, P_c) \parallel \mathsf{miPool}(\mathsf{p}_p, P_p, 2) \parallel \mathsf{miPool}(\mathsf{p}_a, P_a, 6)$

*Customer process* :

$P_c = \mathsf{start}(\mathsf{e}_1, \mathsf{e}_2) \parallel \mathsf{misTask}(\mathsf{e}_2, 2, \mathsf{taskSnd}(\mathsf{e}'_2, \mathsf{SendCakeRequest}, \mathsf{a}, \mathsf{exp}_1, \mathsf{A}_1, \mathsf{CakeRequest}:\tilde{\mathsf{exp}}_2, \mathsf{e}'_3), \mathsf{c}_1, \mathsf{false}, \mathsf{e}_3) \parallel$
$\mathsf{misTask}(\mathsf{e}_3, 2, \mathsf{taskSnd}(\mathsf{e}'_3, \mathsf{SendDecorationsRequest}, \mathsf{a}, \mathsf{exp}_3, \mathsf{A}_2, \mathsf{Decorations}:\tilde{\mathsf{exp}}_4, \mathsf{e}'_4), \mathsf{c}_2, \mathsf{false}, \mathsf{e}_4) \parallel$
$\mathsf{misTask}(\mathsf{e}_4, 2, \mathsf{taskRcv}(\mathsf{e}'_4, \mathsf{ReceiveCake}, \mathsf{a}, \mathsf{true}, \mathsf{A}_3, \mathsf{Cake}:\tilde{\mathsf{t}}_1, \mathsf{e}'_5), \mathsf{c}_3, \mathsf{false}, \mathsf{e}_5) \parallel$
$\mathsf{task}(\mathsf{e}_5, \mathsf{JudgeCakes}, \mathsf{a}, \mathsf{exp}_5, \mathsf{A}_4, \mathsf{e}_6) \parallel \mathsf{xorSplit}(\mathsf{e}_6, \{(\mathsf{e}_7, \mathsf{Evaluation.result} = \mathsf{false}), (\mathsf{e}_8, \mathsf{Evaluation.result} = \mathsf{true})\}) \parallel$
$\mathsf{end}(\mathsf{e}_7) \parallel \mathsf{task}(\mathsf{e}_8, \mathsf{Celebrate}, \mathsf{a}, \mathsf{true}, \epsilon, \mathsf{e}_9) \parallel \mathsf{end}(\mathsf{e}_9)$

*Templates*, *expressions*, *assignments* :

$A_1 = \mathsf{get}(\mathsf{LayersInfo}), \mathsf{LayersDesiderata.top} := \mathsf{LayersInfo.top}, \mathsf{LayersDesiderata.middle} := \mathsf{LayersInfo.middle},$
$\mathsf{LayersDesiderata.bottom} := \mathsf{LayersInfo.bottom}, \mathsf{push}(\mathsf{LayersDesiderata})$

$\tilde{\mathsf{exp}}_2 = \langle \mathsf{LayersInfo.top}, \mathsf{LayersInfo.middle}, \mathsf{LayersInfo.bottom}, \mathsf{LayersInfo.cakeID} \rangle$

$A_2 = \mathsf{get}(\mathsf{LayersDesiderata}), \mathsf{get}(\mathsf{DecorationsInfo}), \mathsf{CakesDesiderata.cake} := \mathsf{LayersDesiderata.top} + \text{`\&'} +$
$\mathsf{DecorationsInfo.top} + \text{`on'} + \mathsf{LayersDesiderata.middle} + \text{`\&'} + \mathsf{DecorationsInfo.middle} + \text{`on'} +$
$\mathsf{LayersDesiderata.bottom} + \text{`\&'} + \mathsf{DecorationsInfo.bottom},$
$\mathsf{CakesDesiderata.cakeID} := \mathsf{LayersDesiderata.cakeID}, \mathsf{push}(\mathsf{CakesDesiderata})$

$\tilde{\mathsf{exp}}_4 = \langle \mathsf{DecorationsInfo.top}, \mathsf{DecorationsInfo.middle}, \mathsf{DecorationsInfo.bottom}, \mathsf{LayersDesiderata.cakeID} \rangle$

$A_3 = \mathsf{push}(\mathsf{Cakes})$

$\tilde{\mathsf{t}}_1 = \langle ?\mathsf{Cakes.cake}, ?\mathsf{Cakes.cakeID} \rangle$

$A_4 = \mathsf{Evaluation.result} := \mathsf{compare}(\mathsf{CakesDesiderata}, \mathsf{Cakes})$

*Pastry Chef process* :

