Science of Computer Programming 166 (2018) 35–70

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A formal approach to modeling and verification of business process collaborations

Flavio Corradini, Fabrizio Fornari, Andrea Polini, Barbara Re *, Francesco Tiezzi

University of Camerino, School of Science and Technology, Italy

ARTICLE INFO

Article history:
Received 6 February 2017
Received in revised form 6 May 2018
Accepted 24 May 2018
Available online 7 June 2018

Keywords:
Business process modeling
BPMN collaboration
Operational semantics
Maude
Verification

ABSTRACT

In the last years we are observing a growing interest in verification of business process models that, despite their lack of formal characterization, are widely adopted in industry and academia. To this aim, a formalization of the execution semantics of business process modeling languages is essential. In this paper, we focus on the OMG standard BPMN 2.0. Specifically, we provide a direct formalization of its semantics in terms of Labeled Transition Systems. This approach permits to avoid possible miss-interpretations, due to the usage of the natural language in the standard specification, and to overcome issues due to the mapping of BPMN to other formal languages, which are equipped with their own semantics. Our operational semantics is given for a relevant subset of BPMN elements focusing on the capability to model collaborations among organizations via message exchange. One of its distinctive aspects is the suitability to model business processes with arbitrary topology. This allows designers to freely specify their processes according to the reality, without the limitation of defining well-structured models. The provided formalization is also implemented by exploiting the capabilities of Maude. This implementation takes a collaboration model as an input and, explores all the model executions. By relying on it, automatic verification of properties related to collaborations has been carried out via the Maude model checker. We illustrate the benefits of our approach by means of a simple, yet realistic, running example concerning a travel booking scenario.

1. Introduction

For some years now, complex organizations recognized the importance of having tools to model the different and interrelated aspects governing their behavior. Among the different perspectives, used to model organizations, and summarized for instance in the Zachman's framework [1], particularly relevant is certainly the description of how the activities are structured in order to reach the organization objectives. This point of view is generally reflected in a Business Process (BP) model that is characterized as "*a collection of related and structured activities undertaken by one or more organizations in order to pursue some particular goal. [...] BPs are often interrelated since the execution of a BP often results in the activation of related BPs within the same or other organizations*" [2].

Different languages and graphical notations have been proposed to represent BP models with differences in the level of formality used to define the notation elements. BPMN 2.0,[1] which has been standardized by OMG [3], is currently acquiring

* Corresponding author.
E-mail address: barbara.re@unicam.it (B. Re).

[1] We use BPMN or BPMN 2.0 interchangeably to refer to version 2.0 of the notation.

https://doi.org/10.1016/j.scico.2018.05.008
0167-6423/

a clear predominance among the various proposal, due to the intuitive graphical notation, the acceptance by industry and academia, and the support provided by a wide spectrum of modeling tools.[2]

BPMN success comes from its versatility and capability to represent BPs for different purposes. The notation acquired, at first, acceptance among business analysts and operators, who use it to design BP models and mainly for communicative purposes. Successively, it has been more and more adopted by IT specialists to lead the development and settlement of IT systems supporting the execution of a specified BP model. This shift in the usage of the notation is particularly relevant and poses the basis for our work. Indeed the OMG standard does not provide a precise definition for the semantics of the notation. The lack of a precise semantics may not represent a big issue when the notation is used just for communicative purposes. Instead its adoption for shaping IT systems, and even more to apply model driven approaches to code generation, does require the definition of a precise semantics enabling also the introduction of formal verification strategies.

Among the various characteristics of BPMN particularly interesting is the possibility to model *collaborations* involving different organizations, exchanging messages and cooperating to reach a shared objective. Collaboration diagrams are the focus of our work. Such diagrams contain enough information to assess the alignment of participants behavior with respect to the message flow, so to permit a successful cooperation. In this case the definition of a precise semantics has probably even more important consequences, given that collaborations among organizations are more and more mediated through IT systems, according for instance to the Service-Oriented Computing paradigm. In fact, when a modeling notation is used in a homogeneous context, such as a single organization, the precise definition of the meaning of the various elements constituting the notation can be sometime overlooked. Nevertheless mutual understanding is still possible thanks to the direct communications among the involved stakeholders, and from the emergence of established and accepted practices. This cannot be the case when two or more organizations are involved. In fact, in order to correctly collaborate, different organizations have to share the same understanding of the communication flow, in particular when this has to be supported by software systems.

In defining the notation, OMG did not intend to provide a rigorous semantics for the various graphical elements; instead the meaning is given using natural language descriptions. However, the use of formal tools to define the semantics of the various elements, and hence of a BP model, is required to enable automatic analysis activities. This aspect seems to be even more relevant when organizations get in contact with each other and they need to analyze the impact of collaborative actions. Consider for instance the merging of two companies in which there is not a common understanding of the meaning and effect of the activities within a process. In this case a precise semantics of the collaboration model is crucial to favor a more seamless integration, by also enabling analysis activities aimed at discovering possible flaws in the collaboration. In fact, a BPMN model is a complex artefact that can include many errors difficult to spot by humans. To tackle this issue we propose a verification approach able to spot both errors due to internal behavior of a participating process, and errors due to interaction among the participants (e.g., protocol mismatch).

In the last years, a relevant effort has been devoted by the research community to provide a formal semantics to the BPMN notation (we refer to Section 7 for a wide overview of major contributions on the subject). In this paper, we intend to contribute to such a research effort aiming at providing a precise characterization of a subset of BPMN elements largely used in practice. In selecting the BPMN elements to be included in our formalization, we follow a pragmatic approach. On the one hand, to keep the considered fragment of the language manageable (to have, e.g., a simpler formal semantics), we only retain the core features of BPMN. However, even if we focus on a restricted number of elements, we do not consider such design choice a major limitation of the work. Indeed, despite the BPMN specification is quite wide, only less than 20% of its vocabulary is used regularly in designing BP models [4]. On the other hand, we deal with elements concerning collaborations, which are overlooked by other formalization proposals, as they typically focus only on process elements. Our special emphasis on communication aspects within collaboration diagrams is mainly motivated by the need of achieving inter-organizational correctness, which is still a challenge [5]. Notably, the approach we propose here does not aim at checking that a participant is able to enter a prescriptive interaction scenario (generally referred as a choreography). Instead, it intends to support the checking that a collection of processes are actually able to correctly interact when integrated all-together (e.g., absence of deadlock situations due to lack of expected messages, or capability of processing all produced messages).

Our formalization also includes the management of termination end events, used as opportunities to quickly abort processes. This feature is usually not supported in other approaches, especially those based on Petri Nets [6]. Indeed, while for the basic BPMN modeling elements the encoding in Petri Nets is rather straightforward, defining an encoding for terminating events is quite difficult. This is due to the inherent complexity of managing non-local propagation of tokens in Petri Nets, which instead is supported by our semantics.

Differently from other proposals, we do not impose any syntactical restriction on the usage of the modeling notation, such as *well-structuredness* (which, roughly, imposes gateways in a process to form single-entry–single-exit fragments). Even though our semantics works and enables analysis for well-structured models, we refer here to process models with an arbitrary topology. Well aware of costs and benefits of the structuredness in process modeling [7], we aim at removing such restriction to allow more flexibility for BP modelers. Indeed, well-structuredness per se does not guarantee that models are correct (e.g., sound [8]), but it is just a way to contribute to solve some modeling issues [9]. Structuredness restrictions,

[2] BPMN is currently supported by more than 70 tools (see http://www.bpmn.org for a detailed list).

in fact, may be not easily applicable by all model designers or may undermine their modeling freedom. Indeed, on the one hand, designers with limited modeling experiences are prone to model 'spaghetti' processes. On the other hand, more expert designers should be free to express their creativity in modeling the process according to the reality they feel [10]. In addition, it is well known that not all process models with an arbitrary topology can be transformed into equivalent well-structured processes [11,12].

The presence of unstructured models in practice is confirmed by reference documents and some empirical investigation we did. In the *BPMN by example* document provided by OMG, including process models that were collected from the IBM Company, most of the models follow an arbitrary topology [13]. A similar scenario emerges when other large model collections are considered. In particular, we analyzed models made available by the BPM Academic Initiative,[3] comprising more than ten thousands models of various process modeling languages [14]. We selected models specified in BPMN; the resulting dataset turns out to be particularly suited to investigate modeling practices, due to its heterogeneity in terms of contributing modelers [15]. Overall, the analysis we performed on the dataset shows that defining unstructured models is a common practice (see Section 2 for additional details).

More specifically, the contribution of our paper is a novel formalization that provides an operational semantics to BPMN in the SOS style [16] by relying on the notion of Labeled Transition System (LTS). The major benefits of our semantics are as follows:

- it is a direct semantics, given in terms of features and constructs of BPMN, rather than in terms of their low-level encoding into another formalism (equipped with its own syntax and semantics) as in most proposals in the literature (see Section 7);
- besides core elements, such as tasks, gateways, etc., it takes into account collaborations, message exchange and termination events, which are overlooked by other formalizations;
- it has a multi-layer structure, which ensures to decouple the process behavior and the semantics of the interactions at collaboration level;
- it is suitable to model business processes with arbitrary topology, as well as well-structured ones.

An additional contribution of the paper is a Maude[4] implementation of our formalization. It permits, on the one hand, to concretely validate our theoretical definitions and, on the other hand, to enable the verification of properties of interest for BPMN collaborations by exploiting the LTL model checker of Maude. Even if we do not propose a novel verification technique, the presentation of the enabled verification approach is essential for giving evidence of the benefits and potentialities of the proposed formalization.

The rest of the paper is organized as follows. Section 2 provides an overview on BPMN 2.0, proposes a BPMN collaboration used as running example, presents relevant properties for BPMN collaboration models and reports empirical results on the level of structureness of models available in a public repository. Sections 3 and 4 introduce the BPMN syntax and operational semantics we propose. Section 5 illustrates our verification approach, while Section 6 presents the implementation of the semantics in Maude. Section 7 presents a detailed comparison of our approach with the related ones available in the literature. Finally, Section 8 closes the paper with some remarks and opportunities for future work.

2. An overview of BPMN

In this section we introduce BPMN 2.0, probably the most used language for business process modeling. Specifically, we first provide some basic notions on BPMN collaboration diagrams. Then, we present an airline collaboration scenario, used throughout the paper as a running example. Finally, we illustrate the results of our investigation on the level of well-structuredness in a public repository of business process models.

2.1. Basic notions on BPMN

The focus of this section is not a complete presentation of the standard, but a discussion of the main concepts of BPMN [3] we use in the following. These concepts are briefly described below and reported in Fig. 1.

- **Pools** are used to represent participants or organizations involved in the collaboration, and include details on internal process specifications and related elements. Pools are drawn as rectangles.
- **Tasks** are used to represent specific works to perform within a process. Tasks are drawn as rectangles with rounded corners.
- **Connecting edges** are used to connect process elements in the same or different pools. *Sequence Edge* is a solid connector used to specify the internal flow of the process, thus ordering elements in the same pool, while *Message Edge* is a dashed connector used to visualize communication flows between organizations.[5]

[3] http://bpmai.org/.

[4] http://maude.cs.illinois.edu.

[5] As a matter of terminology, Sequence Edge and Message Edge are referred in the BPMN specification as Sequence Flow and Message Flow, respectively.

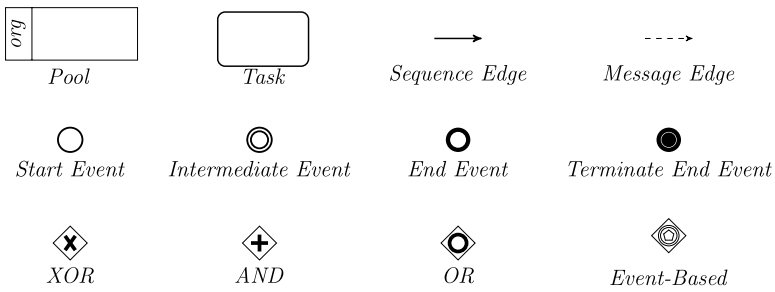


Fig. 1. Considered BPMN 2.0 elements.

- **Events** are used to represent something that can happen. An event can be a *Start Event* representing the point from which the process starts, an *Intermediate Event* representing something that happens during process execution, or an *End Event* representing the process termination. Events are drawn as circles. When an event is source or target of a message edge, it is called *Message Event*.[6] According to the different kinds of message edge connections, we can observe the following situations.
  - *Start Message Event* is a start event with an incoming message edge; the event element catches a message and starts a process.
  - *Throw Intermediate Event* is an intermediate event with an outgoing message edge; the event element sends a message.
  - *Catch Intermediate Event* is an intermediate event with an incoming message edge; the event element receives a message.
  - *End Message Event* is an end event with an outgoing message edge; the event element sends a message and ends the process.

  We also refer to a particular type of end event, the *Terminate End Event*, displayed by a thick circle with a darkened circle inside; it stops and aborts the running process.
- **Gateways** are used to manage the flow of a process both for parallel activities and choices. Gateways are drawn as diamonds and act as either join nodes (merging incoming sequence edges) or split nodes (forking into outgoing sequence edges). Different types of gateways are available.
  - A *XOR gateway* gives the possibility to describe choices. In particular, a XOR-split gateway is used after a decision to fork the flow into branches. When executed, it activates exactly one outgoing edge. A XOR-join gateway acts as a pass-through, meaning that it is activated each time the gateway is reached. A XOR gateway is drawn with a diamond marked with the symbol "$\times$".
  - An *AND gateway* enables parallel execution flows. An AND-split gateway is used to model the parallel execution of two or more branches, as all outgoing sequence edges are activated simultaneously. An AND-join gateway synchronizes the execution of two or more parallel branches, as it waits for all incoming sequence edges to complete before triggering the outgoing flow. An AND gateway is drawn with a diamond marked with the symbol "$+$".
  - An *OR gateway* gives the possibility to select an arbitrary number of (parallel) flows. In fact, an OR-split gateway is similar to the XOR-split one, but its outgoing branches do not need to be mutually exclusive. An OR-join gateway synchronizes the execution of two or more parallel branches, as it waits for all *active* incoming branches to complete before triggering the outgoing flow. An OR gateway is drawn with a diamond marked with the symbol "$\bigcirc$".
  - An *Event-Based gateway* is similar to the XOR-split gateway, but its outgoing branches activation depends on taking place of catching events. Basically, such events are in a race condition, where the first event that is triggered wins and disables the other ones. An event-based gateway is drawn with a diamond marked with the symbol "$\pentagon$" double rounded.

  Notably, even if XOR and OR splitting gateways may have guard conditions in their outgoing sequence edges, in this work we do not consider such possibilities. Indeed, conditions have a significant role only when actual input values are taken into account, while our aim is to enable the verification of processes as a whole, i.e. considering all possible flows and not only those triggered by specific input values.

It is worth noting that we focus on the control flow and communication aspects of business processes. This is mainly motivated by the need of keeping the semantics of the considered part of the specification language rigorous but still manageable. Therefore, as already mentioned in the Introduction, based on practical reasons [4] we intentionally left out other aspects and constructs of BPMN, such as timed events, data objects, sub-processing, error handling, and multiple instances. Moreover, in order to simplify our formal treatment, like in [6] and in accordance with the guidelines in [17,18],

[6] In the standard [3], message events are represented by circles enclosing an envelope, which is white in case of catching events and black in case of throwing ones. However, the information provided by the presence and the color of the envelope is redundant, as the type of the event is made clear by the possible connection with a message edge. Therefore, to keep the notation clean, which is useful for our formalization purpose, we omit the envelope from message events without loss of clear meaning.

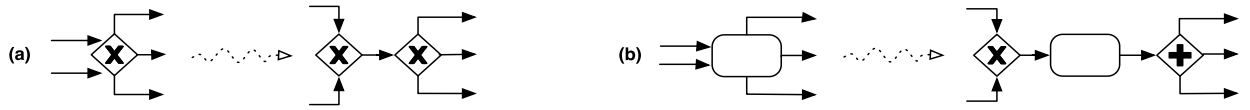


Fig. 2. Examples of transformations: (a) mixed gateway and (b) task with multiple edges.

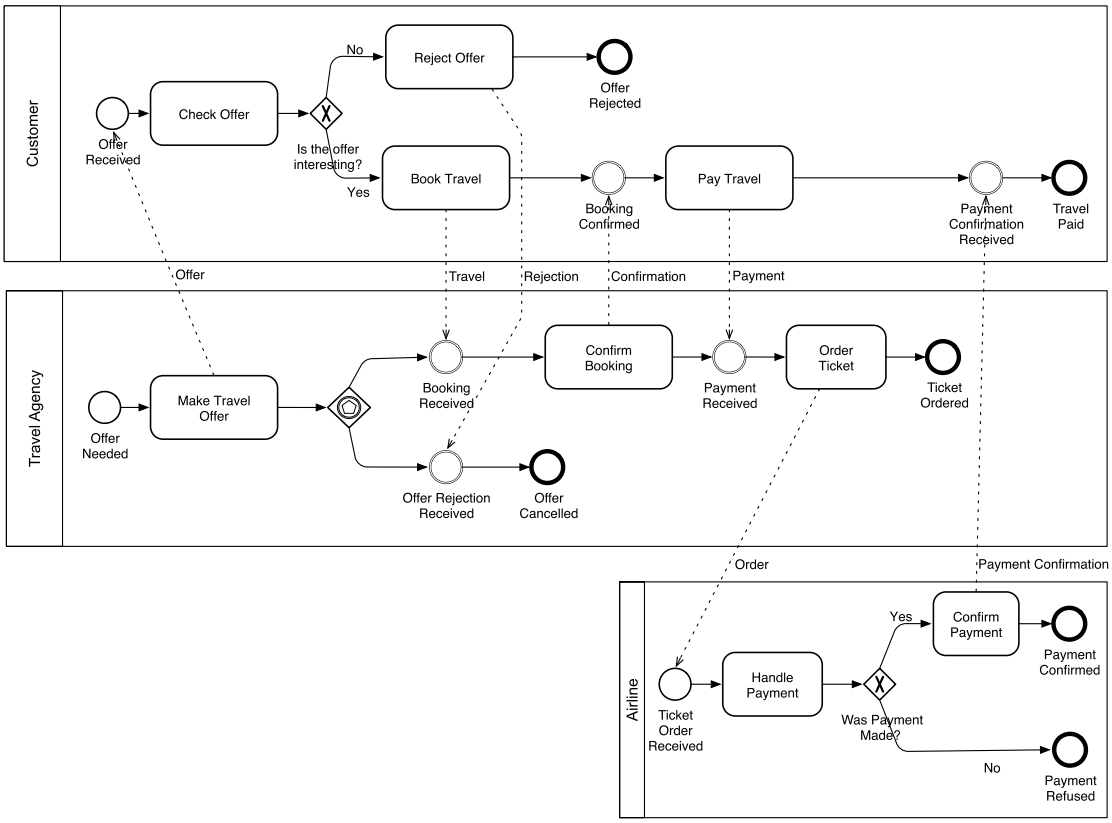


Fig. 3. Airline collaboration example (source [19, p. 115]).

we assume that processes do not contain mixed gateways and tasks with multiple edges. This is not a limitation, as these elements can be safely transformed in others with exactly the same meaning; examples of such transformations are shown in Fig. 2.

2.2. An airline collaboration scenario

We introduce here a BPMN collaboration specification, borrowed from [19], used throughout the paper as a running example.

