A First-Order Logic Verification Framework for Communication-Parametric and Time-Aware BPMN Collaborations

Sara Houhou, Souheib Baarir, Pascal Poizat, Philippe Quéinnec, Laïd Kahloud

A First-Order Logic Verification Framework for Communication-Parametric and Time-Aware BPMN Collaborations

Sara Houhou[a,b,d,*], Souheib Baarir[a,b], Pascal Poizat[a,b,*], Philippe Quéinnec[c], Laid Kahloul[d]

[a] *Sorbonne Université, CNRS, LIP6, F-75005, Paris, France*
[b] *Université Paris Lumières, Université Paris Nanterre, F-92000, Nanterre, France*
[c] *IRIT - Université de Toulouse, F-31000 Toulouse, France*
[d] *Biskra University, LINFI Laboratory, Biskra, Algeria*

Abstract

The BPMN standard notation allows business process designers to model both intra-organizational processes and inter-organizational collaborations. A great effort has been devoted in proposing formal semantics for BPMN, and, fewer, in providing dedicated verification tools. Still, some advanced features of BPMN, namely communication or time-related constructs, are often set aside. This becomes an issue as BPMN gains interest outside of its original scope, e.g., for the IoT where communication and time play an important role. In this paper, we propose a formal semantics for a subset of BPMN. This semantics takes into account not only the usual gateways, but also sub-processes, inter-process communication, and time-related constructs. In contrast to transformational approaches, which give a semantics to BPMN by mapping it to some formal model (e.g., transition systems or Petri nets), our approach is based on a direct formalization in first-order logic that is then realized in a straightforward way into the TLA$^+$ formal language. We build on the TLA$^+$ model-checker, TLC, to provide process designers with a verification framework, fbpmn, that one may use to check BPMN and workflow specific properties. Our tools and our model database are open source and freely available online.

Keywords: BPMN, Formal Semantics, Collaboration, Communication, Time, Verification, First-Order Logic, TLA$^+$, Tool

1. Introduction

BPMN supports the modelling of the internal processes of organizations and of the way they interact to reach objectives. This modelling can be achieved through the use

☆This work was supported by project PARDI funded by the French National Agency for Research under grant ANR-16-CE25-0006.

*Corresponding author
Email addresses: Sara.houhou@lip6.fr (Sara Houhou), souheib.baarir@lip6.fr (Souheib Baarir), pascal.poizat@lip6.fr (Pascal Poizat), philippe.queinnec@irit.fr (Philippe Quéinnec), l.kahloul@univ-biskra.dz (Laid Kahloul)

of process and collaboration diagrams. In the standard [1], natural language is used to describe the BPMN execution semantics. This leaves room for interpretation and hampers the formal analyses that would be desirable in order to find defaults at design time rather than when running the processes and collaborations over business process engines. This issue has been addressed in the last decade in different proposals for a formalization of the BPMN execution semantics (see Section 7), some of them being supported with tools. However, these proposals often leave apart features related to communication and time. Meanwhile, BPMN is gaining interest as a modeling language in new areas, such as the Internet of Things (IoT) [2, 3]. There, taking into consideration the communication between the nodes of the system, the configuration of different communication modes, and time constraints such as durations and timeouts is a requirement.

Diving into BPMN. Let us now briefly introduce BPMN before presenting a motivating example in the next part. BPMN (Business Process Model and Notation) is a workflow based notation. It defines a set of element notations (see Fig. 2 for the part of them that we support). These elements can be classified into two types: flow objects (nodes), and connection objects (edges). In BPMN, nodes fall into four categories: *events*, *activities*, *gateways*, and *data objects* (not supported here). The events denote things that happen while a process is running. They affect the process flow and usually have a trigger or a result. Events may indicate the starting point of a process, identified using a *start event* or an ending point of a process, identified using an *end event*. BPMN also defines intermediate events which occur between start and end events. These events affect the process flow in the sense that they must occur for the process to go on. There are different types of intermediate events. In this work we focus on message-related and time-related ones. An intermediate event can be attached to the boundary of an activity, and is then called a *boundary event*. This is used to interrupt the activity or to launch activities in parallel, based on some condition (*e.g.*, a message reception or the deadline of a timeout). An activity is a unit of work that can be either an atomic *task* or a compound *sub-process*. Gateways are used to control the process flow and in particular the activity execution ordering. There are five main types of gateways in BPMN and we are taking into account the main ones [4]. The *exclusive* type (*XOR* in Fig. 2) is used to choose one out of a set of mutually exclusive alternative incoming or outgoing branches. It can also be used to represent looping behaviors. The *parallel* type (*AND* in Fig. 2) synchronizes concurrent flows for all its incoming branches and creates concurrent flows for all its outgoing branches. The *inclusive* type (*OR* in Fig. 2) states that any number of branches among its incoming or outgoing branches may be taken[1]. These three types of gateways can be merging, splitting, or mixed (both merging and splitting). We support all three cases. Finally, with the *event-based gateway* (*EB* in Fig. 2), the process flow is based on which of the events (or message reception tasks) that follow the gateway occurs first. Nodes are connected with edges that can be *sequence flows* (flows of control), or *message flows* (flows for message exchange). The sequence flow category can in turn be decomposed into normal sequence flows, *conditional sequence flows* (expressing the condition for some branch to be activated), and *default sequence flows* (the default branch to activate if all others, conditional ones, cannot be). BPMN defines three main kinds