$P_p = \mathsf{startRcv}(\mathsf{CakeRequest}:\tilde{\mathsf{t}}_{21}, \mathsf{e}_{21}) \parallel \mathsf{task}(\mathsf{e}_{21}, \mathsf{CheckCakeRequest}, \mathsf{a}, \mathsf{exp}_{21}, \mathsf{A}_{21}, \mathsf{e}_{22}) \parallel$
$\mathsf{mipTask}(\mathsf{e}_{22}, 3, \mathsf{taskSnd}(\mathsf{e}'_{22}, \mathsf{AssignLayers}, \mathsf{a}, \mathsf{exp}_{22}, \mathsf{A}_{22}, \mathsf{LayerRequest}:\tilde{\mathsf{exp}}_{23}, \mathsf{e}'_{23}), \mathsf{c}_5, \mathsf{false}, \mathsf{e}_{23}) \parallel$
$\mathsf{taskRcv}(\mathsf{e}_{23}, \mathsf{ReceiveDecorations}, \mathsf{a}, \mathsf{exp}_{24}, \mathsf{A}_{23}, \mathsf{Decorations}:\tilde{\mathsf{t}}_{22}, \mathsf{e}_{24}) \parallel$
$\mathsf{mipTask}(\mathsf{e}_{24}, 3, \mathsf{taskSnd}(\mathsf{e}'_{24}, \mathsf{AssignDecorations}, \mathsf{a}, \mathsf{exp}_{25}, \mathsf{A}_{24}, \mathsf{Decoration}:\tilde{\mathsf{exp}}_{26}, \mathsf{e}'_{25}), \mathsf{c}_6, \mathsf{false}, \mathsf{e}_{25}) \parallel$
$\mathsf{misTask}(\mathsf{e}_{25}, 3, \mathsf{taskRcv}(\mathsf{e}''_{25}, \mathsf{Receive\&CombineLayers}, \mathsf{a}, \mathsf{exp}_{27}, \mathsf{A}_{25}, \mathsf{Layer}:\tilde{\mathsf{t}}_{23}, \mathsf{e}'_{26}), \mathsf{c}_7, \mathsf{false}, \mathsf{e}_{26}) \parallel$
$\mathsf{taskSnd}(\mathsf{e}_{26}, \mathsf{SendCake}, \mathsf{a}, \mathsf{exp}_{28}, \epsilon, \mathsf{Cake}:\tilde{\mathsf{exp}}_{29}, \mathsf{e}_{27}) \parallel \mathsf{end}(\mathsf{e}_{27})$

*Templates*, *expressions*, *assignments* :

$\tilde{\mathsf{t}}_{21} = \langle ?\mathsf{LayersRequest.top}, ?\mathsf{LayersRequest.middle}, ?\mathsf{LayersRequest.bottom}, ?\mathsf{LayersRequest.cakeID} \rangle$

$A_{21} = \mathsf{LayersPlan.position} := \text{`bottom'}, \mathsf{LayersPlan.layerColor} := \mathsf{LayersRequest.bottom},$
$\mathsf{LayersPlan.cakeID} := \mathsf{LayerRequest.cakeID}, \mathsf{push}(\mathsf{LayersPlan}),$
$\mathsf{LayersPlan.position} := \text{`middle'}, \mathsf{LayersPlan.layerColor} := \mathsf{LayersRequest.middle},$
$\mathsf{LayersPlan.cakeID} := \mathsf{LayerRequest.cakeID}, \mathsf{push}(\mathsf{LayersPlan}),$
$\mathsf{LayersPlan.position} := \text{`top'}, \mathsf{LayersPlan.layerColor} := \mathsf{LayersRequest.top},$
$\mathsf{LayersPlan.cakeID} := \mathsf{LayerRequest.cakeID}, \mathsf{push}(\mathsf{LayersPlan})$

$A_{22} = \mathsf{get}(\mathsf{LayersPlan})$

$\tilde{\mathsf{exp}}_{23} = \langle \mathsf{LayersPlan.layerColor}, \mathsf{LayersPlan.position}, \mathsf{LayersPlan.cakeID} \rangle$

$A_{23} = \mathsf{LayersPlan.position} := \text{`bottom'}, \mathsf{LayersPlan.decorationColor} := \mathsf{DecorationsRequest.bottom},$
$\mathsf{LayersPlan.cakeID} := \mathsf{LayerRequest.cakeID}, \mathsf{push}(\mathsf{LayersPlan}),$
$\mathsf{LayersPlan.position} := \text{`middle'}, \mathsf{LayersPlan.decorationColor} := \mathsf{DecorationsRequest.middle},$
$\mathsf{LayersPlan.cakeID} := \mathsf{LayerRequest.cakeID}, \mathsf{push}(\mathsf{LayersPlan}),$
$\mathsf{LayersPlan.position} := \text{`top'}, \mathsf{LayersPlan.decorationColor} := \mathsf{DecorationsRequest.top},$
$\mathsf{LayersPlan.cakeID} := \mathsf{LayerRequest.cakeID}, \mathsf{push}(\mathsf{LayersPlan})$

$\tilde{\mathsf{t}}_{22} = \langle ?\mathsf{DecorationsRequest.top}, ?\mathsf{DecorationsRequest.middle}, ?\mathsf{DecorationsRequest.bottom}, \mathsf{LayerRequest.cakeID} \rangle$

$A_{24} = \mathsf{get}(\mathsf{LayersPlan})$

$\tilde{\mathsf{exp}}_{26} = \langle \mathsf{LayersPlan.position}, \mathsf{LayersPlan.decorationColor}, \mathsf{LayerRequest.cakeID} \rangle$

$A_{25} = \mathsf{Cake.cake} := \mathsf{addLayer}(\mathsf{DecoratedLayer.layer}, \mathsf{DecoratedLayer.position}),$
$\mathsf{Cake.numLayers} := \mathsf{Cake.numLayers} + 1, \mathsf{Cake.cakeID} := \mathsf{LayersRequest.cakeID}$

$\tilde{\mathsf{t}}_{23} = \langle ?\mathsf{DecoratedLayer.layer}, ?\mathsf{DecoratedLayer.position}, \mathsf{LayerRequest.cakeID} \rangle$

$\mathsf{exp}_{28} = \mathsf{Cake.numLayers} = 3$

$\tilde{\mathsf{exp}}_{29} = \langle \mathsf{Cake.cake}, \mathsf{Cake.cakeID} \rangle$

**Fig. C.26.** Textual representation of the running example (first revision, part 1/2).

*Assistant process* :