Fig. 3 shows a collaboration that combines the activities of a Travel Agency, a Customer, and an Airline reservation system. The collaboration allows these participants to interact in the market in order to complete a commercial transaction related to the booking of a travel. After the Travel Agency makes a travel offer, it proposes such offer to the Customer, by sending an offer message. The Customer evaluates the received offer, and takes a decision; this is represented in figure by means of a XOR gateway. According to this decision, one of the two exclusive paths outgoing the gateway is activated. The upper path is activated if the customer rejects the offer; a rejection message is sent to the travel agency and the execution of the customer process terminates. In this case also the Travel Agency terminates its execution. The lower path is activated if the Customer accepts the offer; thus, the Customer books the travel, by sending the travel message to the Travel Agency, and waits for the confirmation message. Once the travel is confirmed, the customer pays the travel and sends the payment message to the Travel Agency. Finally, the customer waits for the payment confirmation message before terminating. This message exchange is supported by the behaviors of the Travel Agency and the Airline. In particular, the Travel Agency, after the offer is sent, waits for the decision of the customer. This is represented by means of an event-based gateway. If the customer rejects the offer, the Travel Agency cancels it as soon as the rejection message is received. On the other hand, if the customer accepts the offer, the Travel Agency receives the travel message and, hence, confirms the booking by sending

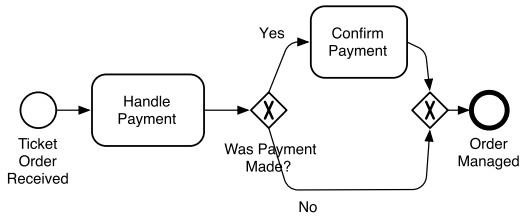


Fig. 4. A well-structured version of the Airline Process.

the confirmation message. Then, as soon as the payment message is received, before terminating the travel agency activates the airline to order the flight tickets, by sending the order message. The Airline proceeds by handling the payment and, according to this, by activating one of two exclusive paths by means of a XOR gateway. The upper path is activated if the payment is confirmed; this leads to the sending of the payment confirmation message to the Customer and, then, to the successful termination of the collaboration. The lower path is activated if the payment is not properly made; in this case the airline immediately terminates thus refusing the ticket order.

2.3. Properties of collaboration models

Properties of interest for BPMN collaboration models may relate both to the internal characteristics of a single process in a collaboration, and to the whole collaboration itself. This holds both for generic properties that are well-established in the business process domain, such as *safeness* [9], and *soundness* [20], and for ad-hoc properties specifically defined for given application scenarios.

Informally, safeness refers to the occurrence of no more than one token at the same time along the same sequence edge of a process during its execution. Instead, soundness typically includes: a process can always complete, once started (*Option to Complete*); and there exists no running or enabled activity for the process when it completes (*Proper Completion*). Safeness and soundness properties naturally extend to process collaborations, requiring that the processes of all involved organizations satisfy them considering the overall collaboration execution.

Concerning ad-hoc properties, instead, in our running scenario for example we can check, at process level, if after completing a given task (say Handle Payment of the Airline process) a related one (say Confirm Payment) can eventually complete. At collaboration level, we can check, e.g., if in all cases the three processes involved in the airline collaboration successfully communicate with each other. This means that the processes are capable to abide by the protocol prescribed by the collaboration.

More details on the analyzed properties are provided in Section 5.

2.4. On the level of well-structuredness in BPMN models

BPMN, as well as most business process modeling notations, allows models to have arbitrary topology. However, to avoid undesired behaviors, modeling guidelines suggest to use structured building blocks [18], thus obtaining *well-structured* process model [21]. Roughly, in a well-structured process, for every split gateway there is a corresponding join gateway such that the fragment of the model between the split and the join forms a single-entry–single-exit process component. As an example, the process in Fig. 4 is the well-structured version of the unstructured process for the Airline organization in Fig. 3.

To investigate the level of structuredness in modeling practice, we have analyzed the BPMN 2.0 collaboration models made available by the BPM Academic Initiative. Starting from the raw dataset including at the time of writing 16.032 models, to consider those with a certain level of quality, we have filtered the latest revision of the models with 100% of connectedness.[7] This results on 7.541 models suitable for our investigation. Then, using the PromniCAT research platform,[8] in the obtained dataset we have automatically selected those models that are well-structured (WS).

Overall, 75% of models are WS. Anyway, more interesting is the trend of the number of well-structured models with respect to their size. Results of this study are reported in Table 1. The trend shows that the level of structuredness decreases as the model size increases. More specifically, let us consider the results concerning classes of models of increasing dimensions. We can observe that the class with models including less than 10 elements contains the highest number of WS models. However, this class is not particularly relevant to our analysis, since it does not include specifications of particularly complex collaborations. The class with size 10–19 is still not very relevant; anyway, even if it contains small models, the percentage of those that are non-WS (28%) is not insignificant. This makes evident that producing models with arbitrary topology is relatively common in practice. Increasing the size and considering the class of models with 20–29 elements, including 943 models, the percentage of non-WS models increases as well, to reach almost the 50% of the total number. The

[7] Connectedness of a model measures the size of the largest connected graph within the model with respect to the size of the overall model, where the model size is the number of its nodes. We do not consider models with connectedness lower than 100%, as they typically are uncompleted models.

[8] https://github.com/tobiashoppe/promnicat.

Table 1
Analysis of the level of structuredness in the dataset (7541 models): result data.

Size	Dataset models	WS models	Non-WS models	% of non-WS models
0–9	3.350	3.133	217	6%
10–19	2.663	1.917	746	28%
20–29	943	484	459	49%
30–39	351	112	239	68%
40–49	109	25	84	77%
50–59	43	12	31	72%
60–69	33	0	33	100%
70–79	11	4	7	64%
80–89	17	3	14	82%
90–99	12	0	12	100%
100–199	9	0	9	100%

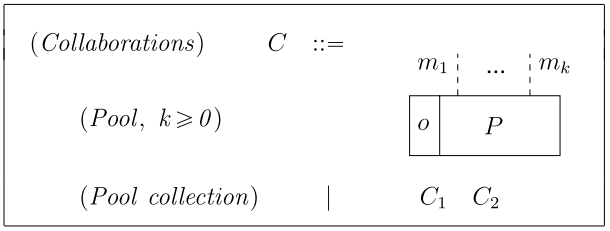


Fig. 5. BPMN syntax: collaborations.

classes that surely cannot be neglected, as they are suitable to model realistic collaboration scenarios, are those with size 30–39, 40–49 and 50–59 including 351, 109 and 43 models, respectively. Overall they include 503 models, 354 out of which are non-WS (around 70%). Finally, in the rest of the classes, the percentage of unstructured models increases up to 100%, as one may expect considering their size. Overall they include 82 models, and only 7 of them are WS.

To sum up, the data discussed above show that in practice BPMN models are often unstructured, especially when they intend to model complex real-world scenarios. These findings make evident that analysis techniques for unstructured processes are highly needed in reality.

3. BNF syntax

The syntax of BPMN 2.0 is given in [3] by a metamodel in classical UML-style. In this section we provide an alternative syntax, in BNF-style, that is more suitable for defining a formal operational semantics.

The syntax is defined by grammar productions of the form $N ::= A_1 \mid \ldots \mid A_n$, where N is a non-terminal symbol and alternatives $A_1, \ldots, A_n$ are compositions of terminal and non-terminal symbols. In particular, in the proposed grammar the non-terminal symbols are C and P, representing *Collaborations* and *Processes* respectively, while the terminal symbols are the typical graphical elements of a BPMN model, i.e. pools, events, tasks, gateways, and edges.

Intuitively, a BPMN collaboration model is rendered in our syntax as a collection of pools (Fig. 5), where message edges can connect different pools. Each pool contains a process, defined as a collection of nodes (Fig. 6), each one with incoming and/or outgoing sequence edges. Such nodes are events, tasks and gateways. We identify *collaborations* (resp. *processes*) up to commutativity and associativity of *pool* (resp. *node*) *collection*. Thus, e.g., $C_1 \;\; C_2$ and $C_2 \;\; C_1$ are identified as the same collaboration due to the commutativity property, while $P_1 \;\; (P_2 \;\; P_3)$ and $(P_1 \;\; P_2) \;\; P_3$ are identified as the same process due to the associativity property.

To obtain a compositional definition, each (message/sequence) edge is divided in two parts: the outgoing one from the source and the incoming one into the target. In fact, a term of the syntax can be straightforwardly obtained from a BPMN model by decomposing the collaboration in a collection of pools, processes in collection of nodes, and edges in two parts.

We use the following disjoint sets of names: the set of *organization* names (ranged over by o), the set of *message* names (ranged over by m), the set of *sequence edge* names (ranged over by e), and the set of *task* names (ranged over by t). As a matter of notation, we use edges of the form $\overset{m}{\text{- - - -}}$ to denote message edges of the form $\overset{m}{\text{- - - -}}\!\!\rightarrow$ either incoming into or outgoing from pools/nodes.

We only consider specifications that are *well-defined*, in the sense that they comply with the following four syntactic constraints:

- distinct pools have different organization names;
- in a collaboration, for each message edge labeled by m outgoing from a pool, there exists only one corresponding message edge labeled by m incoming into another pool, and vice versa;

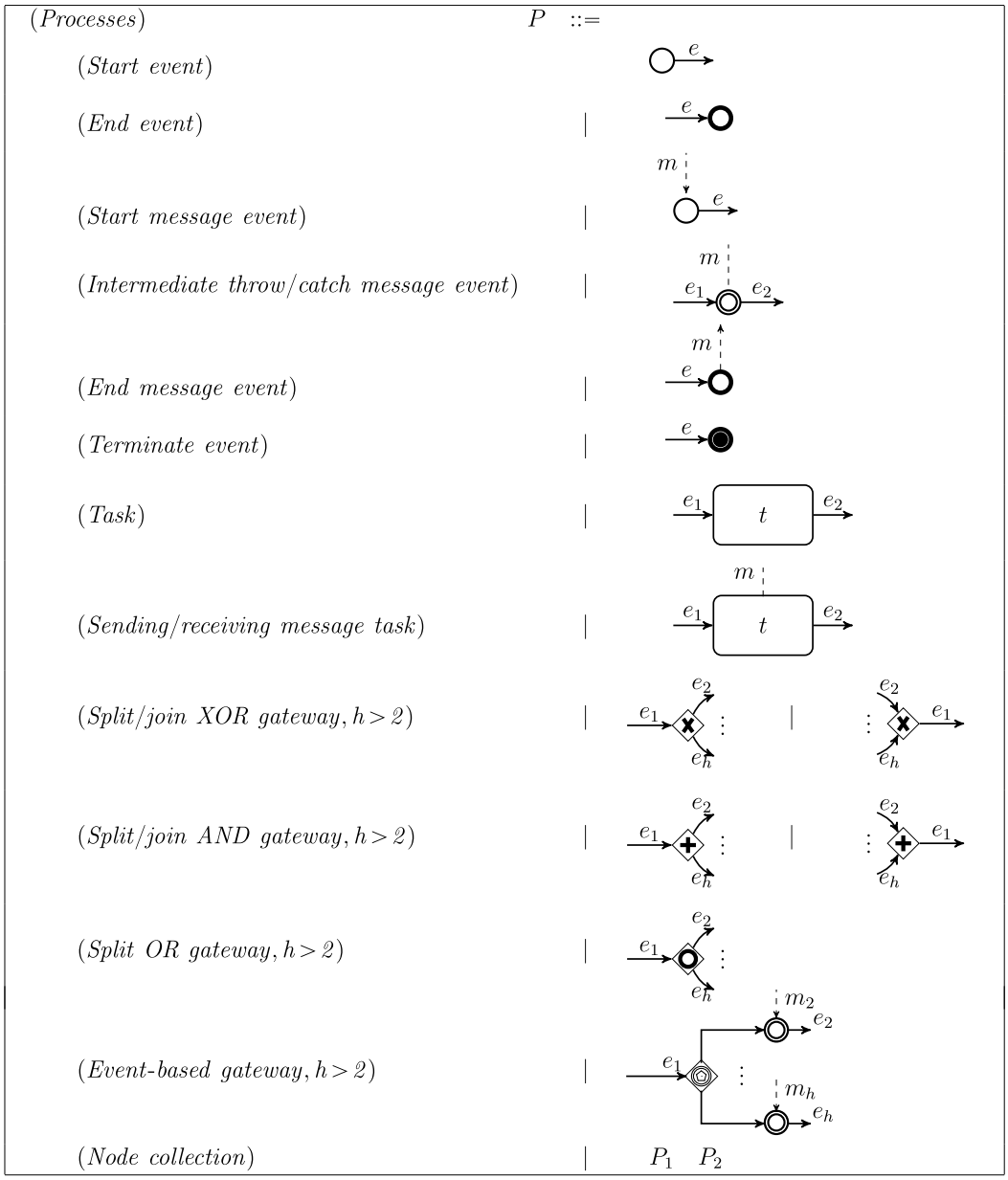


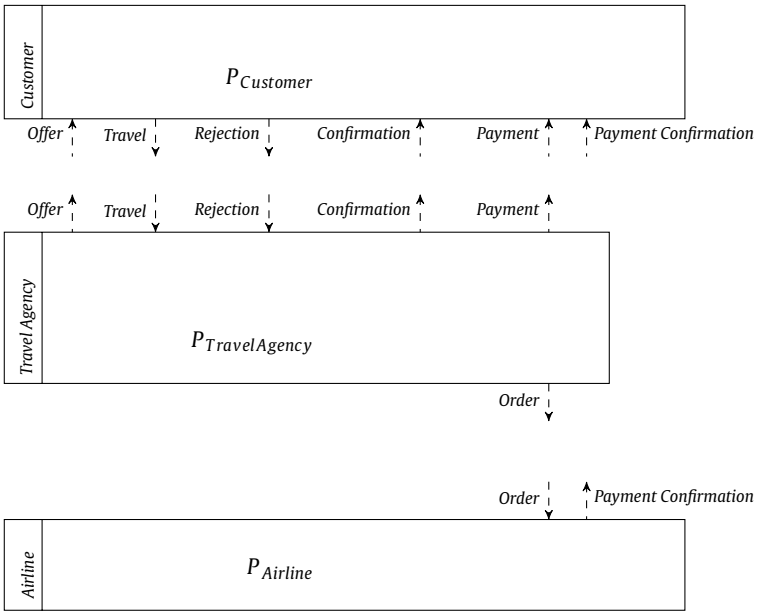
Fig. 6. BPMN syntax: processes.

- in a process, for each sequence edge labeled by *e* outgoing from a node, there exists only one corresponding sequence edge labeled by *e* incoming into another node, and vice versa;
- for each incoming (resp. outgoing) message edge labeled by *m* at pool level, there exists only one corresponding incoming (resp. outgoing) message edge labeled by *m* at the level of the process within the pool, and vice versa.

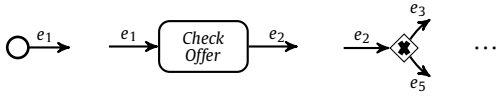
Well-definiteness could be easily checked through a standard (and trivial) static analysis; more practically, the rationale is that each term of the language can be easily derived from a BPMN model whose only constraint is to have (organization and message/sequence edge) unique names.

Notably, in this work we do not consider specifications using the OR-Join gateway, because it is not of common use in practice (in our dataset, only around the 4% of models include this element). Indeed, this is mainly due to the fact that its semantics is quite intricate (as explained, e.g., in [22–25]) and, hence, its use may lead to models with puzzling meaning for most of their target audience. Specifically, the semantics of the OR-join gateway is context dependent, i.e. the evaluation of its outgoing conditions depends on the current status of the process graph upstream of the gateway.

Running Example (1/3). The BPMN model presented in Section 2.2 is expressed in our syntax as the following collaboration:



where (an excerpt of) process $P_{Customer}$ is defined as follows:



and processes $P_{TravelAgency}$ and $P_{Airline}$ are defined in a similar way. □

4. Operational semantics

Defining the operational semantics of BPMN collaborations requires to deal with both workflow and collaborative aspects at the same time, while keeping the formalization easy to understand and fruitfully exploitable. More specifically, dealing with workflow aspects requires synchronizing the process elements in order to model the internal execution flow of an organization, while dealing with collaborative aspects requires managing asynchronous message exchanges between different organizations. A further issue concerns the proper management of the killing effect of the terminate end event, which must be confined within a single process (and not propagated to the other organizations). To overcome these issues we defined the operational semantics by separating process and collaboration layers. In the former layer we deal with internal interactions among process elements, including the effects of the terminate end event, while in the latter layer we only focus on inter-communication between organizations. Decoupling these aspects contributes to improve the understandability of the semantics, while allowing to deal with relevant aspects for collaborations.

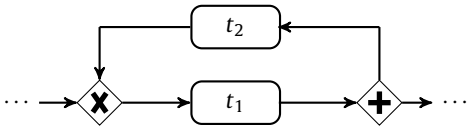
We give the semantics of BPMN[9] in terms of *marked collaborations*, i.e. collections of pools equipped with a marking. A *marking* is a distribution of tokens over pool message edges and process elements. This resembles the notions of token and marking in Petri Nets, that is not surprising as such formalism has strongly inspired the workflow constructs of BPMN. Our tokens, similarly to the token-passing semantics in [27,28], move along the syntax constructs, acting as sort of program counters.

Therefore, the operational semantics of BPMN is defined over an enriched syntax w.r.t. the one given in Section 3, where message edges, sequence edges, events and tasks are marked, i.e. labeled, by (possibly multiple) tokens. As a matter of notation, a single token is denoted by $\bullet$, while multiple tokens labeling a message edge m (resp. sequence edge e) are denoted by $m.n$ (resp. $e.n$), where $n \in \mathbb{N}$ is the token multiplicity. Finally, when tokens mark a terminate event, n is simply written beside the event (because, due to the graphical representation of this kind of event, a token inside would not be readable).

In this work we only consider business processes enacted with single instances. In fact, dealing with multiple instances in presence of message interactions would require to properly deliver each message to its appropriate instance; this would

[9] This is a revised and extended version of the operational semantics in [26].

add complexity to our formal treatment, which we want to avoid in order to keep it as easy to understand as possible. On the other hand, the use of tokens with multiplicity is necessary also with single instances, e.g. due to the behavior of the combined use of AND and XOR gateways as in the following piece of BPMN model.



Therefore, the initial marking of a collaboration assigns a single token to each start event of each process in the collaboration. Notably, in the initial marking a message start event is marked by a token, like a (non-communicating) start event, to indicate that the process instance is ready to start by receiving a message.

Formally, the operational semantics of marked collaborations is defined in the SOS style by relying on the notion of Labeled Transition System (LTS). The labeled transition relation of the LTS defining the semantics of collaborations, at pool layer, is induced by the inference rules in Fig. 7. We write $C \overset{l}{\longmapsto} C'$ to mean that "collaboration C can perform a transition labeled by l and become C' in doing so". Transition labels are generated by the following production rule.

$$(\textit{Labels}) \qquad l \ ::= \ o : \tau \ \mid \ o : ?m \ \mid \ o_1 \to o_2 : m$$

The meaning of labels is as follows. Label $o : \tau$ denotes an internal action τ performed by the process instance of organization o, label o:?m refers to the taking place of a receiving action from the organization o, while $o_1 \to o_2 : m$ denotes the exchange of a message m from organization o_1 to o_2. The definition of the above relation relies on an auxiliary transition relation defining the semantics of process instances and induced by the inference rules in Figs. 8, 9, 10, 11 and 14. We write $P \overset{\alpha}{\longmapsto} P'$ to mean that "process P can perform a transition labeled by α and become P' in doing so". The labels used by this auxiliary transition relation are generated by the following production rules.

$$(\textit{Actions}) \qquad \alpha \ ::= \ \tau \ \mid \ !m \ \mid \ ?m$$

$$(\textit{Internal actions}) \quad \tau \ ::= \ \textit{running}\ t \mid \ \textit{completed}\ t \mid \ (-\tilde{e_1}, +\tilde{e_2}) \mid \ \textit{kill}$$

where notation $\tilde{e}$ indicates a set of edges. The meaning of labels is as follows. Label τ denotes an action internal to the process, while $!m$ and $?m$ denote sending and receiving actions, respectively. The meaning of internal actions is as follows. Labels *running* t and *completed* t denote the start and completion of the execution of task t, respectively. The pair $(-\tilde{e_1}, +\tilde{e_2})$ denotes movement of workflow tokens in the process graph. In particular, one token is removed from each edge in $\tilde{e_1}$ and one is added to each edge in $\tilde{e_2}$; whenever one of the two sets of edges is empty, its field is omitted from the pair. Finally, *kill* denotes the termination of the whole process.