[1]This is indeed a simplification for this introduction. The semantics as defined in the standard and hence as supported in our work is a bit more complex, see Section 2.2.

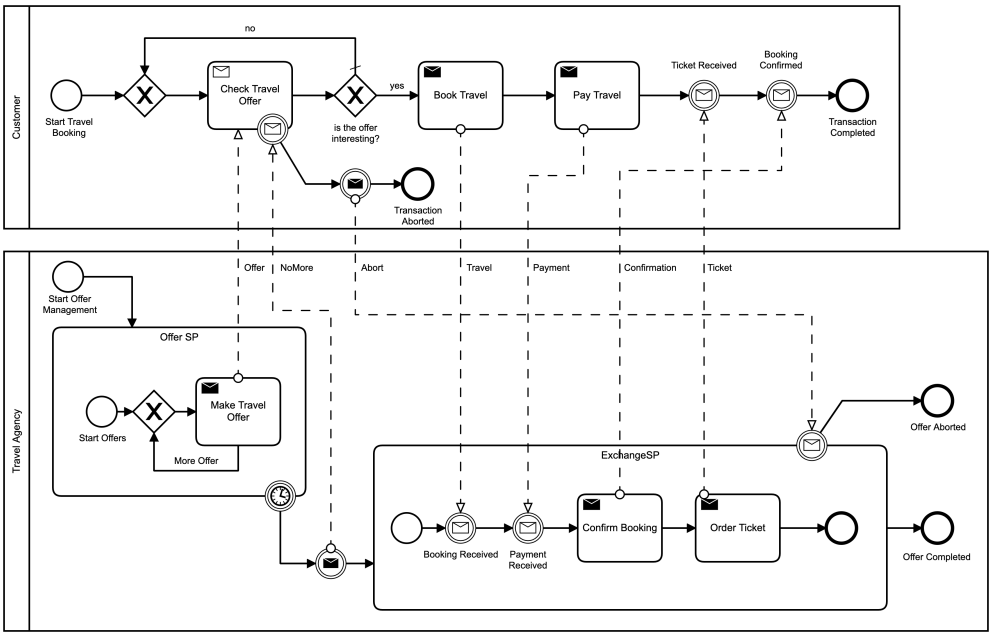

Figure 1: Travel agency case study (extended from an example in [5]).

of models: *process diagrams*, *collaboration diagrams*, and *choreography diagrams*. In this work, we deal with the first two. A process diagram is used to model the activities of a single organization. Collaboration diagrams can be defined with a set of different processes (for different organizations), exchanging messages and cooperating to reach a shared objective. In our work, we abstract away from data (data objects, data stores and message payloads). Therefore, in the sequel, message (resp. event instance) and message type (resp. event type) are used interchangeably. For more information BPMN, see [1].