$P_a = \mathsf{startRcv}(\mathsf{LayerRequest}:\tilde{\mathsf{t}}_{31}, \mathsf{e}_{31}) \parallel \mathsf{task}(\mathsf{e}_{31}, \mathsf{PrepareLayer}, \mathsf{a}, \mathsf{exp}_{31}, \mathsf{A}_{31}, \mathsf{e}_{32}) \parallel$
$\mathsf{taskRcv}(\mathsf{e}_{32}, \mathsf{ReceiveDecoration}, \mathsf{a}, \mathsf{true}, \epsilon, \mathsf{Decoration}:\tilde{\mathsf{t}}_{32}, \mathsf{e}_{33}) \parallel \mathsf{task}(\mathsf{e}_{33}, \mathsf{ApplyDecoration}, \mathsf{a}, \mathsf{exp}_{32}, \mathsf{A}_{32}, \mathsf{e}_{34}) \parallel$
$\mathsf{taskSnd}(\mathsf{e}_{34}, \mathsf{SendDecoratedLayer}, \mathsf{a}, \mathsf{exp}_{33}, \epsilon, \mathsf{Layer}:\tilde{\mathsf{exp}}_{34}, \mathsf{e}_{35}) \parallel \mathsf{end}(\mathsf{e}_{35})$

*Templates*, *expressions*, *assignments* :

$\tilde{\mathsf{t}}_{31} = \langle ?\mathsf{LayerPlan.layerColor}, ?\mathsf{LayerPlan.position}, ?\mathsf{Layer.id} \rangle$

$A_{31} = \mathsf{Layer.status} := \text{`prepared'}$

$\tilde{\mathsf{t}}_{32} = \langle \mathsf{LayerPlan.position}, ?\mathsf{LayerPlan.decorationColor}, \mathsf{Layer.id} \rangle$

$\mathsf{exp}_{32} = \mathsf{Layer.status} = \text{`prepared'}$

$A_{32} = \mathsf{Layer.status} := \text{`decorated'}, \mathsf{Layer.position} := \mathsf{LayerPlan.position}, \mathsf{Layer.layer} := \mathsf{LayerPlan.layerColor} +$
$\text{`\&'} + \mathsf{LayerPlan.decorationColor}$

$\mathsf{exp}_{33} = \mathsf{Layer.status} = \text{`decorated'}$

$\tilde{\mathsf{exp}}_{34} = \langle \mathsf{Layer.layer}, \mathsf{Layer.position}, \mathsf{Layer.id} \rangle$

**Fig. C.27.** Textual representation of the running example (first revision, part 2/2).

Please cite this article as: F. Corradini, C. Muzi, B. Re et al., Formalising and animating multiple instances in BPMN collaborations, Information Systems (2019) 101459, https://doi.org/10.1016/j.is.2019.101459.

*Overall cake preparation collaboration scenario:*

$\mathsf{miPool}(\mathsf{p}_c, P_c, 2) \parallel \mathsf{miPool}(\mathsf{p}_p, P_p, 2) \parallel \mathsf{miPool}(\mathsf{p}_a, P_a, 6)$

*Customer process* :

$P_c = \mathsf{start}(\mathsf{e}_1, \mathsf{e}_2) \parallel \mathsf{taskSnd}(\mathsf{e}_2, \mathsf{SendCakeRequest}, \mathsf{a}, \mathsf{exp}_1, \mathsf{A}_1, \mathsf{CakeRequest}:\tilde{\mathsf{exp}}_2, \mathsf{e}_3) \parallel$
$\mathsf{taskSnd}(\mathsf{e}_3, \mathsf{SendDecorationsRequest}, \mathsf{a}, \mathsf{exp}_3, \mathsf{A}_2, \mathsf{Decorations}:\tilde{\mathsf{exp}}_4, \mathsf{e}_4) \parallel$
$\mathsf{taskRcv}(\mathsf{e}_4, \mathsf{ReceiveCake}, \mathsf{a}, \mathsf{true}, \epsilon, \mathsf{Cake}:\tilde{\mathsf{t}}_1, \mathsf{e}_5) \parallel$
$\mathsf{xorSplit}(\mathsf{e}_5, \{(\mathsf{e}_6, \mathsf{Cake.cake} \neq \mathsf{Desiderata.cake}), (\mathsf{e}_7, \mathsf{Cake.cake} = \mathsf{Desiderata.cake})\}) \parallel$
$\mathsf{end}(\mathsf{e}_6) \parallel \mathsf{task}(\mathsf{e}_7, \mathsf{Celebrate}, \mathsf{a}, \mathsf{true}, \epsilon, \mathsf{e}_8) \parallel \mathsf{end}(\mathsf{e}_8)$

*Templates, expressions, assignments* :

$A_1 =$ Desiderata.top := LayersInfo.top, Desiderata.middle := LayersInfo.middle,
Desiderata.bottom := LayersInfo.bottom

$\tilde{\mathsf{exp}}_2 = \langle$LayersInfo.top, LayersInfo.middle, LayersInfo.bottom, customerName()$\rangle$

$A_2 =$ Desiderata.cake := Desiderata.top + '&'+DecorationsInfo.top + 'on'+Desiderata.middle
+'&'+DecorationsInfo.middle + 'on'+Desiderata.bottom + '&'+DecorationsInfo.bottom

$\tilde{\mathsf{exp}}_4 = \langle$DecorationsInfo.top, DecorationsInfo.middle, DecorationsInfo.bottom, customerName()$\rangle$

$\tilde{\mathsf{t}}_1 = \langle$?Cake.cake, customerName()$\rangle$

*Pastry Chef process* :