We now briefly comment the rules in Fig. 7. The first two rules allow a single pool, representing an organization o, to evolve according to the evolution of its enclosed process P. In particular, if P performs an internal action (rule *C-Internal*), or a receiving action (rule *C-Receive*), the pool performs the corresponding action at collaboration layer, i.e. the label is enriched with the name o of the organization performing the action. Notably, rule *C-Receive* can be applied only if there is at least one message queued in the message edge m of the pool (condition $n > 0$); this because one token is consumed by this transition. Rule *C-Deliver*, instead, involves two pools: one corresponding to the sending organization o_1, another corresponding to the receiving organization o_2. The resulting transition has the effect of increasing in the pool of o_2 the number of tokens queued in the message edge labeled by m. It is worth noticing that, as prescribed by the BPMN 2.0 specification [3, Sections 8.3.2 and 13.2.3], inter-organization communication is *asynchronous*: the sending action is not blocking, while the receiving one is blocking when there is no message to consume. Finally, the *C-Interleaving* rule permits 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. Notice that we do not need symmetric versions of rules *C-Deliver*, and *C-Interleaving* because, as already mentioned in Section 3, we identify collaborations up to commutativity and associativity of pool collection.

Rules in Fig. 8 deal with events: start, end and intermediate. All these rules are axioms, i.e., they have no premises. Rule *E-Start* moves a token from a start event to its outgoing edge e, thus producing the transition label $+e$. The propagation of this marking update to the incoming edge e (connected to another node of the process) is dealt with by the interleaving rules in Fig. 14 (see comments below). Instead, rule *E-End* just consumes a token from the edge e incoming in the end event, thus producing the transition label $-e$. Notably, incoming tokens are simply consumed, so that the end event will never be marked. In case of message events, a message edge is marked by a token when the corresponding sending or receiving action is performed.[10] For example, when a start event is enabled (i.e., it is marked by a token), its marking

[10] This means that, differently from the use of tokens at the collaboration layer, a token marking a message edge at the process layer does not represent a queued message, but a status of the event. The status of communicating tasks is dealt with accordingly.

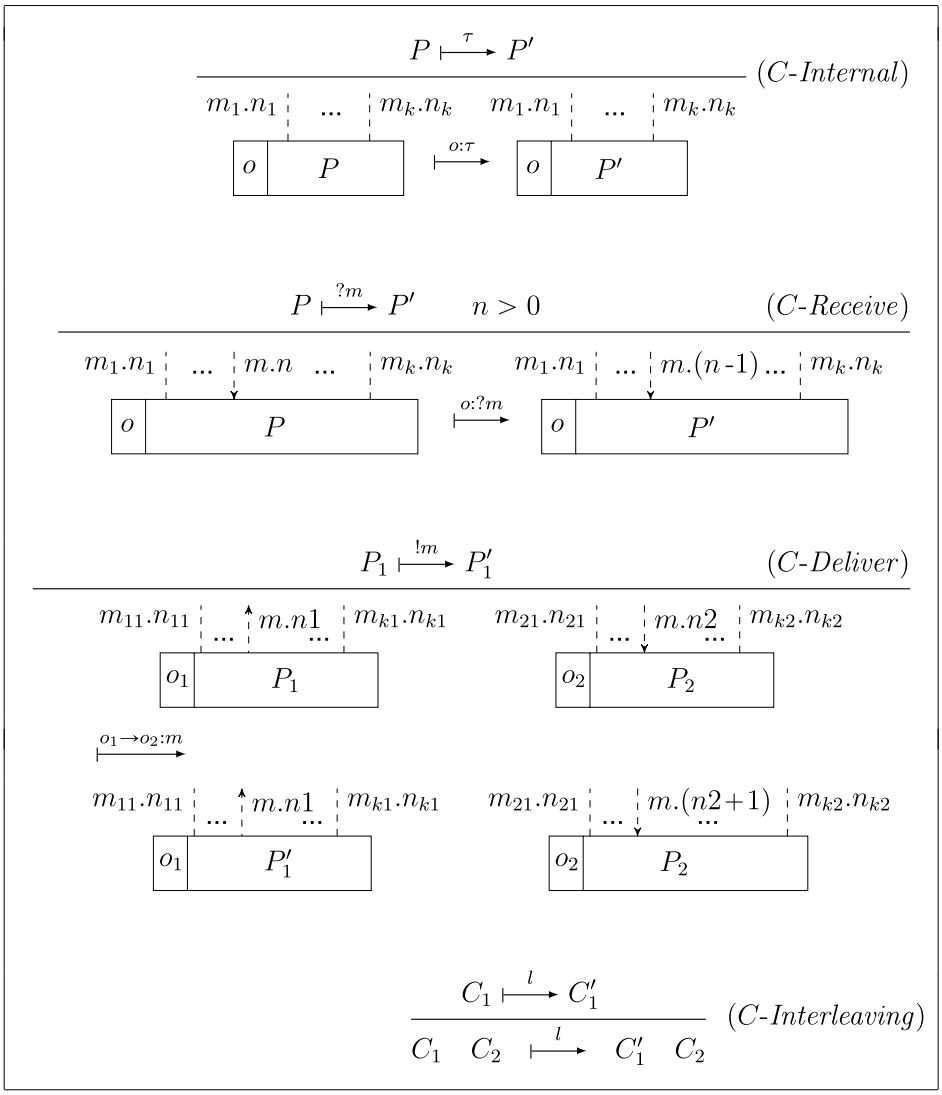


Fig. 7. BPMN operational semantics: collaboration layer.

can only evolve upon the reception of an incoming message m (rule *E-Receive*). In particular, the token is moved to the message edge and the label $?m$ is observed. Then, by applying rule *E-MsgStart*, the token is moved to the outgoing edge e, by producing the corresponding label. Now, no further messages can be received by the event (because it is not marked), as we do not consider process models with multiple instances. Rules *E-MsgEnd* and *E-Send* deal with message end events. In particular, rule *E-MsgEnd* moves a token from the edge e to the end event, thus producing the transition label $-e$. Then, rule *E-Send* removes the token from the end event and produces the transition label $!m$. In this case, more messages can be sent, one for each token that reaches the event. To obtain this behavior, the message edge labeled by m is not marked by a token, thus allowing rule *E-Send* to be reapplied. Concerning the terminate event, the first time a token reaches the event (rule *E-Terminate*), the *kill* label is produced and all tokens in the term are removed (rule *E-Kill*). The killing effect is propagated to all the other elements of the process by the interleaving rules in Fig. 14. Finally, the rest of the rules in Fig. 8 define the behavior of the intermediate events. These rules are similar to the ones previously commented, with the only peculiarity that at most one token can mark the event; this disallows the event to concurrently receive or send messages.

Rules in Fig. 9 deal with gateways. The effect of these rules is simply changing the marking of the process, by moving tokens among edges and producing transition labels of the form $(-\tilde{e_1}, +\tilde{e_2})$. For example, the effect of the rule *G-AndSplit* is to consume a token from the incoming edge e_1 of the AND gateway and to add a token to each outgoing edge e_i, with $2 \leq i \leq h$.

Rules in Fig. 10 are axioms devoted to the evolution of event-based gateways. Rule *G-EnableEvent* simultaneously enables all the intermediate message events connected to the gateway. As soon as one of them receives a message, the other receive events are disabled (rule *G-ReceiveEvent*) and the corresponding sequence edge is activated (rule *G-CompleteEvent*).

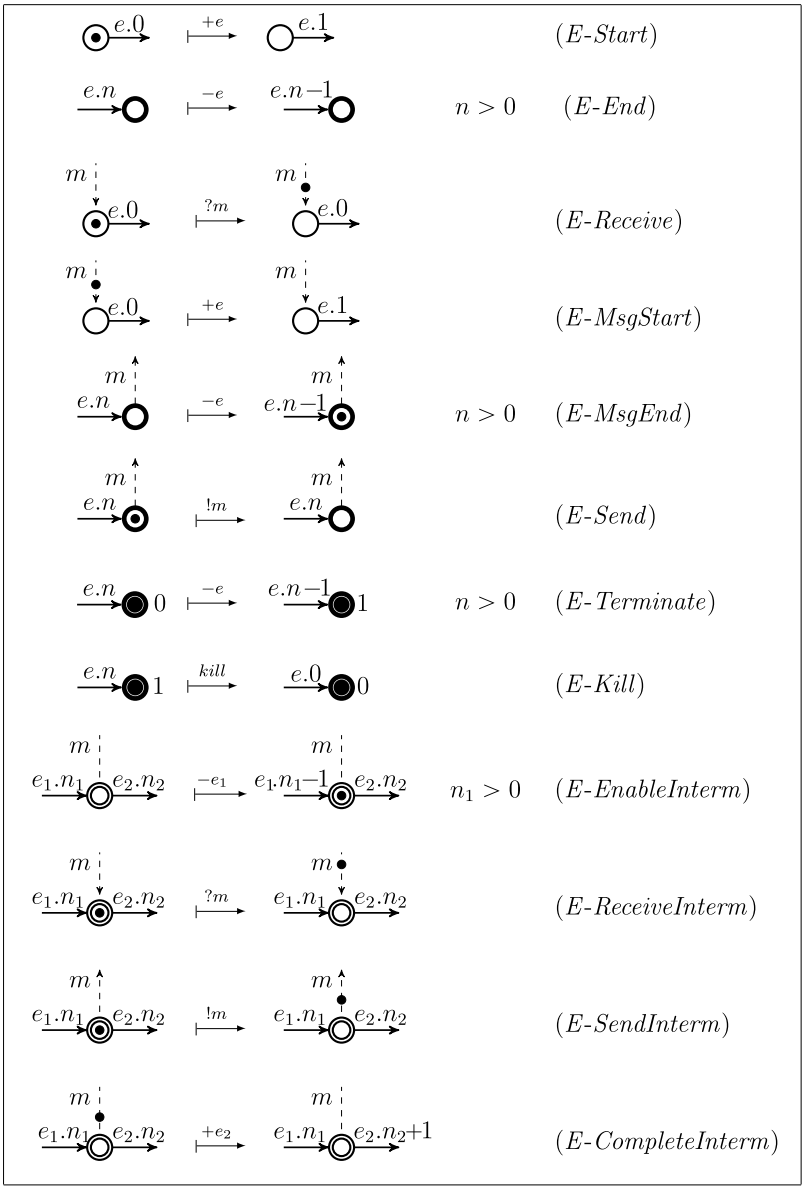


Fig. 8. BPMN operational semantics: process layer (event constructs).

Rules in Fig. 11 are axioms devoted to the evolution of tasks. The status of a task depends on the position of the token marking the task (Figs. 12 and 13).

- *Disabled*: the task is not active; this status is denoted by the absence of a token within the task rectangle (and its message edge, if present).
- *Enabled*: the task has been activated, by the consumption of a token from its incoming edge, and is going to run; this status is denoted by a token placed on the left of the task name.
- *Running*: the task is in execution; this status is denoted by a token placed on the top of the task name. In case of a communicating task, after the message is sent/received, the token is moved to the message edge; this permits to distinguish the two different phases of the running status of communicating tasks.
- *Completed*: the task execution is terminated and the flow can proceed, by the insertion of a token in its outgoing edge; this status is denoted by a token placed on the right of the task name.

The status of a task changes from disabled to enabled by means of rules *T-Enable* or *T-Enable*$_{msg}$. Notably, a task can be activated only when no token marks it; this means that parallel executions of the same task are not allowed. When the

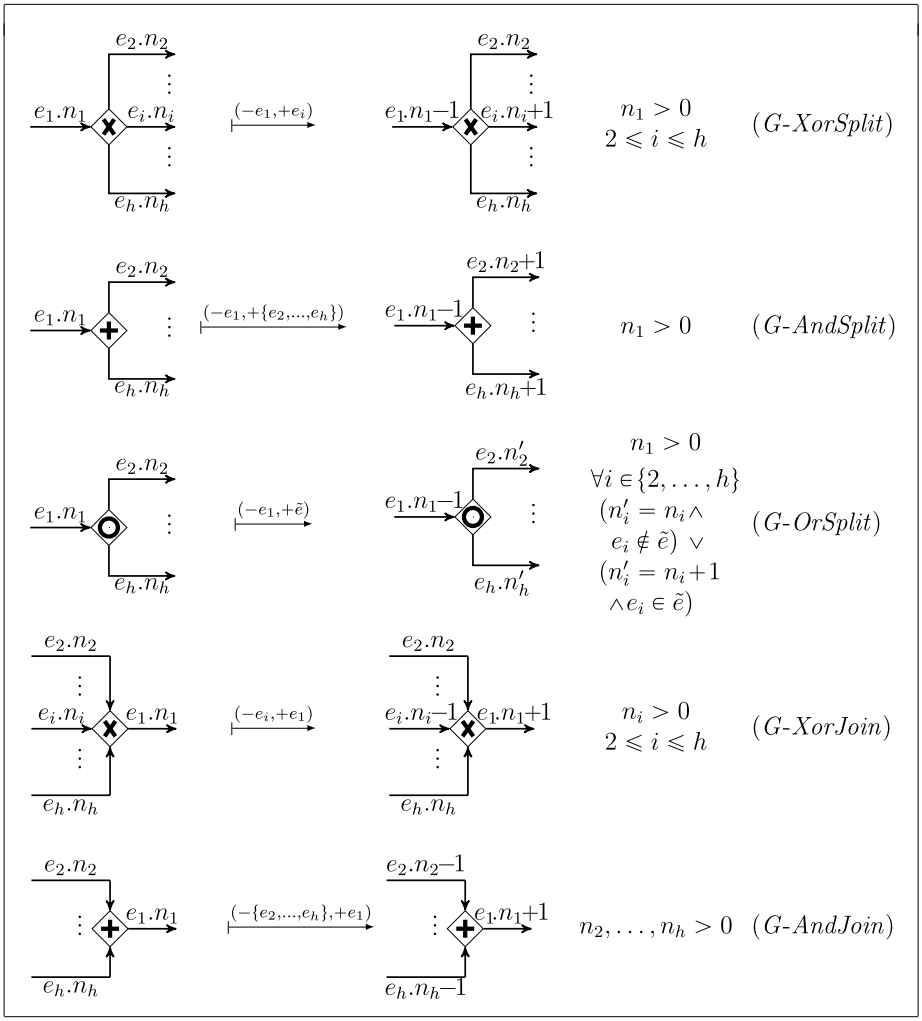


Fig. 9. BPMN operational semantics: process layer (gateway constructs).

status of task t changes to running (rules *T-Running* or *T-Running*$_{msg}$), the corresponding label *running t* is generated. Then, in case of communicating tasks, when the sending or receiving action is performed, the corresponding label $!m$ or $?m$ is generated and the message edge m is marked (rules *T-Send* and *T-Receive*). At the end of the execution of task t, the completion of t is notified by means of label *completed t* (rules *T-Complete* and *T-Complete*$_{msg}$). Thus, the task status is set to disabled and the execution flow can proceed (rules *T-Proceed* and *T-Proceed*$_{msg}$).

The last group of rules, shown in Fig. 14, deal with interleaving of process node evolutions. The first two rules are applied when the evolution involves a change in the marking of process edges, while the last one is applied in the other cases. In particular, rule *N-MarkingUpd* relies on the *marking updating function* $P \cdot (-\tilde{e_1}, +\tilde{e_2})$, which returns a process obtained from P by unmarking (resp. marking) edges in $\tilde{e_1}$ (resp. $\tilde{e_2}$). Formally, this function is inductively defined on the structure of process P and, with abuse of notation, extends to terms of the form $e.n$ as follows:

$$e.n \cdot (-\tilde{e_1}, +\tilde{e_2}) = \begin{cases} e.n-1 & \text{if } e \in \tilde{e_1} \\ e.n+1 & \text{if } e \in \tilde{e_2} \\ e.n & \text{otherwise} \end{cases}$$

Notably, in the above definition we exploit the fact that, since self-loop are not admitted in a process, it holds $\tilde{e_1} \cap \tilde{e_2} = \emptyset$. In each base case of the inductive definition of the marking updating function, we simply apply the function to the terms $e.n$ labeling the edges of process nodes. We report below few significant cases of the marking updating function definition (the others are similar):

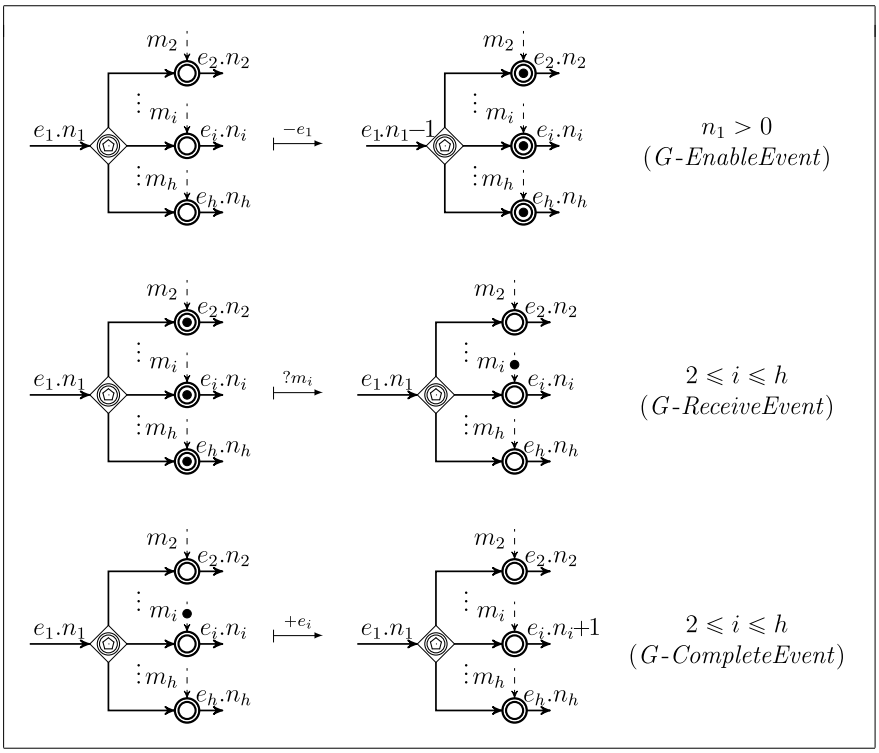
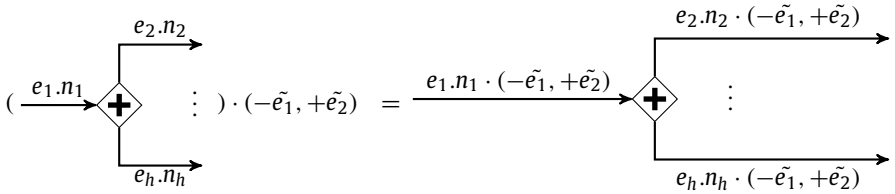
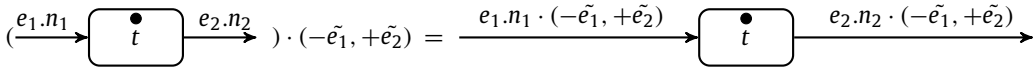


Fig. 10. BPMN operational semantics: process layer (event-based gateway construct).

$$(P_1 \quad P_2) \cdot (-\tilde{e_1}, +\tilde{e_2}) \;=\; P_1 \cdot (-\tilde{e_1}, +\tilde{e_2}) \quad P_2 \cdot (-\tilde{e_1}, +\tilde{e_2})$$

$$(\xrightarrow{e.n_1}\bigcirc) \cdot (-\tilde{e_1}, +\tilde{e_2}) \;=\; \xrightarrow{e.n_1 \cdot (-\tilde{e_1}, +\tilde{e_2})}\bigcirc$$





Similarly, rule *N-Kill* relies on the killing function $P\dagger$, which returns a process obtained from P by completely unmarking it. The significant cases defining thus function are as following:

$$(P_1 \quad P_2)\dagger \;=\; P_1\dagger \quad P_2\dagger$$

$$(\xrightarrow{e.n_1}\bigcirc)\dagger \;=\; \xrightarrow{e.0}\bigcirc$$

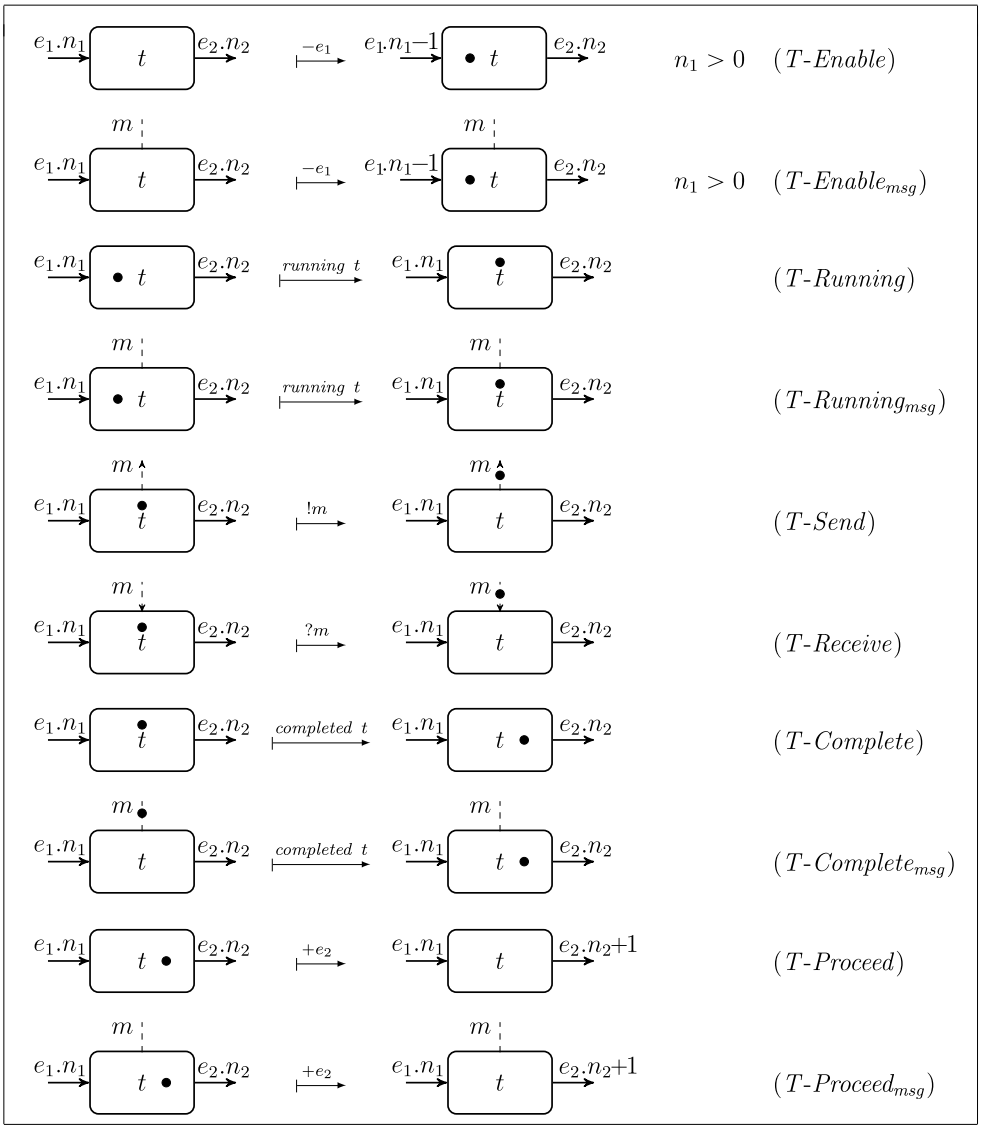


Fig. 11. BPMN operational semantics: process layer (task constructs).

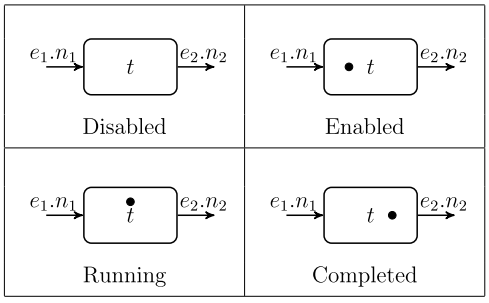


Fig. 12. BPMN operational semantics: status of non-communicating tasks.

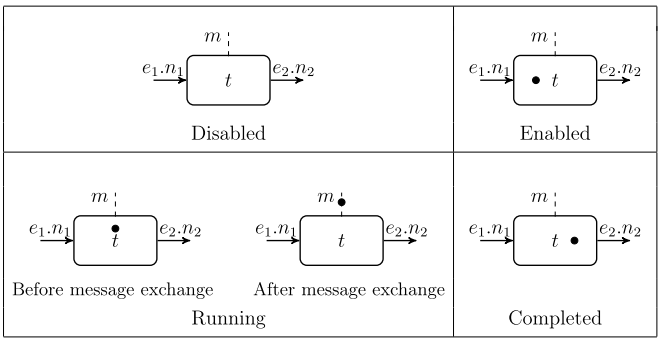


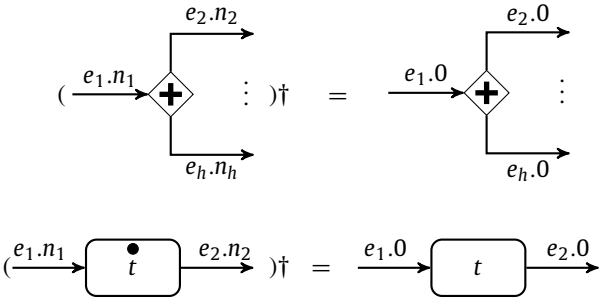
Fig. 13. BPMN operational semantics: status of communicating tasks.

$$\frac{P_1 \overset{(-\tilde{e_1},+\tilde{e_2})}{\longmapsto} P_1'}{P_1 \quad P_2 \overset{(-\tilde{e_1},+\tilde{e_2})}{\longmapsto} P_1' \quad P_2 \cdot (-\tilde{e_1},+\tilde{e_2})} \quad (N\text{-}MarkingUpd)$$

$$\frac{P_1 \overset{kill}{\longmapsto} P_1'}{P_1 \quad P_2 \overset{kill}{\longmapsto} P_1' \quad P_2\dagger} \quad (N\text{-}Kill)$$

$$\frac{P_1 \overset{\alpha}{\longmapsto} P_1' \qquad \alpha \notin \{(-\tilde{e_1},+\tilde{e_2}), kill\}}{P_1 \quad P_2 \overset{\alpha}{\longmapsto} P_1' \quad P_2} \quad (N\text{-}Interleaving)$$

Fig. 14. BPMN operational semantics: process layer (node collection).

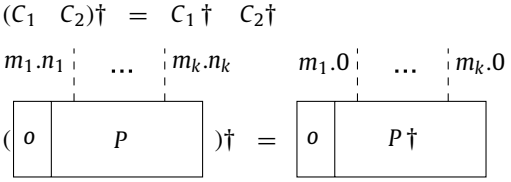


We conclude by presenting some properties of the operational semantics. Although these properties are typically expected, they have the key purpose of formally validating the soundness of our theoretical framework. First, we show that the structure of a process is preserved along its execution. Technically, we show that the process resulting from a transition coincides with the one triggering the transition, once all tokens are removed. To remove tokens from processes, we simply resort to the killing function $P\dagger$ defined above. Then, we extend the result to the evolution of marked collaborations.

Property 1 *(Process structure preservation). Let P be a process in a marked collaboration, if $P \overset{\alpha}{\longmapsto} P'$ then $P\dagger = P'\dagger$.*

Proof. We proceed by induction on the derivation of $P \overset{\alpha}{\longmapsto} P'$ (see Appendix A). □

To extend the above property to collaborations, with abuse of notation we extend the killing function as follows:



Property 2 *(Collaboration structure preservation). Let C be a collaboration, if $C \xmapsto{l} C'$ then $C\dagger = C'\dagger$.*

Proof. We proceed by induction on the derivation of $C \xmapsto{l} C'$ and by relying on Property 1 (see Appendix A). □

From the above properties, we obtain that also well-definedness (defined in Section 3) is preserved.

Corollary 1 *(Well-definedness preservation). Let C be a well-defined collaboration, if $C \xmapsto{l} C'$ then C' is well-defined.*

Proof. It trivially derives from Property 2, as well-definedness depends only on the structure of the collaboration and, according to Property 2, collaborations C and C' have the same structure. □

Running Example (2/3). We describe here the semantics of the BPMN model informally introduced in Section 2.2 and formalized in Section 3. Notably, for the sake of readability, in the marked collaborations reported in Figs. 15 and 16 we have merged sequence/message edges with the same name, omitted sequence edge names, and omitted notation of multiple tokens when their multiplicity is zero.

The initial state of the execution is represented by the collaboration in Fig. 15(a), where the start events of the processes of the three organizations are marked by a token each. Thus, the execution of the processes can start and, as a possible evolution, after few computational steps the state of the collaboration execution becomes the one shown in Fig. 15(b). In such a configuration, according to the position of the three tokens, the Travel Agency is performing the *Make Travel Offer* task (it is running before message exchange), while the Customer and the Airline are still in the initial state. More specifically, let e_1 be the edge outgoing from the *Offer Needed* start event, by applying rules *E-Start* and *N-MarkingUpd* at process layer the produced transition label is $+e_1$. This permits, at collaboration layer, to apply rules *C-Internal* and *C-Interleaving*, thus producing the transition label $TravelAgency : +e_1$. Then, by applying rules *T-Enable*$_{msg}$ and *N-MarkingUpd* at process layer, and the same rules previously used at collaboration layer, we obtain the transition label $TravelAgency : -e_1$. Finally, by applying rules *T-Running*$_{msg}$ and *N-Interleaving* at process layer, and again the same rules at collaboration layer, the produced label is $TravelAgency$: *running MakeTravelOffer*. It is worth noticing that, even if the *Customer* process can evolve with a transition labeled by ?*Offer* (by applying rule *E-Receive*), this evolution is inhibited at collaboration layer. Indeed, the only rule that could be applied at this level to deal with such label is *C-Receive*; however, it cannot be applied because the premise requiring the presence of an *Offer* message enqueued in the *Customer* pool is not satisfied.

Now, the status of the collaboration can further evolve and, after some steps, the Customer is performing *Check Offer* task and the Travel Agency is ready to receive messages according to the behavior of the event-based gateway, see Fig. 16(c). Finally, after further steps, the collaboration reaches the final configuration where no token is available. □

5. Model checking verification

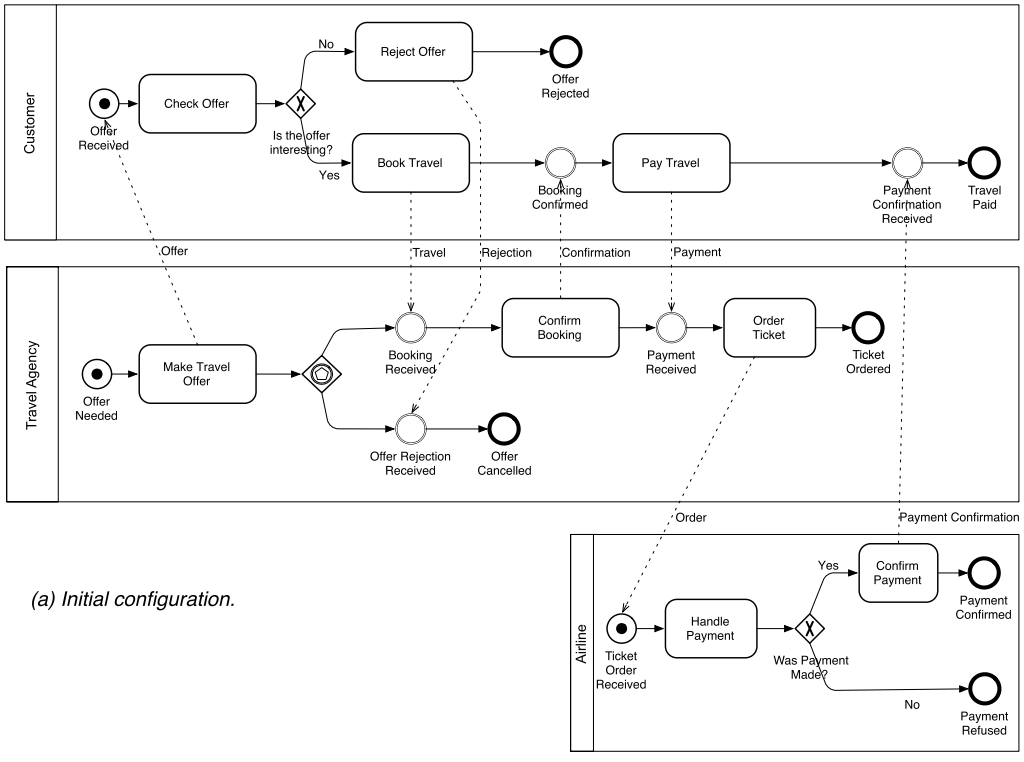
Verifying properties of BPMN collaborations is useful to ensure their correct execution. This verification includes inter-organizational properties, to detect potential errors in message exchanges. We recall that we abstract from data values, as we aim at exhaustively analyzing all executions of a given model, and not only those resulting from specific input values. This may lead to false positive errors that do not occur when considering specific data in collaboration models, which could not exhibit the erroneous behaviors. Notably, even if we focus on message exchange, our semantics also considers the internal execution of processes in the collaboration, thus enabling the verification of properties related to internal behaviors (e.g., absence of deadlock) as well.

Our BPMN formalization enables the verification of properties of interest for collaboration models. In our verification approach we express these properties in terms of LTL formulae [29]. The choice of LTL as logic for expressing properties is motivated by the fact that it is expressive enough for characterizing the relevant properties of collaborations and Maude already provides an LTL model checker [30]. We show in Section 6.2 how these properties are actually verified using our Maude implementation.

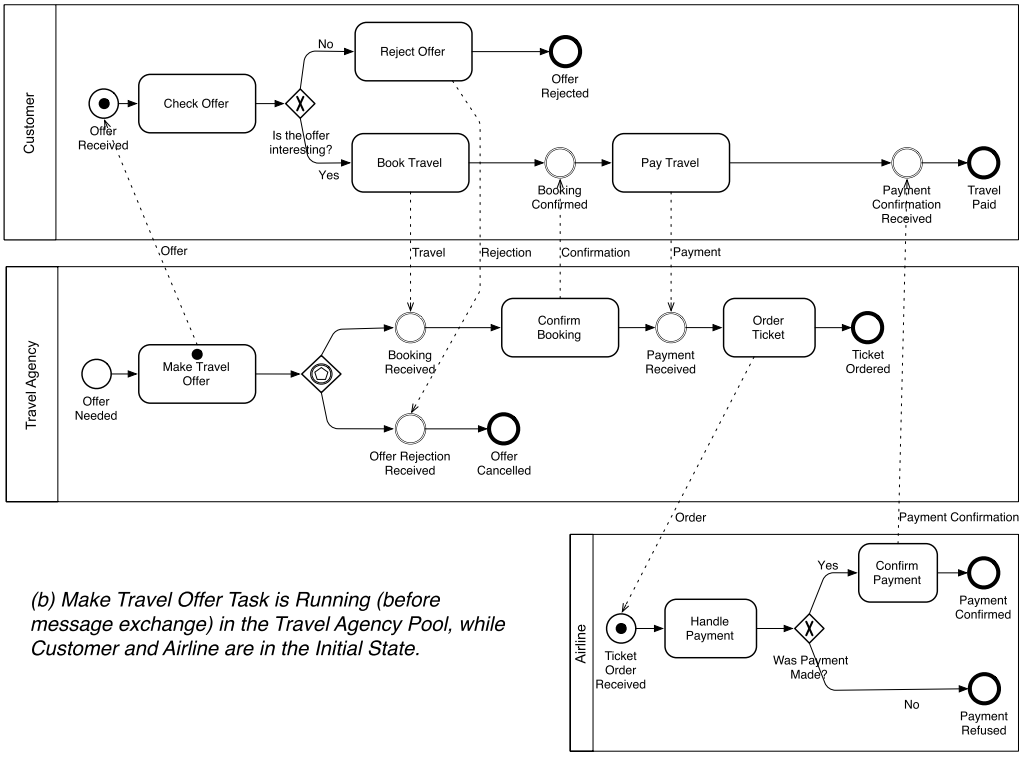
The LTL formulas we consider here have the following form:

- `<>` ϕ, where the operator `<>` (corresponding to the LTL operator F) is used to verify if a subformulae ϕ **eventually** holds;
- `[]` ϕ, where the operator `[]` (corresponding to the LTL operator G) is used to verify if a subformulae ϕ **globally** holds;
- ϕ `->` φ, where the operator `->` (corresponding to the LTL implication operator) is used to verify if the holding of a subformulae ϕ **implies** the holding of a subformulae φ;
- ϕ `|->` φ where the operator `|->` (corresponding to the LTL leads-to operator) is used to say it is always true that, if ϕ holds, then φ will eventually holds.

To ease the specification and analysis of typical properties of collaboration models, we have defined some recurrent subformulae representing high-level BPMN-related concepts. They can be composed using the LTL logical operators introduced

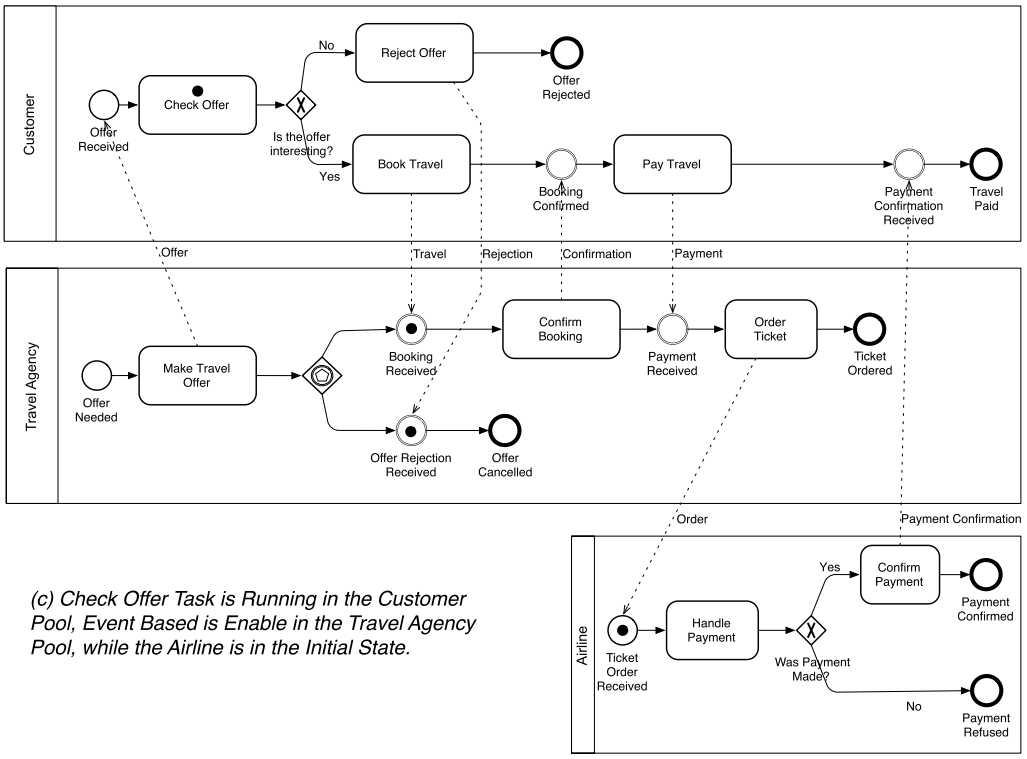


(a) Initial configuration.