$P_p = \mathsf{startRcv}(\mathsf{CakeRequest}:\tilde{\mathsf{t}}_{21}, \mathsf{e}_{21}) \parallel \mathsf{task}(\mathsf{e}_{21}, \mathsf{CheckCakeRequest}, \mathsf{a}, \mathsf{exp}_{21}, \mathsf{A}_{21}, \mathsf{e}_{22}) \parallel$
$\mathsf{mipTask}(\mathsf{e}_{22}, 3, \mathsf{taskSnd}(\mathsf{e}'_{22}, \mathsf{AssignLayers}, \mathsf{a}, \mathsf{exp}_{22}, \mathsf{A}_{22}, \mathsf{LayerRequest}:\tilde{\mathsf{exp}}_{23}, \mathsf{e}'_{23}), \mathsf{c}_1, \mathsf{false}, \mathsf{e}_{23}) \parallel$
$\mathsf{taskRcv}(\mathsf{e}_{23}, \mathsf{ReceiveDecorations}, \mathsf{a}, \mathsf{exp}_{24}, \mathsf{A}_{23}, \mathsf{Decorations}:\tilde{\mathsf{t}}_{22}, \mathsf{e}_{24}) \parallel$
$\mathsf{mipTask}(\mathsf{e}_{24}, 3, \mathsf{taskSnd}(\mathsf{e}'_{24}, \mathsf{AssignDecorations}, \mathsf{a}, \mathsf{exp}_{25}, \mathsf{A}_{24}, \mathsf{Decoration}:\tilde{\mathsf{exp}}_{26}, \mathsf{e}'_{25}), \mathsf{c}_2, \mathsf{false}, \mathsf{e}_{25}) \parallel$
$\mathsf{misTask}(\mathsf{e}_{25}, 3, \mathsf{taskRcv}(\mathsf{e}''_{25}, \mathsf{Receive\&CombineLayers}, \mathsf{a}, \mathsf{exp}_{27}, \mathsf{A}_{25}, \mathsf{Layer}:\tilde{\mathsf{t}}_{23}, \mathsf{e}'_{26}), \mathsf{c}_3, \mathsf{false}, \mathsf{e}_{26}) \parallel$
$\mathsf{taskSnd}(\mathsf{e}_{26}, \mathsf{SendCake}, \mathsf{a}, \mathsf{exp}_{28}, \epsilon, \mathsf{Cake}:\tilde{\mathsf{exp}}_{29}, \mathsf{e}_{27}) \parallel \mathsf{end}(\mathsf{e}_{27})$

*Templates, expressions, assignments* :

$\tilde{\mathsf{t}}_{21} = \langle$?LayersRequest.top, ?LayersRequest.middle, ?LayersRequest.bottom, ?LayersRequest.customerName$\rangle$

$A_{21} =$ LayersPlan.position := 'bottom', LayersPlan.layerColor := LayersRequest.bottom,
LayersPlan.customerName := LayersRequest.customerName, push(LayersPlan),
LayersPlan.position := 'middle', LayersPlan.layerColor := LayersRequest.middle,
LayersPlan.customerName := LayersRequest.customerName, push(LayersPlan),
LayersPlan.position := 'top', LayersPlan.layerColor := LayersRequest.top,
LayersPlan.customerName := LayersRequest.customerName, push(LayersPlan)

$A_{22} =$ get(LayersPlan)

$\tilde{\mathsf{exp}}_{23} = \langle$LayersPlan.layerColor, LayersPlan.position, LayersPlan.customerName$\rangle$

$A_{23} =$ LayersPlan.position := 'bottom', LayersPlan.decorationColor := DecorationsRequest.bottom,
LayersPlan.customerName := LayersRequest.customerName, push(LayersPlan),
LayersPlan.position := 'middle', LayersPlan.decorationColor := DecorationsRequest.middle,
LayersPlan.customerName := LayersRequest.customerName, push(LayersPlan),
LayersPlan.position := 'top', LayersPlan.decorationColor := DecorationsRequest.top,
LayersPlan.customerName := LayersRequest.customerName, push(LayersPlan)

$\tilde{\mathsf{t}}_{22} = \langle$?DecorationsRequest.top, ?DecorationsRequest.middle, ?DecorationsRequest.bottom, LayersRequest.customerName$\rangle$

$A_{24} =$ get(LayersPlan)

$\tilde{\mathsf{exp}}_{26} = \langle$LayersPlan.position, LayersPlan.decorationColor, LayersPlan.customerName$\rangle$

$A_{25} =$ Cake.cake := addLayer(DecoratedLayer.layer, DecoratedLayer.position),
Cake.numLayers := Cake.numLayers + 1, Cake.cakeID := LayersRequest.customerName

$\tilde{\mathsf{t}}_{23} = \langle$?DecoratedLayer.layer, ?DecoratedLayer.position, LayersRequest.customerName$\rangle$

$\mathsf{exp}_{28} =$ Cake.numLayers = 3

$\tilde{\mathsf{exp}}_{29} = \langle$Cake.cake, Cake.cakeID$\rangle$

*Assistant process* :

$P_a = \mathsf{startRcv}(\mathsf{LayerRequest}:\tilde{\mathsf{t}}_{31}, \mathsf{e}_{31}) \parallel \mathsf{task}(\mathsf{e}_{31}, \mathsf{PrepareLayer}, \mathsf{a}, \mathsf{exp}_{31}, \mathsf{A}_{31}, \mathsf{e}_{32}) \parallel$
$\mathsf{taskRcv}(\mathsf{e}_{32}, \mathsf{ReceiveDecoration}, \mathsf{a}, \mathsf{true}, \epsilon, \mathsf{Decoration}:\tilde{\mathsf{t}}_{32}, \mathsf{e}_{33}) \parallel$
$\mathsf{task}(\mathsf{e}_{33}, \mathsf{ApplyDecoration}, \mathsf{a}, \mathsf{exp}_{32}, \mathsf{A}_{32}, \mathsf{e}_{34}) \parallel$
$\mathsf{taskSnd}(\mathsf{e}_{34}, \mathsf{SendDecoratedLayer}, \mathsf{a}, \mathsf{exp}_{33}, \epsilon, \mathsf{Layer}:\tilde{\mathsf{exp}}_{34}, \mathsf{e}_{35}) \parallel \mathsf{end}(\mathsf{e}_{35})$