(b) Make Travel Offer Task is Running (before message exchange) in the Travel Agency Pool, while Customer and Airline are in the Initial State.

Fig. 15. Semantics of the running example (part I of II).



(c) Check Offer Task is Running in the Customer Pool, Event Based is Enable in the Travel Agency Pool, while the Airline is in the Initial State.

Fig. 16. Semantics of the running example (part II of II).

above, allowing their reuse in different properties. We report below some predicates we have used to define the formulae in Table 2 verified over the running example.

- **aBPstarts**. It is satisfied (i.e., its evaluation will return true) if at least a process in the collaboration can start. This, according to our formal semantics, happens when at least a start event is marked or, in case of message start event, it has also a message.
- **aBPstartsParameterized**. It is satisfied if the process associated to a specific Pool (identified by its Organization Name) can start.
- **aBPSndMsg** (resp. **aBPRcvMsg**). It is satisfied if a specified message is sent (resp. received).
- **aBPends**. It is satisfied if in a Collaboration there is at least one process associated to a pool which can always reach an end state. This happens when the end event is marked or, in case of message end event, the message is sent.
- **aBPendsParameterized**. It is satisfied if a process associated to a specific Pool (identified by its Organization Name) can reach an end event.
- **aTaskIsEnabledParameterized**. It is satisfied if a specified Task in the Collaboration can always be marked as “enabled”. This, according to our formal semantics, happens when a Task passes from the “disable” to the “enabled” state.
- **aTaskIsRunningParameterized**. It is satisfied if a specified Task in the Collaboration will be able to run. This, according to our formal semantics, happens when the specified Task passes from the “enabled” to the “running” state.
- **aTaskCompletes**. It is satisfied if in the all Collaboration, there is at least one Task that can always be marked as “completed”. This, according to our formal semantics, happens when a Task passes from the “running” to the “completed” state.
- **aTaskCompletesParameterized**. It is satisfied if a specified Task in the Collaboration will always complete. This, according to our formal semantics, happens when the specified Task passes from the “running” to the “complete” state.
- **PendingTokenParameterized**. It is satisfied if in a specific Pool there is at least one token to be consumed, whether the token is in a message flow or in a sequence flow it is not relevant.
- **PendingMsgTokenParameterized**. It is satisfied if in a specific Pool there is at least one token in a message flow to be consumed. This, according to our formal semantics, for instance happens when a Message remains in the message queue since no task is able to process it (i.e., a Message Task is not “enabled” even if a message is available).
- **PendingFlowTokenParameterized**. It is satisfied if in a specific Pool there is at least one token in a sequence flow or within a task to be consumed. This, according to our formal semantics, for instance happens when a Task is not able to pass from the “enabled” to the “completed” status due to a missing message.
- **SafeStateParameterized**. It is satisfied if a specific Process there is at most one token for each sequence flow.

Table 2
LTL formulae verified in Maude.

1	`<> aBPstarts`	true
2	`<> aBPstartsParameterized("Airline")`	false
3	`<> aBPends`	true
4	`<> aBPendsParameterized("Customer")`	false
5	`<> aTaskCompletes`	true
6	`<> aTaskCompletesParameterized("Handle Payment")`	false
7	`[] (aTaskIsRunningParameterized("Confirm Booking") ->(<> aTaskCompletesParameterized("Confirm Booking")))`	true
8	`aTaskCompletesParameterized("Handle Payment") |-> aTaskCompletesParameterized("Confirm Payment")`	false
9	`aBPSndMsg("Customer", "Payment") |-> aBPRcvMsg("Customer","Payment Confirmation")`	false
10	`aBPSndMsg("Customer","Payment") |-> aBPRcvMsg("Travel Agency","Payment") |-> aBPSndMsg("Travel Agency","Order")`	true
11	`[] (aTaskIsEnabledParameterized("Confirm Payment") -> (<> aTaskCompletesParameterized("Confirm Payment")))`	true
12	`[] ((aBPstartsParameterized("Customers") -> <> aBPends("Customers")) ∧ (aBPstartsParameterized("Travel Agency") -> <> aBPends("Travel Agency"))∧ (aBPstartsParameterized("Airline") -> <> aBPends("Airline")))`	false
13	`[] (SafeStateParameterized("Customers") ∧ SafeStateParameterized("Travel Agency") ∧ SafeStateParameterized("Airline"))`	true
14	`[] ((aBPendsParameterized("Customers") -> ¬PendingToken("Customers")) ∧ (aBPendsParameterized("Travel Agency") -> ¬PendingToken("Travel Agency"))∧ (aBPendsParameterized("Airline") -> ¬PendingToken("Airline")))`	true
15	`[] ((aBPendsParameterized("Customers") -> ¬PendingMsgToken("Customers")) ∧ (aBPendsParameterized("Travel Agency") -> ¬PendingMsgToken("Travel Agency")) ∧ (aBPendsParameterized("Airline") -> ¬PendingMsgToken("Airline")))`	true
16	`[] ((aBPendsParameterized("Customers") -> ¬PendingFlowToken("Customers")) ∧ (aBPendsParameterized("Travel Agency") -> ¬PendingFlowToken("Travel Agency")) ∧ (aBPendsParameterized("Airline") -> ¬PendingFlowToken("Airline")))`	true

We describe below the meaning of the properties in Table 2 by means of the results of the verification carried out on the running example.

Running Example (3/3). First of all, we may wonder if the collaboration presents at least one process that can start; if it does not, probably there is something wrong with the model. To verify this, we can use *Property 1*, which verifies whether there is at least one process associated to a pool in the collaboration that will always be able to start. The property in our case is true since at least the process associated to Pool Travel Agency (and also the one associated to pool Customer) can always start.

We may also want to verify if pool Airline has a process that can always start. To verify this, we can use *Property 2*, which tests whether a process associated to a specific pool in the collaboration will always be able to start. The property in our case is false since it is possible that the tested pool Airline does not start; this can happen if the Customer rejects the travel offer.

Another property to verify may be the one that checks end state reachability of a process. *Property 3*, in our case verifies whether at least one process associated to a pool in the collaboration will always reach an end state. The property is true since at least one pool Travel Agency can always reach an end event between Ticket Ordered and Offer Canceled. The former if the Customer is intended to book and pay for a travel; the latter if the Customer does not intend to book a travel.

If we want to verify whether a certain pool will always reach an end state, we can use *Property 4*. In our case we verified that the process associated to the selected Pool Customer cannot always reach an end state. However, this pool can, and will, always start; we conclude that there must be some issues with the way the process is modeled. In particular, in our scenario, we can notice that the process associated to pool Customer may not reach the end event Travel Paid if the Airline process ends in the Payment refused event without providing a notification to the Customer; so the Customer process will be stuck waiting for a Payment Confirmation. This represents a clear case of deadlock.

Another property, *Property 5*, allows us to verify whether at least one Task in the collaboration will always complete. This property, on our scenario, is true since at least the task Make Travel Offer will always complete.

Property 6 can be used to verify whether a specific task in the Collaboration will always complete. This property is false since the selected task Handle Payment can be avoided if the Customer rejects the travel offer executing task Reject Offer;

in this case, Travel Agency will not order a ticket to Airline so the Airline represented process will not start and the task Handle Payment will not be executed (neither completed).

Property 7 and *Property 8* are constructed by verifying multiple properties. The former verifies whether having a specified task Confirmation Booking in the running states implies that the same task will sooner or later complete (passing from the state running to complete); the latter verifies whether the completion of a specified task Handle Payment implies the completion of another task such as Confirm Payment. *Property 7* is true, since the Confirmation Booking task, once in the running state it can always reach the completed state. *Property 8* is encoded using the temporal operator "|->" which corresponds to the LTL operator leads-to and it allows to express the same thing of *Property 7* but in a more compact way. *Property 8* is false, since the Handle Payment task can actually be executed without the implication of executing the task Reject Offer.

Property 9 is false, since even if the Customer has sent the payment it may happen that the corresponding confirmation is never received. The result of this property is not surprising considered the result of the verification of Property 8. More generally, the property is interesting because it is expressed in terms of messages exchanges rather than task execution.

Property 10 is true, since when the Customer performs the payment, the corresponding message will be received by the Travel Agency, thus triggering the send of the Order message.

Property 11 is true, since it verifies that whenever the specified task Confirm Payment is enabled then it will eventually complete (i.e., it is able to pass from the status enabled to completed). This shows that our approach allows fine-grain specification of properties, as it permits dealing with the different statuses of the task lifecycle.

Property 12 models the Option to Complete property of Soundness (see Section 2.3). Specifically, it checks if all started participants will always complete. In our scenario, the property is evaluated to false, due to the decision taken by Airline that affects the behavior of Customer, which, even if it is able to start, it is not able to complete in all the possible executions. Indeed, if the Airline does not send the payment confirmation to the Customer, this remains stuck on the Payment Confirmation Receive event.

Property 13 models Safeness (see Section 2.3). It is true, since in all the processes there is at most one token for each sequence flow. It is worth noticing that the verification of this property sheds light on the differences between the notion of safeness on BPMN collaborations with respect to that on Petri Nets applied as it is to translations of BPMN collaborations according to the mapping in [6]. In fact, safeness of a BPMN collaboration only refers to the tokens on the sequence edges of the involved processes, while in its Petri Nets translation we considered refers to both message and sequence edges. Indeed, such distinction is not considered in the translations, because a message is rendered as a (standard) token in a place. Hence, a safe BPMN collaboration may be considered unsafe by relying on the Petri Nets notion.

Properties 14–16 model different forms of Proper Completion of Soundness according to the requirements we impose on message queues. All three properties are true, since when the processes included in the collaboration complete, it results that there are no more pending tokens in the processes (neither control flow tokens nor message flow ones). More generally, these properties permit to clarify the effect of the terminate event on soundness. Indeed, the use of a terminate event in place of an end event may make sound a collaboration that was not.

6. Maude implementation

To concretely validate our theoretical definitions, and to practically enable verification of BPMN collaborations, both the syntax and the operational semantics we defined have been implemented using Maude. Maude is a language that can be used to model systems and the actions within those systems [31]. Systems are specified by defining algebraic data types representing states of the system, and rewrite rules defining state transitions. Rewrite rules are the building blocks of the rewriting logic introduced by José Meseguer in [32].

Our Maude implementation of BPMN is inspired by the seminal work in [33] about the implementation of CCS [34] in Maude, where inference systems are mapped into rewriting logic. However, our technical development significantly differs from [33], due to the differences between CCS and BPMN in terms of target audience, richness and complexity. We also present here the verification of some properties on the application scenario.

Our choice of Maude as language for implementing our BPMN semantics is motivated by the fact that Maude permits to code rules of the operational semantics simply as rewriting rules. This one-to-one correspondence ensures the faithfulness of the Maude implementation of the semantics with respect to its formal definition. On the one hand, this permits to validate our pen-and-paper formalization. This also simplifies the validation of our prototypical Maude implementation, which is carried out through a careful code inspection, testing of single rules and application to real scenarios. Moreover, since Maude specifications are executable, we obtain an interpreter for BPMN models enabling: *(i)* the study of their states evolution, and *(ii)* their formal verification by means of the Maude LTL model checker [30]. The encoding from the XML serialization of BPMN models (generated by any BPMN modeling environment) into the proposed Maude syntax is supported by an automatic procedure.[11] On the other hand, as Maude uses inference mechanisms for computing the execution steps, in case of large models with high degree of parallelism the verification performance may be significantly affected. However, this is not a major limitation for our study, because real-world models have typically a small size (around 30 elements in

[11] Our BPMNtoMaude tool is available at https://goo.gl/EYaiVL.

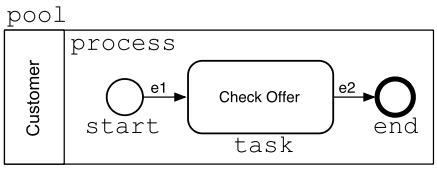


Fig. 17. Minimal collaboration example.

average, see Table 1). Moreover, our aim here is to provide just a proof-of-concept implementation, for assessing the benefits of our approach in modeling and verification of BPMN collaborations. An optimization of this Maude implementation and its validation are presented in [35].

The Maude source code, the running example specification and the code for the analysis are available at: https://goo.gl/qQbGgg.

6.1. Syntax and semantics

We summarize here the main ideas used in our implementation. For the sake of readability we provide below just an exemplification of syntax and semantics implementation in Maude by focusing on a minimal collaboration example including: a start event, a task, and an end event, which are enclosed in a pool. The example is reported in Fig. 17. A more complete example, referring to the application scenario depicted in Fig. 3, is reported in Appendix B.

In our implementation each BPMN element is declared as a Maude operation written according to the following general form.

```
op element(_, ... ,_)  : Sort-1 ... Sort-k  -> RSort .
```

Its meaning is:

- the keyword `op` indicates the definition of an operation;
- `element` is the name of the operation that we define;
- `(_, ... ,_)` specifies that the `element` operation is characterized by a number of parameters, whose sorts `Sort-1` , ... , `Sort-k` are reported after the `:` symbol;
- the `->` symbol is followed by the resulting sort `RSort` of the `element` operation;
- finally, the `.` symbol is used to end the line of code.

We present now the Maude operations representing the syntax of the BPMN elements used in our minimal collaboration example. Specifically, we present first the syntax of start and end events, which are quite similar, then the syntax of task, pool and collaboration elements. In the operation definition some sorts are defined by means of other operations shared between the elements (e.g., those referring edges), which will be presented only the first time we use them.

The implementation of the syntax of the start event is as follows.

```
op start(_,_) : Status Edge -> ProcElement .
```

Its meaning is:

- `start` is the name of the operation corresponding to the start event element;
- `(_,_)` specifies that the `start` operation is characterized by two parameters with sorts `Status` and `Edge`;
- the resulting sort `ProcElement` indicates that the `start` operation is an element of the process.

The sort `Status` is defined by means of constants, i.e. an operation without parameters meaning that an element of the sort `Status` can only assume the values specified in the declaration (e.g., `disabled` and `enabled` in case of start events). We use the keyword `ops` to define multiple declarations.

```
ops disabled enabled running completed ...  : -> Status .
```

An element of the sort `Edge` is defined by means of the `.` operation, which allows to compose an `EdgeName` with an `EdgeToken`. Indeed an edge consists of its name and a natural value indicating the number of tokens marking the edge.

```
op _._ : EdgeName EdgeToken -> Edge .
```

In the minimal example, the start event is followed by a task, and the syntax for the task implementation in Maude is defined by means of the following code.

```
op task(_,_,_,_) : Status Edge Edge TaskName -> ProcElement .
```

Its meaning is:

- `task` is the name of the operation corresponding to a task element;
- `(_,_,_,_)` specifies that the `task` operation is characterized by four parameters, whose sorts `Status`, `Edge`, `Edge`, `TaskName` are respectively reported after the `:` symbol. Indeed, the task is formed by aggregating a status, an incoming edge, an outgoing edge, and a task name;
- the resulting sort `ProcElement` of the `task` operation indicates that a task is an element of the process.

Another BPMN element reported in the minimal example is the end event, whose syntax implementation is as follows.

```
op end(_) : Edge -> ProcElement .
```

Its meaning is:

- `end` is the name of the operation corresponding to the end event element;
- `(_)` specifies that the `end` operation is characterized by one parameter, whose sort is `Edge`;
- the resulting sort `ProcElement` indicates that an end event is an element of the process.

The presented BPMN elements are grouped together by means of the `|` operation, representing the node collection operator, that allows to compose terms of sorts `ProcElement` and `ActProcElement`:

```
op _|_ : ProcElements ProcElements -> ProcElements .
op _|_ : ActProcElement ProcElements -> ActProcElement .
```

The resulting composition is enclosed by an element of the sort `Process`, whose syntax implementation is as follows.

```
op proc(_) : ActProcElement -> Process  .
```

Its meaning is:

- `proc` is the name of the operation corresponding to a process;
- `(_)` specifies that the `proc` operation is characterized by one parameter, whose sort is `ActProcElement`;
- the resulting sort is `Process`.

The sort `ActProcElement` comprehends a list of process elements preceded by an `Action` (which is a label used to keep track of the evolution of the process) enclosed in curly brackets. Its definition is:

```
op {_}_ : Action ProcElement -> ActProcElement .
```

This approach, borrowed from [33], permits expressing labeled transitions in terms of rewrites. Indeed, roughly, the label is moved from the transition to the target state of the transition itself.

In addition, the process is included in a `Pool`, whose sort is defined by the following operation.

```
op pool(_,_,in:_,out:_) : OrgName Process Msgs Msgs ->
Collaboration .
```

Its meaning is:

- `pool` is the name of the operation corresponding to pool element;
- `(_,_,in:_,out:_)` specifies that the `pool` operation is characterized by four parameters, whose sorts are `OrgName`, `Process`, `Msgs`, `Msgs`, respectively. Indeed, the pool is formed by aggregating an organization name, a process, and two lists of messages (the one after symbol `in:` for the incoming messages, the other after symbol `out:` for outgoing messages). Note, the lists of messages may also be empty (in that case we use a defined constant `emptyMsgSet` after `in:` and `out:`);
- the resulting sort `Collaboration` indicates that a pool is an element of the collaboration.

Finally, the element that encloses the ones mentioned above is the collaboration element, whose syntax implementation is as follows.

```
op collaboration(_) : ActCollaboration -> Model .
```

Its meaning is:

- `collaboration` is the name of the operation corresponding to a collaboration;

- `(_)` specifies that this operation is characterized by one parameter, whose sort is `ActCollaboration`;
- the resulting sort `Model` indicates that a collaboration is an element of the overall model.

The sort `ActCollaboration`, comprehends a list of pool elements (defined as `Collaboration`) preceded by a `CollaborationAction` (which corresponds to a transition label l of the formal semantics, used to keep track of the evolution of the collaboration) enclosed in curly brackets. Its definition is:

```
op {_}_: CollaborationAction Collaboration -> ActCollaboration.
```

Using the syntax reported above, we provide below the implementation of the minimal example in Maude.

```
collaboration(
 pool( "Customer" ,
  proc(
   {emptyAction}
   start( enabled , "e1" . 0 ) |
   task( disabled ,  "e1" . 0 , "e2" . 0, "Check Offer"  ) |
   end( "e2" . 0 )
  ) , in: emptyMsgSet , out: emptyMsgSet
 )
) .
```

According to the presented syntax, we implemented the semantics by means of rewriting rules and conditional rewriting rules, which we write respectively in the following forms.

```
rl [Label] : Term-1 => Term-2 .
crl [Label] : Term-1 => Term-2 if Condition-1 /\.../\ Condition-N .
```

Its meaning is:

- the keyword `rl` stands for rewriting rule, while `crl` for conditional rewriting rule;
- in the square brackets we report the `Label` (i.e., the name) assigned to the rule;
- `Term-1 => Term-2` specifies that `Term-1` will be rewritten as `Term-2`;
- the `if` statement is present only in conditional rewriting rules and requires that all the listed conditions (which are in fact composed by means of the logical conjunction operator `/\`) are verified before rewriting `Term-1` in `Term-2`.