*Templates, expressions, assignments* :

$\tilde{\mathsf{t}}_{31} = \langle$?LayerPlan.layerColor, ?LayerPlan.position, ?Layer.id$\rangle$

$A_{31} =$ Layer.status := 'prepared'

$\tilde{\mathsf{t}}_{32} = \langle$LayerPlan.position, ?LayerPlan.decorationColor, Layer.id$\rangle$

$\mathsf{exp}_{32} =$ Layer.status = 'prepared'

$A_{32} =$ Layer.status := 'decorated', Layer.position := LayerPlan.position,
Layer.layer := LayerPlan.layerColor + '&'+LayerPlan.decorationColor

$\mathsf{exp}_{33} =$ Layer.status = 'decorated'

$\tilde{\mathsf{exp}}_{34} = \langle$Layer.layer, Layer.position, Layer.id$\rangle$

**Fig. C.28.** Textual representation of the running example (second revision).

Please cite this article as: F. Corradini, C. Muzi, B. Re et al., Formalising and animating multiple instances in BPMN collaborations, Information Systems (2019) 101459, https://doi.org/10.1016/j.is.2019.101459.

*Overall cake preparation collaboration scenario:*
$\mathsf{miPool}(\mathsf{p}_c, P_c, 2) \parallel \mathsf{miPool}(\mathsf{p}_p, P_p, 2) \parallel \mathsf{miPool}(\mathsf{p}_a, P_a, 6)$

*Customer process* :

$P_c = \mathsf{start}(\mathsf{e}_1, \mathsf{e}_2) \parallel \mathsf{taskSnd}(\mathsf{e}_2, \mathsf{SendCakeRequest}, \mathsf{a}, \mathsf{exp}_1, A_1, \mathsf{CakeRequest}:\tilde{\mathsf{exp}}_2, \mathsf{e}_3) \parallel$
$\mathsf{taskSnd}(\mathsf{e}_3, \mathsf{SendDecorationsRequest}, \mathsf{a}, \mathsf{exp}_3, A_2, \mathsf{Decorations}:\tilde{\mathsf{exp}}_4, \mathsf{e}_4) \parallel$
$\mathsf{taskRcv}(\mathsf{e}_4, \mathsf{ReceiveCake}, \mathsf{a}, \mathsf{true}, \epsilon, \mathsf{Cake}:\tilde{\mathsf{t}}_1, \mathsf{e}_5) \parallel \mathsf{task}(\mathsf{e}_5, \mathsf{JudgeCake}, \mathsf{a}, \mathsf{true}, A_3, \mathsf{e}_6) \parallel$
$\mathsf{task}(\mathsf{e}_6, \mathsf{TakeDecision}, \mathsf{a}, \mathsf{exp}_5, A_4, \mathsf{e}_7) \parallel \mathsf{xorSplit}(\mathsf{e}_7, \{(\mathsf{e}_8, \mathsf{Decision.result} = \mathsf{false}), (\mathsf{e}_9, \mathsf{Decision.result} = \mathsf{true})\}) \parallel$
$\mathsf{end}(\mathsf{e}_8) \parallel \mathsf{task}(\mathsf{e}_9, \mathsf{Celebrate}, \mathsf{a}, \mathsf{true}, \epsilon, \mathsf{e}_{10}) \parallel \mathsf{end}(\mathsf{e}_{10})$

*Templates, expressions, assignments* :

$A_1 = \mathsf{Desiderata.top} := \mathsf{LayersInfo.top}, \mathsf{Desiderata.middle} := \mathsf{LayersInfo.middle},$
$\mathsf{Desiderata.bottom} := \mathsf{LayersInfo.bottom}$
$\tilde{\mathsf{exp}}_2 = \langle \mathsf{LayersInfo.top}, \mathsf{LayersInfo.middle}, \mathsf{LayersInfo.bottom}, \mathsf{customerName}() \rangle$
$A_2 = \mathsf{Desiderata.cake} := \mathsf{Desiderata.top} + \text{`\&'} + \mathsf{DecorationsInfo.top} + \text{`on'} + \mathsf{Desiderata.middle}$
$+\text{`\&'} + \mathsf{DecorationsInfo.middle} + \text{`on'} + \mathsf{Desiderata.bottom} + \text{`\&'} + \mathsf{DecorationsInfo.bottom}$
$\tilde{\mathsf{exp}}_4 = \langle \mathsf{DecorationsInfo.top}, \mathsf{DecorationsInfo.middle}, \mathsf{DecorationsInfo.bottom}, \mathsf{customerName}() \rangle$
$\tilde{\mathsf{t}}_1 = \langle ?\mathsf{Cake.cake}, \mathsf{customerName}() \rangle$
$A_3 = \mathsf{Judgment.evaluation} := \mathsf{Judgment.evaluation} \wedge (\mathsf{Cake.cake} = \mathsf{Cake.desiderata}),$
$\mathsf{Judgment.counter} := \mathsf{Judgment.counter} + 1$
$\mathsf{exp}_5 = \mathsf{Judgment.counter} = 2$
$A_4 = \mathsf{Decision.result} := \mathsf{Judgment.evaluation}$