The rewriting rule corresponding to the operational rule *E-Start* (shown in Fig. 8) is:

```
rl [E-Start] : start( enabled , IEName . IEToken )
=> {tUpd(emptyEdgeSet , IEName . IEToken)}
start( disabled , IEName . increaseToken( IEToken ) ) .
```

Its meaning is:

- `E-Start` is the label associated to the rewriting rule;
- `start( enabled , IEName . IEToken )` is the element on which the rule acts. It is a `start` element with `Status` set to `enabled` and with an `Edge` composed by a name and a token;
- the `=>` symbol is followed by the result of the rewriting rule application. It is a `start` element with `Status` set to `disabled` and with an `Edge` which has the same name and a token value increased by one (by means of the `increaseToken` function). In addition, the start element is preceded by a 'token update' `Action` (corresponding to the transition label of the operational rule) of the form `{tUpd(emptyEdgeSet , IEName . IEToken)}`. This `Action` is the implementation of an internal action $(-\tilde{e_1}, +\tilde{e_2})$ described in Section 4. The term `emptyEdgeSet` is defined as a constant that represents an empty set of edges. In this case the label means that no token has to be removed (since a start event does not have input edges) while a token has to be added to the edge `IEName`.

The implementation of the rewriting rules that acts on a task (shown in Fig. 11) are reported below. The implementation of rule *T-Enable* is:

```
crl [T-Enable] :
task( disabled , IEName . IEToken , OEName . OEToken , TName )
=>
{tUpd(IEName . IEToken , emptyEdgeSet)}
task( enabled , IEName . decreaseToken( IEToken ) ,
        OEName . OEToken , TName )
if IEToken > 0 .
```

Its meaning is:

- `T-Enable` is the label associated to the rewriting rule;
- the element on which the rule acts is a `task` element named `TName` with `Status` set to `disabled` and with an input and output edges both composed by a name and a token;
- the result of the rewriting rule application is a `task` element with `Status` set to `enabled`, with the same output edge and name, but with the input edge marked by the token value decreased by one (via the `decreaseToken` function). To avoid negative token values, as in the original operational rule, the rewriting rule can be applied only if the token value before the decrease is greater than zero. In addition, the produced task element is preceded by an `Action` in the form {`tUpd(IEName . IEToken , emptyEdgeSet)`}. The {`tUpd(IEName . IEToken , emptyEdgeSet)`} is the implementation of an internal action $(-\vec{e_1}, +\vec{e_2})$ described in Section 4. In this case, the label means that a token has to be removed from the edge `IEName` while no token has to be added.

Below, the implementation of rules *T-Running* and *T-Complete*.

```
rl [T-Running] :
task(enabled , IEName . IEToken , OEName . OEToken , TName)
=>
{running(TName)}
task(running , IEName . IEToken , OEName . OEToken , TName).

rl [T-Complete] :
task(running , IEName . IEToken , OEName . OEToken , TName)
=>
{completed(TName)}
task(completed , IEName . IEToken , OEName . OEToken , TName).
```

Their meaning is:

- `T-Running` and `T-Complete` are the labels associated to the rewriting rules;
- the element on which the rules acts is a `task` element with `Status` set to `enabled` for rule `T-Running` and set to `running` for rule `T-Complete`;
- the result of the rewriting rule application is a `task` element with `Status` set to `running` for rule `T-Running` and set to `completed` for rule `T-Complete`, with the same input edge, output edge and task name. In addition, the task element is preceded by an `Action` in the form {`running(TName)`} (representing the internal action *running t*) and {`completed(TName)`} (representing *completed t*) for rules `T-Running` and `T-Complete`, respectively.

Below, the implementation of rule *T-Proceed*.

```
rl [T-Proceed] :
task( completed , IEName . IEToken , OEName . OEToken , TName)
=>
{tUpd(emptyEdgeSet , OEName . OEToken)}
task( disabled , IEName . IEToken ,
        OEName . increaseToken( OEToken ) , TName )
```

Its meaning is:

- `T-Proceed` is the label associated to the rewriting rule;
- the element on which the rule acts is a `task` element with `Status` set to `completed`;
- the result of the rewriting rule application is a `task` element with `Status` set to `disabled`, with the same input edge and task name, but with the token value marking the output edge increased by one. As usual, the task element is preceded by an `Action` of the form {`tUpd(emptyEdgeSet , OEName . OEToken)`}.

The implementation of the rewriting rule that acts on an end event (shown in Fig. 14) is:

```
crl [E-End] :
end( IEName . IEToken )
=>
{tUpd(IEName . IEToken , emptyEdgeSet)}
end( IEName . decreaseToken( IEToken ) )
if IEToken > 0 .
```

Its meaning is:

- `E-End` is the label associated to the rewriting rule;

- the element on which the rule acts is an `end` element;
- the result of the rewriting rule application consists of decreasing the token value on the edge of is the `end` element, provided that the value is greater than zero. An `Action` of the form {`tUpd(IEName . IEToken, emptyEdgeSet)`} is produced.

The rules in charge of updating the marking of tokens along all the process elements are the ones reported in Fig. 14. In the following we just consider *N-MarkingUpd* and the *N-Interleaving*. The implementation of the former rule consists of the following code.

```
crl [N-MarkingUpd] :
ProcElem1 | ProcElem2
=>
{tUpd(Edges1 , Edges2)}
ProcElem1' | markingUpdate(ProcElem2,tUpd(Edges1 , Edges2))
if ProcElem1 => {tUpd(Edges1 , Edges2)}ProcElem1' .
```

Its meaning is:

- `N-MarkingUpd` is the label associated to the rewriting rule;
- `ProcElem1 | ProcElem2` is the term on which the rule acts. It is the composition of two process elements with sort `ProcElement` composed by means of the | operation (a `ProcElement` can be a single process element or a composition of more process elements);
- the result of the rewriting rule application is the composition of `ProcElem1'`, i.e. the continuation of process `ProcElem1`, and the process resulting from the application of the `markingUpdate` operation to `ProcElem2` (which takes care of applying the marking changes due to the action performed by `ProcElem1`). In addition, the result is preceded by the same action that `ProcElem1` has performed.
- the rule condition requires that `ProcElem1` evolves to `ProcElem1'` doing an action involving movements of tokens.

The `markingUpdate` operation is implemented by the following code. It updates the `EdgeTokens` of all the elements that have an `Edge` included in the sets `Edges1` or `Edges2`.

```
op markingUpdate(_,_) : ProcElement InternalAction -> ProcElement .
```

The implementation of *N-Interleaving* consists of the following code.

```
crl [N-Interleaving] :
ProcElem1 | ProcElem2
=>
{Action1}
(ProcElem1' | ProcElem2 )
if ProcElem1=>{Action1}ProcElem1'/\ isInterleaving(Action1) .
```

Its meaning is:

- `N-Interleaving` is the label associated to the rewriting rule;
- `ProcElem1 | ProcElem2` is the composition of processes on which the rule acts;
- the result of the rewriting rule application is the composition of the `ProcElem1` continuation process and and `ProcElem2` (unchanged);
- the condition for applying the rewrite rule requires that `ProcElem1` evolves to `ProcElem1'` doing an interleaving action, i.e. an action different from a marking update and a kill.

Finally, the only rule at the collaboration level that enters in the game in the minimal example is *C-Internal*, whose implementation consists of the following code.

```
crl [C-Internal] :
pool(OrgName1,proc({Action1}ProcElem1),
        in:inputMsgSet ,out:outputMsgSet)
=>
{collab(OrgName1 , Action1')}
pool(OrgName1,proc({Action1'}ProcElem1'),
        in:inputMsgSet ,out:outputMsgSet)
if ProcElem1 => {Action1'}ProcElem1' /\ isInternal(Action1') .
```

Its meaning is:

- `C-Internal` is the label associated to the rewriting rule;
- the element on which the rule acts is a pool with name `OrgName1` and process `ProcElem1`;

- the result of the rewriting rule application is the evolution of the pool's process to `ProcElem1'`;
- the condition for applying the rewrite rule requires that `ProcElem1` evolves to `ProcElem1'` doing an internal action, i.e. an action different from the exchange of a message.

6.2. Properties verification

The Maude implementation of BPMN presented so far allows to verify the properties presented in Section 5 by using the Maude LTL model checker. In order to verify an LTL formula ϕ over a collaboration model, say `CollaborationModel`, the Maude command to execute is the following:

```
red modelCheck(CollaborationModel,φ) .
```

The command `red`, in combination with the operator `modelCheck`, invokes the LTL model checker based on the reduction facilities of Maude.

To enable the use of the model checker provided by Maude, we have to provide the implementation of the recurrent subformulae introduced in Section 5. In particular, they are expressed in Maude as equationally-defined computable state predicates. Indeed, each state of the LTS under analysis corresponds to a marked collaboration, representing a specific step of the collaboration execution. By way of example, the Maude implementation of the **aBPstarts** predicate is as follows (besides the case below, the definition of the predicate includes a case concerning the start message event, which is omitted as it is similar).

```
eq collaboration(
  {CollAction1}
  pool(OrgName,
   proc({Action1}ProcElements1 | start(enabled,EName.EToken)),
   in: inputMsgSet , out: outputMsgSet)
  | Coll1)
 |= aBPstarts = true .
```

The verification carried out on the running example has been performed on a Virtual Machine located into a beta version of the OCP cloud platform[12]; the machine runs the operating system Ubuntu-14.04 64 bits and it has 4 VCPU, and 8 GB of RAM. The complete `CollaborationModel` used for the analysis is reported in Appendix B. Notably, in case of Properties 2, 4, 6, 8, 9, and 12 the Maude model checker returns a counterexample showing the execution trace leading to a state that falsifies the property. For reader convenience, in Fig. 18 we graphically report the counterexample resulting from checking Property 8, by highlighting it with thick sequence edges and grey colored tasks.

7. Related work

Much effort has been devoted to the formalization of business processes. In most of the cases such formalizations support formal verification of business process models. Different surveys have been published in this area. Morimoto [36] provided a general overview of business process formal verification. Some years later, Groefsema and Bucur [37] provided a comprehensive overview with reference to process correctness, business compliance and process variability verification, while Fellmann et al. [38] proposed a survey on process compliance.

In this section we refer to the most relevant attempts that inspired our work. We first consider the other direct formalizations available in the literature, and then we discuss some mappings from BPMN to other well-known formalisms. Finally, we present relevant works in the area of verification for domain-specific models.

7.1. BPMN direct formalization

With regard to direct formalizations, we refer to Van Gorp and Dijkman [39], Christiansen et al. [22], El-Saber and Boronat [40], and Borger and Thalheim [41]. Among them, our contribution was mainly inspired by the one presented in [39]. The authors propose a BPMN 2.0 formalization based on in-place graph transformation rules; these rules are defined to be passed as input to the GrGen.NET tool, and are documented visually using BPMN syntax. With respect to our work, the used formalization techniques are different, since we provide an operational semantics in terms of LTS, which allows us to apply verification techniques well-established for this underlying model, such as model checking. The definition of an operational semantics gives us the possibility to be tool independent rather than be constrained to tools specific for graph transformation rules. This is confirmed by the same authors that for using their BPMN formalization in the compliance verification between global and local process models need a further transformation [42].

Another interesting work is described in [22], where Christiansen et al. propose a direct formalization of the BPMN 2.0 Beta 1 Specification using algorithms based on incrementally updated data structures. The semantics is given for BPMNinc,

[12] Open City Platform, http://www.opencityplatform.eu/.

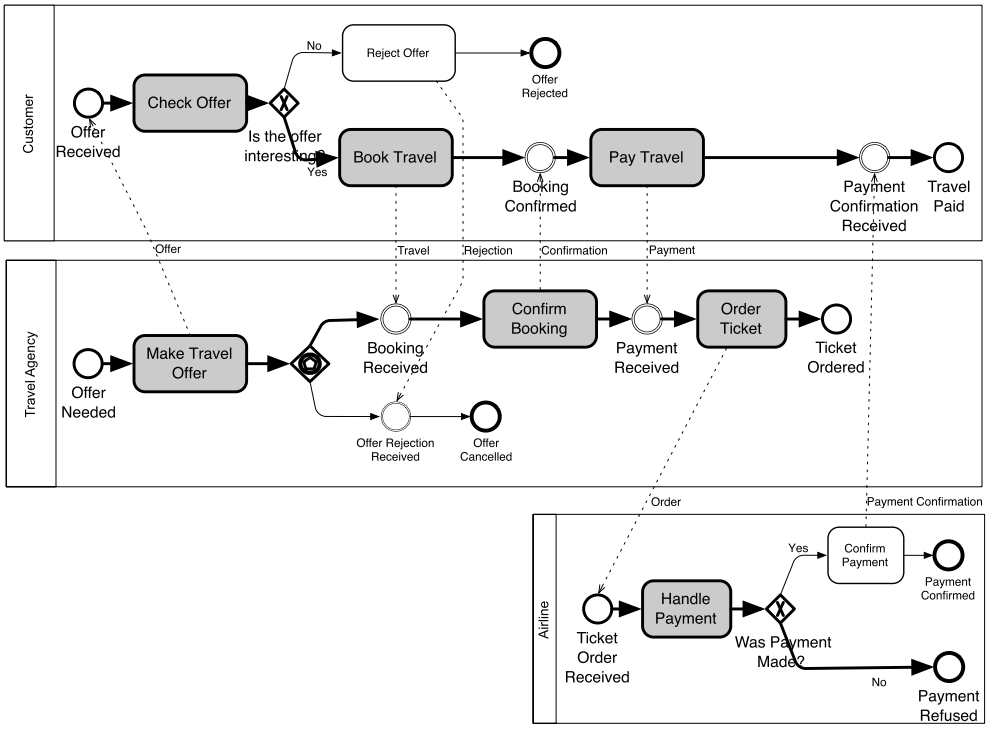


Fig. 18. Property 8 – counterexample.

that is a minimal subset of BPMN 2.0 containing just inclusive and exclusive gateways, start and end events, and sequence edges. This work differs from ours with respect to the formalization method, as it proposes a token-based semantics à la Petri Nets, while we define an operational semantics with a compositional approach à la process calculi. Moreover, the work in [22] also lacks to take into account BPMN organizational aspects and the flow of messages, whose treatment is a main contribution of our work.

El-Saber and Boronat proposed in [40] a formal characterization of well-formed BPMN processes in terms of rewriting logic, using Maude as supporting tool. They also prove the soundness of the well-formed BPMN models without introducing verification, which is postponed as future work. This formalization refers to a subset of the BPMN specification considering elements that are used regularly, such as flow nodes, data elements, connecting flow elements, artefacts, and swimlanes. Interesting it is also the mechanism given to represent and evaluate guard conditions in decision gateways. Differently from the other direct formalizations, this approach can be only applied to well-structured processes. Concerning the well-structuredness requirement, we are aware that enforcing such restriction may have benefits, among which we refer to the importance of structuredness as a guideline to avoid errors in business process modeling [43]. But we are also aware that this requirement may result in a language more complex to use and less expressive [21]. We therefore consider the arbitrary topology as a benefit, because we assume that designers should be free to model the process according to the reality they feel without needing to define well-structured models. In addition, it should be considered that not all process models with an arbitrary topology can be transformed into equivalent well-structured processes [11,12]. Moreover, the work in [40] has in common with our work a Maude implementation. However, while we implemented a formal definition of the syntax and semantics of BPMN, using the standard approach in [33], El-Saber and Boronat provide an ad-hoc implementation of BPMN directly in Maude, without resorting to any formal treatment in terms of operational semantics.

Another direct formalization is those proposed by Borger and Thalheim in [41], they define an extensible semantical framework for business process modeling notations using Abstract State Machines as method. The proposed formalization given by Borger and Thalheim is based on the version 1.0 of BPMN, which does not include notation meta-model and give more freedom to the authors in the interpretation of the language. Few years later, Kossak et al. [44] propose a Abstract State Machines based semantics for BPMN 2.0 single process diagrams; differently from our proposal where collaboration aspects, such as pools and message exchange, play a key role, in [44] they are overlooked.

7.2. BPMN formalization via mapping

The most common formalizations of BPMN are given via mappings to various formalisms, such as Petri Nets [6,45–49], or their extensions such as YAWL [50,51] and ECATNets [52], and process calculi [53–58]. Some approach also translate processes into a model checker input language, e.g. Masalagiu et al. [59] verify BPMN by translating it (via a Petri Net

intermediate model) into the model checker input language TLA+. This kind of formalizations suffers the typical problems introduced by a mapping. In fact, in these cases the semantics of BPMN is not given in terms of features and constructs of the language, but in terms of low-level details of their encodings. This makes the verification of BPMN models less effective, because the verification results refer to the low-level implementation of the models and may be difficult to be interpreted at BPMN level. Moreover, no formal proof of the correctness of these encodings with respect to a direct semantics of BPMN is provided.

Regarding the mapping from BPMN to Petri Nets, the one proposed by Dijkman et al. in [6] is probably the most relevant contribution. It enables the use of standard tools for process analysis, in order to check absence of deadlock and proper completion of BPMN models. However, the approach proposed by Dijkman et al. is based on the version 1.1 of BPMN and as the authors stated it suffers from deficiencies that impact on the proposed formalization. Moreover, differently from our approach, even if the mapping deals with messages, it does not properly consider multiple organization scenarios, and does not provide information to the analysis phase regarding who are the participants involved in the exchange of messages. In the area of Petri Nets-based approaches, another relevant work is the one by Awad et al. [49]. The novelty of this paper is on the definition and use of a modeling language for graphically specifying compliance rules on BPMN, named BPMN-Q [60]. Differently from our approach, the paper deals with verification of compliance rules imposed on internal business processes. Moreover, there are also approaches that are based on Petri Nets and provide solutions for process collaborations, such as [61]. Unfortunately, such approaches do not deal with the BPMN specification and its peculiarities. In principle, we do not envisage major hurdles in extending the translations from BPMN to Petri Nets available in the literature in order to achieve results similar to ours. However, we have preferred to develop a direct semantics because extending available translations may result in generation of convoluted and large Petri Nets, thus undermining the understanding of the formal meaning of the BPMN execution semantics, and the verification of BPMN collaborations.

Other relevant mappings are those from BPMN to YAWL, a language with a strictly defined execution semantics inspired by Petri Nets and able to support verification [62]. Among the proposed mappings, we would like to mention the ones by Ye and Song [50] and Dumas et al. [51]. The former is defined under the well-formedness assumption, which instead we do not rely on. Moreover, although messages are taken into account in the mapping, pools and lanes are not considered; thus it is not possible to identify who is the sender and who is the receiver in the communication. This results in the lack of capability to introduce verification at message level considering the involved organizations. The latter mapping, instead, formalizes a very small portion of BPMN elements. In particular, limitations about pools and messages are similar to the previous approach: pools are treated as separate business processes, while messages flow is not covered by the mapping.