*Pastry Chef process* :

$P_p = \mathsf{startRcv}(\mathsf{CakeRequest}:\tilde{\mathsf{t}}_{21}, \mathsf{e}_{21}) \parallel \mathsf{task}(\mathsf{e}_{21}, \mathsf{CheckCakeRequest}, \mathsf{a}, \mathsf{exp}_{21}, A_{21}, \mathsf{e}_{22}) \parallel$
$\mathsf{mipTask}(\mathsf{e}_{22}, 3, \mathsf{taskSnd}(\mathsf{e}'_{22}, \mathsf{AssignLayers}, \mathsf{a}, \mathsf{exp}_{22}, A_{22}, \mathsf{LayerRequest}:\tilde{\mathsf{exp}}_{23}, \mathsf{e}'_{23}), \mathsf{c}_1, \mathsf{false}, \mathsf{e}_{23}) \parallel$
$\mathsf{taskRcv}(\mathsf{e}_{23}, \mathsf{ReceiveDecorations}, \mathsf{a}, \mathsf{exp}_{24}, A_{23}, \mathsf{Decorations}:\tilde{\mathsf{t}}_{22}, \mathsf{e}_{24}) \parallel$
$\mathsf{mipTask}(\mathsf{e}_{24}, 3, \mathsf{taskSnd}(\mathsf{e}'_{24}, \mathsf{AssignDecorations}, \mathsf{a}, \mathsf{exp}_{25}, A_{24}, \mathsf{Decoration}:\tilde{\mathsf{exp}}_{26}, \mathsf{e}'_{25}), \mathsf{c}_2, \mathsf{false}, \mathsf{e}_{25}) \parallel$
$\mathsf{misTask}(\mathsf{e}_{25}, 3, \mathsf{taskRcv}(\mathsf{e}''_{25}, \mathsf{Receive\&CombineLayers}, \mathsf{a}, \mathsf{exp}_{27}, A_{25}, \mathsf{Layer}:\tilde{\mathsf{t}}_{23}, \mathsf{e}'_{26}), \mathsf{c}_3, \mathsf{false}, \mathsf{e}_{26}) \parallel$
$\mathsf{taskSnd}(\mathsf{e}_{26}, \mathsf{SendCake}, \mathsf{a}, \mathsf{exp}_{28}, \epsilon, \mathsf{Cake}:\tilde{\mathsf{exp}}_{29}, \mathsf{e}_{27}) \parallel \mathsf{end}(\mathsf{e}_{27})$

*Templates, expressions, assignments* :

$\tilde{\mathsf{t}}_{21} = \langle ?\mathsf{LayersRequest.top}, ?\mathsf{LayersRequest.middle}, ?\mathsf{LayersRequest.bottom}, ?\mathsf{LayersRequest.customerName} \rangle$
$A_{21} = \mathsf{LayersPlan.position} := \text{`bottom'}, \mathsf{LayersPlan.layerColor} := \mathsf{LayersRequest.bottom},$
$\mathsf{LayersPlan.customerName} := \mathsf{LayersRequest.customerName}, \mathsf{push}(\mathsf{LayersPlan}),$
$\mathsf{LayersPlan.position} := \text{`middle'}, \mathsf{LayersPlan.layerColor} := \mathsf{LayersRequest.middle},$
$\mathsf{LayersPlan.customerName} := \mathsf{LayersRequest.customerName}, \mathsf{push}(\mathsf{LayersPlan}),$
$\mathsf{LayersPlan.position} := \text{`top'}, \mathsf{LayersPlan.layerColor} := \mathsf{LayersRequest.top},$
$\mathsf{LayersPlan.customerName} := \mathsf{LayersRequest.customerName}, \mathsf{push}(\mathsf{LayersPlan})$
$A_{22} = \mathsf{get}(\mathsf{LayersPlan})$
$\tilde{\mathsf{exp}}_{23} = \langle \mathsf{LayersPlan.layerColor}, \mathsf{LayersPlan.position}, \mathsf{LayersPlan.customerName} \rangle$
$A_{23} = \mathsf{LayersPlan.position} := \text{`bottom'}, \mathsf{LayersPlan.decorationColor} := \mathsf{DecorationsRequest.bottom},$
$\mathsf{LayersPlan.customerName} := \mathsf{LayersRequest.customerName}, \mathsf{push}(\mathsf{LayersPlan}),$
$\mathsf{LayersPlan.position} := \text{`middle'}, \mathsf{LayersPlan.decorationColor} := \mathsf{DecorationsRequest.middle},$
$\mathsf{LayersPlan.customerName} := \mathsf{LayersRequest.customerName}, \mathsf{push}(\mathsf{LayersPlan}),$
$\mathsf{LayersPlan.position} := \text{`top'}, \mathsf{LayersPlan.decorationColor} := \mathsf{DecorationsRequest.top},$
$\mathsf{LayersPlan.customerName} := \mathsf{LayersRequest.customerName}, \mathsf{push}(\mathsf{LayersPlan})$
$\tilde{\mathsf{t}}_{22} = \langle ?\mathsf{DecorationsRequest.top}, ?\mathsf{DecorationsRequest.middle}, ?\mathsf{DecorationsRequest.bottom}, \mathsf{LayersRequest.customerName} \rangle$
$A_{24} = \mathsf{get}(\mathsf{LayersPlan})$
$\tilde{\mathsf{exp}}_{26} = \langle \mathsf{LayersPlan.position}, \mathsf{LayersPlan.decorationColor}, \mathsf{LayersPlan.customerName} \rangle$
$A_{25} = \mathsf{Cake.cake} := \mathsf{addLayer}(\mathsf{DecoratedLayer.layer}, \mathsf{DecoratedLayer.position}),$
$\mathsf{Cake.numLayers} := \mathsf{Cake.numLayers} + 1, \mathsf{Cake.cakeID} := \mathsf{LayersRequest.customerName}$
$\tilde{\mathsf{t}}_{23} = \langle ?\mathsf{DecoratedLayer.layer}, ?\mathsf{DecoratedLayer.position}, \mathsf{LayersRequest.customerName} \rangle$
$\mathsf{exp}_{28} = \mathsf{Cake.numLayers} = 3$
$\tilde{\mathsf{exp}}_{29} = \langle \mathsf{Cake.cake}, \mathsf{Cake.cakeID} \rangle$

*Assistant process* :

$P_a = \mathsf{startRcv}(\mathsf{LayerRequest}:\tilde{\mathsf{t}}_{31}, \mathsf{e}_{31}) \parallel \mathsf{task}(\mathsf{e}_{31}, \mathsf{PrepareLayer}, \mathsf{a}, \mathsf{exp}_{31}, A_{31}, \mathsf{e}_{32}) \parallel$
$\mathsf{taskRcv}(\mathsf{e}_{32}, \mathsf{ReceiveDecoration}, \mathsf{a}, \mathsf{true}, \epsilon, \mathsf{Decoration}:\tilde{\mathsf{t}}_{32}, \mathsf{e}_{33}) \parallel$
$\mathsf{task}(\mathsf{e}_{33}, \mathsf{ApplyDecoration}, \mathsf{a}, \mathsf{exp}_{32}, A_{32}, \mathsf{e}_{34}) \parallel$
$\mathsf{taskSnd}(\mathsf{e}_{34}, \mathsf{SendDecoratedLayer}, \mathsf{a}, \mathsf{exp}_{33}, \epsilon, \mathsf{Layer}:\tilde{\mathsf{exp}}_{34}, \mathsf{e}_{35}) \parallel \mathsf{end}(\mathsf{e}_{35})$

*Templates, expressions, assignments* :

$\tilde{\mathsf{t}}_{31} = \langle ?\mathsf{LayerPlan.layerColor}, ?\mathsf{LayerPlan.position}, ?\mathsf{Layer.id} \rangle$
$A_{31} = \mathsf{Layer.status} := \text{`prepared'}$
$\tilde{\mathsf{t}}_{32} = \langle \mathsf{LayerPlan.position}, ?\mathsf{LayerPlan.decorationColor}, \mathsf{Layer.id} \rangle$
$\mathsf{exp}_{32} = \mathsf{Layer.status} = \text{`prepared'}$
$A_{32} = \mathsf{Layer.status} := \text{`decorated'}, \mathsf{Layer.position} := \mathsf{LayerPlan.position},$
$\mathsf{Layer.layer} := \mathsf{LayerPlan.layerColor} + \text{`\&'} + \mathsf{LayerPlan.decorationColor}$
$\mathsf{exp}_{33} = \mathsf{Layer.status} = \text{`decorated'}$
$\tilde{\mathsf{exp}}_{34} = \langle \mathsf{Layer.layer}, \mathsf{Layer.position}, \mathsf{Layer.id} \rangle$

**Fig. C.29.** Textual representation of the running example (third revision).

## References

[1] OMG, Business Process Model and Notation (BPMN V 2.0), 2011.

[2] Anna Suchenia, et al., Selected approaches towards taxonomy of business process anomalies, in: Advances in Business ICT, in: SCI, vol. 658, Springer, 2017, pp. 65–85.

[3] Flavio Corradini, Andrea Polini, Barbara Re, Francesco Tiezzi, An operational semantics of BPMN collaboration, in: FACS, in: LNCS, vol. 9539, Springer, 2016, pp. 161–180.

[4] Flavio Corradini, Chiara Muzi, Barbara Re, Lorenzo Rossi, Francesco Tiezzi, Global vs. local semantics of BPMN 2.0 or-join, in: SOFSEM, in: LNCS, vol. 10706, Springer, 2018, pp. 321–336.