More recently Kheldoun et al. [52] propose a mapping from BPMN to Recursive ECATNets, which can be expressed in term of conditional re-writing logic and given in input to the Maude LTL model checker. Even if we use the same model checker, the approach in [52] suffers from the mapping problems discussed above and, in particular, it does not consider in the mapping messages as well as the event-based gateway.

Process calculi has been also considered as means for formalizing BPMN. Among the others, Wong and Gibbons presented in [54,53] a translation from a subset of BPMN process diagrams, under the assumption of well-formedness, to a CSP-like language based on Z notation. This enables the introduction of a formal verification to check properties based on the notion of messages, like consistency between BPMN diagrams with different levels of abstraction and compatibility between participants within a business collaboration [63]. Messages are also considered by Arbab et al. in [55], where the main BPMN modeling primitives are represented by means of the coordination language Reo. Differently from the other mappings, this one considers a significantly larger set of BPMN elements. The proposed formalism admits analysis using model checking and bisimulation techniques in expressing compliance concerns. Prandi et al., instead, defined in [56] a semantics in term of a mapping from BPMN to the process calculus COWS [64], which has been specifically devised for modeling service-oriented systems. The analysis is based on the use of the probabilistic model checker PRISM on a case study. Last but not least, also π-calculus was taken as target language of a mapping by Puhlmann and Weske [57,58]. They also provide a characterizations of soundness properties using bisimulation equivalence. Even if our proposal differs from the above ones, as it is a direct semantics rather than a mapping, it has drawn inspiration from those based on process calculi for the use of a compositional approach in the SOS style.

7.3. Verification of domain-specific models

Nowadays, verification has been recognized as an important research topic and there is a very extensive research activity toward verification in different application domains. Here in the following we focus on some of them, such as service oriented computing [65–69], web applications [70–72], social networks [73–75], and software product line [76–79].

First we refer to service oriented computing, since it is usually considered to be the reference technological implementation for BPMN. In particular, Fantechi et al. [65] inspired our paper in stressing the importance of message exchange and kill operators that are specific characteristics in service oriented computing also relevant for BPMN. In particular, the proposed verification approach verifies formulae expressed in the branching-time temporal logic and relies on process algebras designed specifically for service-oriented computing, such as the COWS. Moreover, Foster et al. [66] contribute with a model-based approach to verify web service compositions. Differently from us, the models to be checked are BPEL4WS implementations rather than BPMN.

An increasing number of web applications and the continuous evolution of technologies used in the development of such applications have led to the introduction of formal verification also to design and to maintain high quality web applications. Halle et al. [70] use model checking to verify temporal properties against the control flow of a web application. As well as in our approach, in this case communication properties have been considered. Differently from us, Halle et al. refer to such properties as uncontrolled use of web browser's navigation functionalities.

We also consider the domain of social networks, where modeling and verification are promising techniques, as social networks are characterized by continuous interactive collaborations among people. Detecting malicious users, behaving in unconventional ways, contributes to avoid bad communications. In particular, Maggi et al. [73] propose a formal language able to describe interactions among Twitter accounts. Similarly to our work, such a language, called Twitlang, is implemented in the form of a Maude interpreter. Together with the Maude model checker, they make possible to automatically verify interaction properties of Twitter accounts.

Finally, software product lines development can be also considered as a relevant domain where verification can be applied. Product line systems are designed as families aiming at promoting a systematic reuse of development artefacts. In this regards, ter Beek et al. [76] apply symbolic model checking based on NuSMV. Results show that model checking software product lines is challenging because of the exponential number of products to be checked, even if its application into practice results to be even more a need. This would justify the application in the future of model checking on scenarios where software product lines are combined with BPMN, such as in [80], [81].

Summing up, besides the specific commonalities and differences discussed so far, our approach follows the same methodology of the related works mentioned above, where a formal semantics is used to enable verification. On the other hand, the other approaches mainly differ from ours for the use of ad-hoc, domain-specific formalisms, specifically devised or chosen for dealing with given application domains. Differently from them, in fact, we rely on a widely accepted OMG standard modeling language, which allows us to deal with all application domains that fall under the umbrella of Business Processes.

8. Concluding remarks

The lack of a well-established formal semantics for BPMN was the main driver of our work. This is a critical point of the BPMN specification, considering the wide adoption of the language both from the industry and the research community. In this paper, we define an operational semantics of BPMN, and provide an implementation enabling the verification of properties of collaboration models. We summarize below the key aspects of our proposal, as well as its assumptions and limitations. We then conclude by touching upon directions for future work.

Discussion. Our BPMN operational semantics focuses on the collaboration capability supported by message exchange. The proposed formalization enables designers to freely specify their processes with an arbitrary topology, supporting the adherence to the standard without the requirement of defining well-structured models. We also provide a Maude implementation of the semantics, where the operational rules are rendered in terms of rewriting rules. This enables the exploration of the evolution of BPMN collaborations, and permits to exploit the analysis toolset provided by Maude. In this way, we are able to reason on collaborations and, in particular, we can verify their properties by means of the built-in LTL model checker of Maude. Such properties can deal with task execution, occurrence of events, process termination, presence of deadlocks, as well as, message exchange, which is a distinctive aspect of our work.

On the other hand, it is worth noticing that not all specifications allowed by the syntax in Figs. 5 and 6 are meaningful. Indeed, in a general term of the language generated by the grammar, there could be, e.g., a message edge outgoing from a pool that is not matched by a corresponding message edge incoming into another pool, or similarly for a sequence edge of a given process. Of course, these kinds of situations do not occur on BPMN models, where edges are not split into incoming and outgoing parts. To discard such malformed specifications, we resort to the notion of well-definedness (see Section 3). This assumption should not be thought of as a limitation, but rather as a means to ensure that we are dealing with terms corresponding to BPMN models. Still concerning assumptions on the syntactic structure of specifications, differently from most works in the literature, we do not require specifications to be well-structured (see Section 2.4). Thus, they can have an arbitrary topology. This means that, e.g., besides the standard form of iteration expressed as a loop involving a pair of XOR gateways, we admit any kind of loop, included also the 'bad' ones that can generate infinite tokens along the same sequence edge (see, e.g., the model fragment in page 44). This results in an LTS with infinite states, which cannot be dealt by our verification approach, as Maude and its model checker do not provide any related facilities. Notably, our syntax does not allow self-loops on single process nodes, however they can be easily rendered by enclosing nodes in standard XOR-based loops. The challenge of normalizing BPMN models with arbitrary topology using an operational semantics consists of maintaining a compositional approach while using unstructured constructs. Indeed, well-structured processes can be expressed in terms of structured control-flow blocks, that can be simply combined by nesting them. This permits avoiding the token-based semantics, as the semantics of blocks can be defined similarly to that of control-flow constructs of standard imperative programming languages. Unstructured constructs can be instead freely connected via edges. This formalization approach thus enables the analysis of a larger set of BPMN models.

As mentioned in Section 2.1, to keep our formalization manageable, we have limited our syntax to the core elements of BPMN. In particular, we have left out such aspects and constructs as timed events, data objects, sub-processing, error handling, and multiple instances. Considering the dataset we have drawn from the BPM Academic Initiative repository (see

Section 2.4), our syntax allows to deal with 1336 models out of the 7541 of the dataset. Extending our BPMN formalization would affect its syntax, semantics and implementation. For example, extending it with multiple instances would require: (i) adding specific syntactic constructs (multiple instance activities and multi-instance pools), (ii) extending the semantics to deal with process instantiation and message correlation, (iii) extending the Maude implementation accordingly, also dealing with scalability issues that may arise. This extension would demand for a significative effort, hence it is out of the scope of this paper. Finally, as we exploit Maude for implementing the semantics and enabling automated verification, we inherit its performance characteristics. This may become a limitation when dealing with large size models. Anyway, the purpose of the implementation presented in this paper is mainly demonstrative. In fact, verification performance is not our primary goal, while instead we are interested in the faithfulness of the implementation with respect to our formal definition.

A distinctive aspect of our approach is the use of a direct semantics. As discussed in Section 7.2, approaches relying on mappings aim at exploiting the potentiality of well-known formalisms, such as Petri Nets. However, they may be misguided by the modeling constructs and mechanisms provided by the target formalism. This makes these approaches prone to oversimplification neglecting BPMN peculiarities, such as processes with arbitrary topology, asynchronous communication model, external non-determinism in event-based gateway, detailed task lifecycle, and different forms of process completion. Our direct semantics, instead, aims at normalizing BPMN features as close as possible to their definition in the standard specification, without any bias from the use of another formalism. A faithful semantics, of course, ensures a more effective verification of model properties.

Future work. As a future work, we plan to continue our program to effectively support modeling and verification of BPMN collaborations. Specifically, we are pursuing both a theoretical and a practical line of research. In the former case, we aim at extending our formalization to model more BPMN elements. In particular, we intend to focus on tricky issues concerning multiple instances of the same process and the data management. Moreover, since our formalization also takes into account those BPMN models that produce LTSs with infinite states, we intend to investigate the integration in our Maude implementation of existing solutions (such as abstract interpretation techniques [82]) to address the infinite states issue.

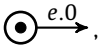
As a more practical work, we are developing a tool chain integrating our Maude-based verification environment with the Eclipse BPMN Modeller.[13] This offers the possibility of going back and forth between the modeling environment and the verification one, by e.g. graphically visualizing on the BPMN model the feedbacks of the verification. We also intend to integrate in our Eclipse-based tool an automatic code generation facility driven by our formal semantics. This will permit to produce, e.g., the Java code corresponding to a BPMN model, in order to support the enactment phase of the business process lifecycle.

Moreover, since the Maude implementation of our semantics provides an interpreter for BPMN collaborations, it may be practically exploited for checking if the semantics implemented by existing BPMN tools (e.g., steppers, simulators, enactment tools) is compliant with ours.

Appendix A. Proofs

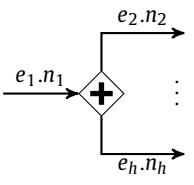
Property 1. *Let P be a process in a marked collaboration, if $P \overset{\alpha}{\longmapsto} P'$ then $P\dagger = P'\dagger$.*

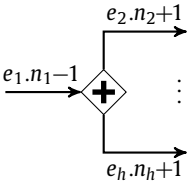
Proof. We proceed by induction on the derivation of $P \overset{\alpha}{\longmapsto} P'$. Base cases (for the sake of brevity, we show here the most interesting cases among the 30 base cases):

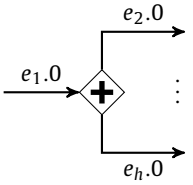
- (*E-Start*): We have that $P =$ 

, $P' =$ 

and $\alpha = +e$. By definition of killing function, we have that $P\dagger = P'\dagger =$ 

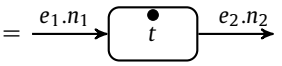
that allows us to conclude.
- (*G-AndSplit*): We have that $P =$ 

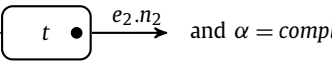
, $P' =$ 

and $\alpha = (-e_1, +\{e_2, \ldots, e_h\})$. By definition of killing function, we have that $P\dagger = P'\dagger =$ 

that allows us to conclude.

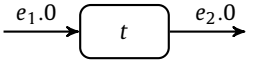
[13] http://www.eclipse.org/bpmn2-modeler/.

- (*T-Complete*): We have that $P =$ 

, $P' =$ 

and $\alpha = \textit{completed } t$.

  By definition of killing function, we have that

  $P\dagger = P'\dagger =$ 

that allows us to conclude.

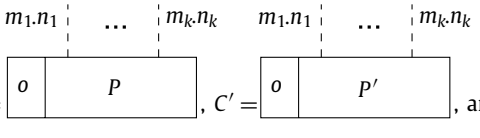
Inductive cases:

- (*N-MarkingUpd*): We have that $P = (P_1 \quad P_2)$, $P' = (P'_1 \quad P'_2)$, $P'_2 = P_2 \cdot (-\tilde{e_1}, +\tilde{e_2})$, and $\alpha = (-\tilde{e_1}, +\tilde{e_2})$. By definition of killing function, we have that $P\dagger = (P_1 \quad P_2)\dagger = P_1\dagger \quad P_2\dagger$, and $P'\dagger = (P'_1 \quad P'_2)\dagger = P'_1\dagger \quad P'_2\dagger$. By induction, we have that $P_1\dagger = P'_1\dagger$. By definition of the marking updating function, we have that P_2 and P'_2 differ for the number of tokens marking the edges in $\tilde{e_1}$ and $\tilde{e_2}$. The application of the killing function removes all tokens in the edges in $\tilde{e_1}$ and $\tilde{e_2}$, thus removing the differences between the two processes, i.e. $P_2\dagger = P'_2\dagger$. This, together with the result $P_1\dagger = P'_1\dagger$ shown above, permits to conclude that $(P_1\dagger \quad P_2\dagger) = (P'_1\dagger \quad P'_2\dagger)$ which is our thesis.
- (*N-Kill*): We have that $P = (P_1 \quad P_2)$, $P' = (P'_1 \quad P'_2)$, $P'_2 = P_2\dagger$, and $\alpha = \textit{kill}$. By induction, we have that $P_1\dagger = P'_1\dagger$. Moreover, we have that $P_2\dagger = P'_2\dagger$, as a second application of the killing function on a process has no effect according to the definition of the killing function. This, like in the previous case, permits to conclude.
- (*N-Interleaving*): We have that $P = (P_1 \quad P_2)$ and $P' = (P'_1 \quad P_2)$. By induction, we have that $P_1\dagger = P'_1\dagger$. Like in the previous case, this directly permits to conclude since the transition does not affect P_2. □

Property 2. *Let C be a collaboration, if $C \overset{l}{\longmapsto} C'$ then $C\dagger = C'\dagger$.*

Proof. We proceed by induction on the derivation of $C \overset{l}{\longmapsto} C'$.

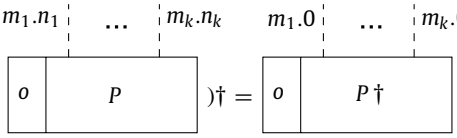
Base cases:

- (*C-Internal*) We have that $C =$ 

, $C' =$

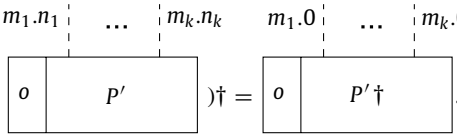
, and $l = o : \tau$.

  By definition of the killing function we have that

  $C\dagger = ($ 

$)\dagger =$

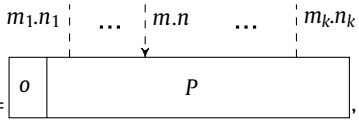
  and

  $C'\dagger = ($ 

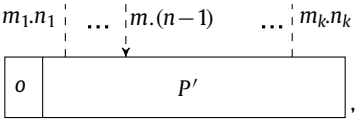
$)\dagger =$

.

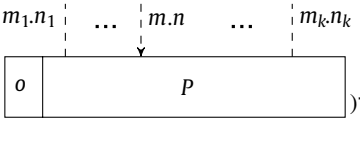
  According to the rule premise, we have that $P \overset{\tau}{\longmapsto} P'$, from which, by Property 1, we obtain $P\dagger = P'\dagger$. From this, we conclude that $C\dagger = C'\dagger$.

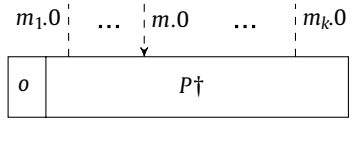
- (*C-Receive*) We have that $C =$ 

,

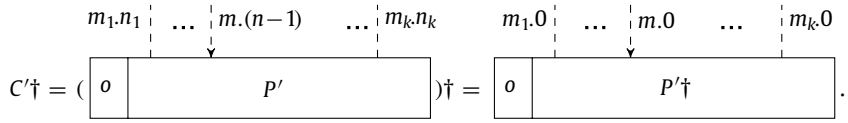
  $C' =$ 

, and $l = o :?m$. By definition of the killing function we have that

  $C\dagger = ($ 

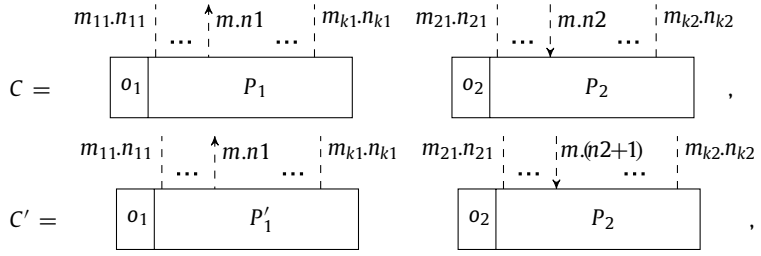
$)\dagger =$ 

and



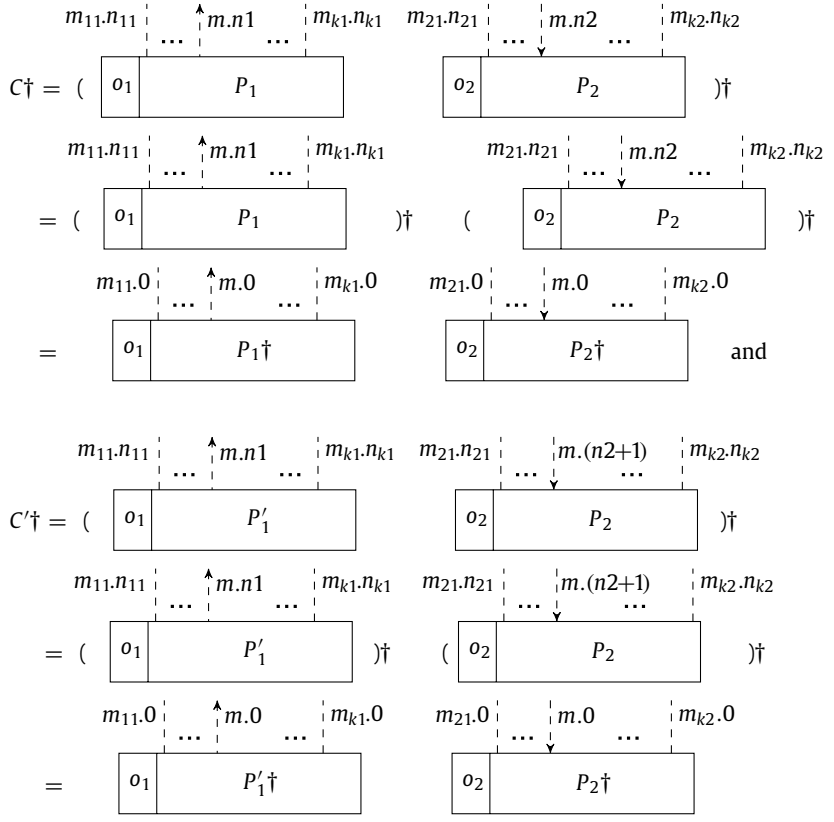
According to the rule premise, we have that $P \xmapsto{?m} P'$, from which, by Property 1, we obtain $P\dagger = P'\dagger$ that allows us to conclude.

- (*C-Deliver*) We have that



and $l = o_1 \to o_2 : m$.

By definition of the killing function we have that



According to the rule premise, we have that $P_1 \xmapsto{!m} P'_1$, from which, by Property 1, we obtain $P_1\dagger = P'_1\dagger$ that allows us to conclude.

Inductive case:

- (*C-Interleaving*) We have that $C = C_1C_2$, and $C' = C'_1C_2$. By definition of the killing function, we have that $C\dagger = (C_1C_2)\dagger = C_1\dagger\, C_2\dagger$ and $C'\dagger = (C'_1C_2)\dagger = C'_1\dagger\, C_2\dagger$. By induction we have that $C_1\dagger = C'_1\dagger$, which directly permits to conclude. □

Appendix B. Airline collaboration example in Maude

```
collaboration(
pool( "Customer" ,
proc( {emptyAction}
startRcv(enabled , "SequenceFlow_11" . 0 , "Offer" .msg 0) |
task( disabled , "SequenceFlow_11" . 0 , "SequenceFlow_12" . 0 , "Check Offer") |
xorSplit( "SequenceFlow_12" . 0 , edges( "SequenceFlow_13" . 0 and "SequenceFlow_14" . 0 ) ) |
taskSnd( disabled , "SequenceFlow_13" . 0 , "SequenceFlow_15" . 0 , "Rejection" .msg 0 , "Reject Offer") |
end( "SequenceFlow_15" . 0 ) |
taskSnd( disabled , "SequenceFlow_14" . 0 , "SequenceFlow_16" . 0 , "Travel" .msg 0 , "Book Travel") |
interRcv( disabled , "SequenceFlow_16" . 0 , "SequenceFlow_17" . 0 , "Confirmation" .msg 0 ) |
taskSnd( disabled , "SequenceFlow_17" . 0 , "SequenceFlow_18" . 0 , "Payment" .msg 0 , "Pay Travel") |
interRcv( disabled , "SequenceFlow_18" . 0 , "SequenceFlow_19" . 0 , "Payment Confirmation" .msg 0 ) |
end( "SequenceFlow_19" . 0 )
) ,
in: "Offer" .msg 0 andmsg "Confirmation" .msg 0 andmsg "Payment Confirmation" .msg 0 andmsg emptyMsgSet ,
out: "Rejection" .msg 0 andmsg "Travel" .msg 0 andmsg "Payment" .msg 0 andmsg emptyMsgSet
) |
pool( "Travel Agency" ,
proc( {emptyAction}
start( enabled , "SequenceFlow_21" . 0 ) |
taskSnd( disabled , "SequenceFlow_21" . 0 , "SequenceFlow_22" . 0 , "Offer" .msg 0 , "Make Travel Offer") |
eventSplit( "SequenceFlow_22" . 0 , eventRcvSplit(
eventInterRcv( disabled , "SequenceFlow_23" . 0 , "Travel" .msg 0 )  |
eventInterRcv( disabled , "SequenceFlow_24" . 0 , "Rejection" .msg 0 ) ) ) |
taskSnd( disabled , "SequenceFlow_23" . 0 , "SequenceFlow_25" . 0 , "Confirmation" .msg 0 , "Confirm Booking") |
interRcv( disabled , "SequenceFlow_25" . 0 , "SequenceFlow_26" . 0 , "Payment" .msg 0 ) |
taskSnd( disabled , "SequenceFlow_26" . 0 , "SequenceFlow_27" . 0 , "Order" .msg 0 , "Order Ticket") |
end( "SequenceFlow_27" . 0 ) |
end( "SequenceFlow_24" . 0 )
) ,
in: "Travel" .msg 0 andmsg "Rejection" .msg 0 andmsg "Payment" .msg 0 andmsg emptyMsgSet ,
out: "Offer" .msg 0 andmsg "Confirmation" .msg 0 andmsg "Order" .msg 0 andmsg emptyMsgSet
) |
pool( "Airline Process" ,
proc( {emptyAction}
startRcv(enabled , "SequenceFlow_30" . 0 , "Order" .msg 0) |
task( disabled , "SequenceFlow_30" . 0 , "SequenceFlow_31" . 0 , "Handle Payment") |
xorSplit( "SequenceFlow_31" . 0 , edges( "SequenceFlow_32" . 0 and "SequenceFlow_33" . 0 ) ) |
taskSnd( disabled , "SequenceFlow_32" . 0 , "SequenceFlow_34" . 0 ,"Payment Confirmation" .msg 0 ,"Confirm Payment")|
end( "SequenceFlow_34" . 0 ) |
end( "SequenceFlow_33" . 0 )
) ,
in: "Order" .msg 0 andmsg  emptyMsgSet ,
out: "Payment Confirmation" .msg 0 andmsg  emptyMsgSet
)
)
```

References

[1] J.A. Zachman, A framework for information systems architecture, IBM Syst. J. 26 (3) (1987) 276–292.
[2] A. Lindsay, D. Downs, K. Lunn, Business processes—attempts to find a definition, Inf. Softw. Technol. 45 (15) (2003) 1015–1019.
[3] OMG, Business Process Model and Notation (BPMN v 2.0), 2011.
[4] M.z. Muehlen, J. Recker, How much language is enough? Theoretical and practical use of the business process modeling notation, in: Advanced Information Systems Engineering, vol. 5074, Springer, Berlin, Heidelberg, 2008, pp. 465–479.
[5] R. Breu, S. Dustdar, J. Eder, C. Huemer, G. Kappel, J. Kopke, P. Langer, J. Mangler, J. Mendling, G. Neumann, S. Rinderle-Ma, S. Schulte, S. Sobernig, B. Weber, Towards Living Inter-Organizational Processes, IEEE, 2013, pp. 363–366.
[6] R.M. Dijkman, M. Dumas, C. Ouyang, Semantics and analysis of business process models in BPMN, Inf. Softw. Technol. 50 (12) (2008) 1281–1294.
[7] M. Dumas, M. La Rosa, J. Mendling, R. Mäesalu, H.A. Reijers, N. Semenenko, Understanding business process models: the costs and benefits of structuredness, in: Advanced Information Systems Engineering, Springer, 2012, pp. 31–46.
[8] J. Dehnert, A. Zimmermann, On the suitability of correctness criteria for business process models, in: Business Process Management, Springer, 2005, pp. 386–391.
[9] W.M. van der Aalst, Workflow verification: finding control-flow errors using Petri-net-based techniques, in: Business Process Management, Springer, 2000, pp. 161–183.
[10] A. Polyvyanyy, C. Bussler, The structured phase of concurrency, in: Seminal Contributions to Information Systems Engineering, Springer, 2013, pp. 257–263.
[11] A. Polyvyanyy, L. García-Bañuelos, M. Dumas, Structuring acyclic process models, Inf. Syst. 37 (6) (2012) 518–538.
[12] A. Polyvyanyy, L. Garcia-Banuelos, D. Fahland, M. Weske, Maximal structuring of acyclic process models, Comput. J. 57 (1) (2014) 12–35.
[13] OMG, Business Process Model and Notation (BPMN V 2.0) by Example, 2011.
[14] M. Kunze, P. Berger, M. Weske, BPM Academic Initiative – Fostering Empirical Research, Tallinn, Estonia, 2012, pp. 1–5.
[15] M. Kunze, A. Luebbe, M. Weidlich, M. Weske, Towards understanding process modeling—the case of the BPM academic initiative, in: Business Process Model and Notation, Springer, 2011, pp. 44–58.
[16] G.D. Plotkin, A structural approach to operational semantics, J. Log. Algebraic Program. 60 (61) (2004) 17–139.
[17] J. Mendling, H.A. Reijers, W.M. van der Aalst, Seven process modeling guidelines (7pmg), Inf. Softw. Technol. 52 (2) (2010) 127–136.
[18] F. Corradini, A. Ferrari, F. Fornari, S. Gnesi, A. Polini, B. Re, G.O. Spagnolo, A guidelines framework for understandable BPMN models, Data Knowl. Eng. 113 (2018) 129–154.
[19] M. Dumas, M. La Rosa, J. Mendling, H.A. Reijers, Fundamentals of Business Process Management, Springer, 2013.
[20] T.M. Prinz, Fast soundness verification of workflow graphs, in: ZEUS, in: LNCS, vol. 1029, Springer, 2013, pp. 31–38.
[21] B. Kiepuszewski, A.H.M. ter Hofstede, C.J. Bussler, On Structured Workflow Modelling, Springer, 2000, pp. 431–445.

[22] D.R. Christiansen, M. Carbone, T. Hildebrandt, Formal semantics and implementation of BPMN 2.0 inclusive gateways, in: Web Services and Formal Methods, vol. 6551, Springer, Berlin, Heidelberg, 2011, pp. 146–160.
[23] B. Gfeller, H. Völzer, G. Wilmsmann, Faster or-join enactment for BPMN 2.0, in: W. van der Aalst, J. Mylopoulos, M. Rosemann, M.J. Shaw, C. Szyperski, R. Dijkman, J. Hofstetter, J. Koehler (Eds.), Business Process Model and Notation, vol. 95, Springer, Berlin, Heidelberg, 2011, pp. 31–43.
[24] M. Dumas, A. Grosskopf, T. Hettel, M. Wynn, Semantics of standard process models with OR-joins, in: R. Meersman, Z. Tari (Eds.), On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, vol. 4803, Springer, Berlin, Heidelberg, 2007, pp. 41–58.
[25] F. Corradini, C. Muzi, B. Re, L. Rossi, F. Tiezzi, Global vs. local semantics of BPMN 2.0 OR-join, in: 44th International Conference on Current Trends in Theory and Practice of Computer Science, in: Lecture Notes in Computer Science, vol. 10706, Springer, 2018, pp. 321–336.
[26] F. Corradini, A. Polini, B. Re, F. Tiezzi, An operational semantics of BPMN collaboration, in: FACS, vol. 9539, Springer, 2015, pp. 161–180.
[27] F.-R. Sinot, Call-by-name and call-by-value as token-passing interaction nets, in: Typed Lambda Calculi and Applications, Springer, 2005, pp. 386–400.
[28] F. Kirchner, F.-R. Sinot, Rule-based operational semantics for an imperative language, Electronic Notes in Theoretical Computer Science 174 (1) (2007) 35–47.
[29] A. Pnueli, The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science, 1977, IEEE, 1977, pp. 46–57.
[30] S. Eker, J. Meseguer, A. Sridharanarayanan, The Maude LTL model checker, Electronic Notes in Theoretical Computer Science 71 (2004) 162–187.
[31] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude-a High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic, Springer-Verlag, 2007.
[32] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theor. Comput. Sci. 96 (1) (1992) 73–155.
[33] A. Verdejo, N. Martí-Oliet, Implementing CCS in Maude 2, Electronic Notes in Theoretical Computer Science 71 (2004) 282–300.
[34] R. Milner, Communication and Concurrency, Prentice-Hall, 1989.
[35] F. Corradini, F. Fornari, A. Polini, B. Re, F. Tiezzi, A. Vandin, BProVe: a formal verification framework for business process models, in: 32nd IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, 2017, pp. 217–228.
[36] S. Morimoto, A survey of formal verification for business process modeling, in: Computational Science – ICCS 2008, vol. 5102, Springer, Berlin, Heidelberg, 2008, pp. 514–522.
[37] H. Groefsema, D. Bucur, A survey of formal business process verification: from soundness to variability, in: International Symposium on Business Modeling and Software Design, 2013, pp. 198–203.
[38] M. Fellman, A. Zasada, State-of-the-art of business process compliance approaches: a survey, in: Proceedings of the 22nd European Conference on Information Systems, ECIS, 2014.
[39] P. Van Gorp, R. Dijkman, A visual token-based formalization of BPMN 2.0 based on in-place transformations, Inf. Softw. Technol. 55 (2) (2013) 365–394.
[40] N. El-Saber, A. Boronat, BPMN Formalization and Verification Using Maude, ACM Press, 2014, pp. 1–12.
[41] E. Börger, B. Thalheim, A method for verifiable and validatable business process modeling, in: Advances in Software Engineering, vol. 5316, Springer, Berlin, Heidelberg, 2008, pp. 59–115.
[42] P.M. Kwantes, P. Van Gorp, J. Kleijn, A. Rensink, Towards compliance verification between global and local process models, in: F. Parisi-Presicce, B. Westfechtel (Eds.), Graph Transformation, vol. 9151, Springer, Cham, 2015, pp. 221–236.
[43] R. Laue, J. Mendling, The impact of structuredness on error probability of process models, in: Information Systems and e-Business Technologies, vol. 5, Springer, Berlin, Heidelberg, 2008, pp. 585–590.
[44] F. Kossak, C. Illibauer, V. Geist, J. Kubovy, C. Natschläger, T. Ziebermayr, T. Kopetzky, B. Freudenthaler, K.-D. Schewe, A Rigorous Semantics for BPMN 2.0 Process Diagrams, Springer, Cham, 2014, pp. 29–152.
[45] W. Huai, X. Liu, H. Sun, Towards Trustworthy Composite Service Through Business Process Model Verification, 2010, pp. 422–427.
[46] R. Koniewski, A. Dzielinski, K. Amborski, Use of petri nets and business processes management notation in modelling and simulation of multimodal logistics chains, in: 20th European Conference on Modeling and Simulation, Warsaw, 2006, pp. 28–31.
[47] M. Ramadan, H.G. Elmongui, R. Hassan, BPMN formalisation using coloured petri nets, in: International Conference on Software Engineering & Applications, 2011.
[48] A. Awad, G. Decker, N. Lohmann, Diagnosing and repairing data anomalies in process models, in: Business Process Management Workshops, Springer, 2010, pp. 5–16.
[49] A. Awad, G. Decker, M. Weske, Efficient compliance checking using bpmn-q and temporal logic, in: Business Process Management, Springer, 2008, pp. 326–341.
[50] J. Ye, W. Song, J. Ye, W. Song, Transformation of BPMN diagrams to YAWL nets, J. Softw. 5 (4) (2010) 396–404.
[51] G. Decker, R. Dijkman, M. Dumas, L. García-Bañuelos, Transforming BPMN diagrams into YAWL nets, in: Business Process Management, Springer, 2008, pp. 386–389.
[52] A. Kheldoun, K. Barkaoui, M. Ioualalen, Specification and verification of complex business processes – a high-level Petri net-based approach, in: H.R. Motahari-Nezhad, J. Recker, M. Weidlich (Eds.), Business Process Management, vol. 9253, Springer International Publishing, Cham, 2015, pp. 55–71.
[53] P.Y. Wong, J. Gibbons, Formalisations and applications of BPMN, Sci. Comput. Program. 76 (8) (2011) 633–650.
[54] P.Y.H. Wong, J. Gibbons, A process semantics for BPMN, in: Formal Methods and Software Engineering, vol. 5256, Springer, Berlin, Heidelberg, 2008, pp. 355–374.
[55] F. Arbab, N. Kokash, S. Meng, Towards using reo for compliance aware business process modeling, in: Leveraging Applications of Formal Methods, Verification and Validation, in: CCIS, vol. 17, Springer, 2008, pp. 108–123.
[56] D. Prandi, P. Quaglia, N. Zannone, Formal analysis of BPMN via a translation into COWS, in: Coordination Models and Languages, Springer, 2008, pp. 249–263.
[57] F. Puhlmann, M. Weske, Investigations on soundness regarding lazy activities, in: Business Process Management, vol. 4102, Springer, Berlin, Heidelberg, 2006, pp. 145–160.
[58] F. Puhlmann, Soundness verification of business processes specified in the pi-calculus, in: On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, vol. 4803, Springer, Berlin, Heidelberg, 2007, pp. 6–23.
[59] C. Masalagiu, W.-N. Chin, Ş. Andrei, V. Alaiba, A rigorous methodology for specification and verification of business processes, Form. Asp. Comput. 21 (5) (2009) 495–510.
[60] A. Awad, Bpmn-q: a language to query business processes, in: Proceedings of EMISA'07, 2007, pp. 115–128.
[61] J. Roa, O. Chiotti, P. Villarreal, A verification method for collaborative business processes, in: International Conference on Business Process Management, in: LNBIP, vol. 99, Springer, 2011, pp. 293–305.
[62] M.T. Wynn, H.M.W. Verbeek, W.M. van der Aalst, A.H. ter Hofstede, D. Edmond, Business process verification-finally a reality!, Bus. Process Manag. J. 15 (1) (2009) 74–92.
[63] P.Y.H. Wong, J. Gibbons, Verifying Business Process Compatibility (Short Paper), IEEE, 2008, pp. 126–131.
[64] R. Pugliese, F. Tiezzi, A calculus for orchestration of web services, J. Appl. Log. 10 (1) (2012) 2–31.
[65] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, F. Tiezzi, A logical verification methodology for service-oriented computing, ACM Trans. Softw. Eng. Methodol. 21 (3) (2012) 1–46.
[66] H. Foster, S. Uchitel, J. Magee, J. Kramer, Model-based verification of web service compositions, in: Proceedings of the 18th IEEE International Conference on Automated Software Engineering, 2003, IEEE, 2003, pp. 152–161.

[67] L. Caires, H.T. Vieira, Analysis of service oriented software systems with the conversation calculus, in: Proceedings of the 7th International Conference on Formal Aspects of Component Software, in: LNCS, Springer, 2012, pp. 6–33.
[68] A. Tarasyuk, E. Troubitsyna, L. Laibinis, Formal modelling and verification of service-oriented systems in probabilistic event-b, in: Proceedings of the 9th International Conference on Integrated Formal Methods, in: LNCS, vol. 7321, Springer, 2012, pp. 237–252.
[69] L. Baresi, R. Heckel, S. Thöne, D. Varró, Modeling and validation of service-oriented architectures: application vs. style, in: ACM SIGSOFT Software Engineering Notes, vol. 28, ACM, 2003, pp. 68–77.
[70] S. Hallé, T. Ettema, C. Bunch, T. Bultan, Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ACM, 2010, pp. 235–244.
[71] M. Haydar, Formal framework for automated analysis and verification of web-based applications, in: Proceedings of the 19th IEEE International Conference on Automated Software Engineering, IEEE Computer Society, 2004, pp. 410–413.
[72] M.H. ter Beek, A. Lluch-Lafuente, Automated specification and verification of web-based applications, J. Log. Algebraic Methods Program. 87 (2017) 51.
[73] A. Maggi, M. Petrocchi, A. Spognardi, F. Tiezzi, A language-based approach to modelling and analysis of Twitter interactions, J. Log. Algebraic Methods Program. 87 (2017) 67–91.
[74] W.M.P. van der Aalst, H.T. de Beer, B.F. van Dongen, Process mining and verification of properties: an approach based on temporal logic, in: On the Move to Meaningful Internet Systems: CoopIS, DOA, and ODBASE, in: LNCS, vol. 3760, Springer, 2005, pp. 130–147.
[75] M. Haydar, H. Sahraoui, A. Petrenko, Specification patterns for formal web verification, in: Eighth International Conference on Web Engineering, 2008, ICWE'08, IEEE, 2008, pp. 240–246.
[76] M.H. ter Beek, A. Legay, A. Lluch-Lafuente, A. Vandin, Statistical model checking for product lines, in: Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques – 7th International Symposium, ISoLA 2016, in: LNCS, vol. 9952, 2016, pp. 114–133.
[77] K. Lauenroth, K. Pohl, S. Toehning, Model checking of domain artifacts in product line engineering, in: 24th IEEE/ACM International Conference on Automated Software Engineering, 2009, ASE'09, IEEE, 2009, pp. 269–280.
[78] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, J.-F. Raskin, Model checking lots of systems: efficient verification of temporal properties in software product lines, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol. 1, ACM, 2010, pp. 335–344.
[79] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, Symbolic model checking of software product lines, in: Proceedings of the 33rd International Conference on Software Engineering, ACM, 2011, pp. 321–330.
[80] R. Cognini, F. Corradini, A. Polini, B. Re, Business process feature model: an approach to deal with variability of business processes, in: Domain-Specific Conceptual Modeling, Concepts, Methods and Tools, Springer, 2016, pp. 171–194.
[81] R. Cognini, F. Corradini, A. Polini, B. Re, Extending feature models to express variability in business process models, in: Advanced Information Systems Engineering Workshops, in: LNBIP, vol. 215, Springer, 2015, pp. 245–256.
[82] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: POPL, ACM, 1977, pp. 238–252.