[5] Flavio Corradini, Fabrizio Fornari, Andrea Polini, Barbara Re, Francesco

Tiezzi, A formal approach to modeling and verification of business process collaborations, Sci. Comput. Program. 166 (2018) 35–70.
[6] Remco M. Dijkman, Marlon Dumas, Chun Ouyang, Semantics and analysis of business process models in BPMN, Inf. Softw. Technol. 50 (12) (2008) 1281–1294.
[7] Gero Decker, Remco Dijkman, Marlon Dumas, Luciano García-Bañuelos, Transforming BPMN diagrams into YAWL nets, in: BPM, in: LNCS, vol. 5240, Springer, 2008, pp. 386–389.
[8] Peter Y.H. Wong, Jeremy Gibbons, A process semantics for BPMN, in: Formal Methods and Soft. Eng., in: LNCS, vol. 5256, Springer, 2008, pp. 355–374.
[9] Egon Börger, Bernhard Thalheim, A method for verifiable and validatable business process modeling, in: Advances in Software Engineering, in: LNCS, vol. 5316, Springer, 2008, pp. 59–115.
[10] Pieter Van Gorp, Remco Dijkman, A visual token-based formalization of BPMN 2.0 based on in-place transformations, Inf. Softw. Technol. 55 (2) (2013) 365–394.
[11] Andreas Hermann, Hendrik Scholta, Sebastian Bräuer, Jörg Becker, Collaborative business process management - a literature-based analysis of methods for supporting model understandability, in: Towards Thought Leadership in Digital Transformation. Wirtschaftsinformatik, 2017.
[12] Jörg Becker, Martin Kugeler, Michael Rosemann, Process Management: A Guide for the Design of Business Processes, Springer Science & Business Media, 2013.
[13] Romain Emens, Irene T.P. Vanderfeesten, Hajo A. Reijers, The dynamic visualization of business process models: a prototype and evaluation, in: BPM, in: LNBIP, vol. 256, Springer, 2016, pp. 559–570.
[14] Flavio Corradini, Chiara Muzi, Barbara Re, Lorenzo Rossi, Francesco Tiezzi, Animating multiple instances in BPMN collaborations: from formal semantics to tool support, in: BPM, in: LNCS, vol. 11080, Springer, 2018, pp. 83–101.
[15] Marlon Dumas, Marcello La Rosa, Jan Mendling, Hajo A. Reijers, et al., Fundamentals of Business Process Management, vol. 1, Springer, 2013.
[16] Rosario Pugliese, Francesco Tiezzi, A calculus for orchestration of web services, J. Appl. Log. 10 (1) (2012) 2–31.
[17] Philipp Fromme, Sebastian Warnke, Patrick Dehn, Bpmn-js token simulation, 2017, https://github.com/bpmn-io/bpmn-js-token-simulation.
[18] Flavio Corradini, Chiara Muzi, Barbara Re, Francesco Tiezzi, Lorenzo Rossi, MIDA: multiple instances and data animator, in: Dissertation Award, Demonstration, and Industrial Track at BPM 2018, CEUR Workshop Proceedings, vol. 2196, 2018, pp. 86–90.
[19] Andreas Meyer, Luise Pufahl, Dirk Fahland, Mathias Weske, Modeling and enacting complex data dependencies in business processes, in: BPM, in: LNCS, vol. 8094, Springer, 2013, pp. 171–186.
[20] Andreas Meyer, et al., Data perspective in process choreographies: modeling and execution, Techn. Ber. BPM Center Report BPM-13-29, 2013.
[21] Ahmed Kheldoun, Kamel Barkaoui, Malika Ioualalen, Formal verification of complex business processes based on high-level petri nets, Inform. Sci. 385–386 (2017) 39–54.
[22] Nissreen A.S. El-Saber, CMMI-CM compliance checking of formal BPMN models using Maude (Ph.D. thesis), Department of Computer Science, 2015.
[23] Petia Wohed, Wil M.P. van der Aalst, Marlon Dumas, Arthur H.M. ter Hofstede, Nick Russell, Pattern-based analysis of UML activity diagrams, Research school for operations management and logistics, Eindhoven, 2004.
[24] Wil M.P. Van Der Aalst, Arthur H.M. Ter Hofstede, YAWL: yet another workflow language, Inform. Syst. 30 (4) (2005) 245–275.
[25] María Teresa Gómez López, et al., Guiding the creation of choreographed processes with multiple instances based on data models, in: BPMWorkshops, in: LNBIP, vol. 281, 2016, pp. 239–251.
[26] David Knuplesch, Rüdiger Pryss, Manfred Reichert, Data-aware interaction in distributed and collaborative workflows: modeling, semantics, correctness, in: CollaborateCom, IEEE, 2012, pp. 223–232.
[27] Michael Hahn, Uwe Breitenbücher, Oliver Kopp, Frank Leymann, Modeling and execution of data-aware choreographies: an overview, Comput. Sci. Res. Dev. (2017) 1–12.
[28] OASIS, Web services business process execution language version 2.0, Technical report, 2007.
[29] Hajo A. Reijers, Irene Vanderfeesten, Marijn G.A. Plomp, Pieter Van Gorp, Dirk Fahland, Wim L.M. van der Crommert, H. Daniel Diaz Garcia, Evaluating data-centric process approaches: does the human factor factor in? Softw. Syst. Model. 16 (3) (2017) 649–662.
[30] Diego Calvanese, Giuseppe De Giacomo, Marco Montali, Foundations of data-aware process analysis: a database theory perspective, in: Proceedings of the 32Nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, in: PODS '13, ACM, New York, NY, USA, 2013, pp. 1–12.
[31] Richard Hull, Jianwen Su, Roman Vaculin, Data management perspectives on business process management: tutorial overview, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, in: SIGMOD'13, ACM, New York, NY, USA, 2013, pp. 943–948.
[32] Andreas Meyer, Mathias Weske, Activity-centric and artifact-centric process model roundtrip, in: Business Process Management Workshops, Springer, 2014, pp. 167–181.
[33] Thomas Allweyer, Stefan Schweitzer, A tool for animating BPMN token flow, in: Int. Workshop on BPMN, in: LNBIP, vol. 125, Springer, 2012, pp. 98–106.
[34] Banu Aysolmaz, PRIME process animation, http://prime.cs.vu.nl/.
[35] Signavio GmbH, Signavio, 2018, http://www.signavio.com/.
[36] Visual Paradigm, Business process design with powerful BPMN software, https://www.visual-paradigm.com/features/bpmn-diagram-and-tools/.
[37] Mathias Weske, Business Process Management, Springer, 2007.
[38] OMG, Decision model and notation (DMN V. 1.1), 2016.
[39] Michele Boreale, Rocco De Nicola, A symbolic semantics for the pi-calculus, Inform. and Comput. 126 (1) (1996) 34–52.

Please cite this article as: F. Corradini, C. Muzi, B. Re et al., Formalising and animating multiple instances in BPMN collaborations, Information Systems (2019) 101459, https://doi.org/10.1016/j.is.2019.101459.