Case study. Let us observe the collaboration model in Figure 1. It involves two participants: a customer and a travel agency. The *agency* sends offers to the client (loop with an exclusive gateway) in a first sub-process, and, after some time (time boundary event), begins a second sub-process to exchange information (booking, payment, confirmation, ticket) with the customer. The *customer* may reject some of the offers, and at some point he/she may agree on one (loop using two exclusive gateways). If so, he/she stops accepting offers and he/she sends booking and payment information to the agency and gets the corresponding ticket and confirmation. The agency and the customer rely on interrupting features to deal with the fact that the customer stops accepting offers as soon as he/she has agreed on one, while, as said before, the agency sends offers for some amount of time only. *If the customer has not agreed on an offer before the end of this amount of time*, the sending of offers by the agency will stop (timer boundary event on the first agency sub-process). The agency will send an interrupting message to the client (message boundary event on the customer offer reception task) which will in turn interrupt the exchange sub-process (message boundary event on the second agency sub-process). *If the customer has agreed on an offer before the end of the amount of*

time, he/she will begin to send information to the agency. The agency will timeout, send an interrupting message to the client (that will not be used), and enter the sub-process which will, this time, not be interrupted.

As one can see, the *interplay between communication, interrupting features, and time* makes the overall behavior of the collaboration difficult to grasp. Will the collaboration always reach one of its ends? Either that the client and the agency have agreed on an offer (them ending in Transaction Completed and Offer Completed, respectively) or that they have not but they are not waiting for some event to happen (them ending in Transaction Aborted and Offer Aborted, respectively). Will the collaboration reach one of its ends but with pending messages, that have been sent but neither received nor treated? Or, worse, will the collaboration deadlock at some point depending on the choices made by the customer and the agency, and the passage of time?

This corresponds indeed to checking *soundness properties* (to be further introduced in Sect. 4). One may note the different order in the customer and in the agency for the reception (resp. sending) of the confirmation and of the ticket. As we will see later on (in Section 4), the communication model used between the agency and the customer has an impact on this. With the verification framework we propose, the process designer will be able to check that the collaboration at hand cannot be ensured to be completely sound: for some executions there may be messages left untreated. But the process designer will also be able to check that, given the right choice of communication model (we analyse collaborations for seven possible communication models presented in Sect. 2.2.4), the collaboration is sound if the message treatment constraint is relaxed. The process designer will also see that for some communication models the collaboration even deadlocks, with the client not being able to reach one of its final events. Playing with our counter-example example animator (see Fig. 4) he/she will be able to correct the model. Here this means either changing the chosen communication model or changing the order in which the client or the agency exchanges information.

This case study is just one example of models that can be analyzed using our proposal. The analysis of other examples, including ones from the literature, is presented in Section 5, and models from our example repository are available at [6] under `/models`.

Contribution. The contribution of this paper is twofold. First, (1.) we provide a formalization of a subset of BPMN execution semantics that *supports sub-processes, communication, and time constructs*, and is *parametric with reference to the properties of the communication*. Second, (2.) we support this formalization with *tools that automatically perform the verification of correctness properties* for BPMN collaboration models, and that *animate counter examples when the properties are not satisfied*.

As far as (1.) is concerned, we have chosen to define a direct logic semantics for BPMN. We use First-Order Logic (FOL) with natural numbers, sets, and maps. Instead of using an intermediary formal model, *e.g.*, Petri nets or process algebra, this choice of a simple yet expressive framework enables one to get a formal semantics that is amenable to implementation in different formal frameworks while still being close to the semi-formal semantics of the standard (hence it can be related to it). Furthermore, with reference to Petri net token based semantics, *e.g.*, [7, 8] as shown in Table 6, our choice to rely on FOL enables us to directly support a non-local semantics for inclusive (OR) gateways, sub-process interruption, and a choice of several communication models. We implement our FOL semantics in TLA$^+$ [9] as a set of TLA$^+$ theories. This corresponds

to a pure syntactic transformation of FOL into the corresponding TLA$^+$ fragment, as demonstrated in Table 2.

Our semantics supports the seven point-to-point communication models that exist when considering local, causal and global message ordering, and it is easily extensible. As far as the subset of BPMN is concerned, we have first based our choice on the analysis of the 825 BPMN processes available in the BIT process library, release 2009 [10], given in [4]. We have then taken more constructs into account, mainly relative to our focus: creation and termination of processes based on messages or time, message and time-related intermediary events and boundary events (interrupting or non-interrupting), and event-based gateways. The subset of the notation that we support is given in Figure 2.

With respect to (2.), our approach relies on two steps. First, one uses the fbpmn tool that we have developed to get a TLA$^+$ representation of the model to verify. Then, one uses the TLC model checker from the TLA$^+$ tool-suite to perform the verification. The properties of interest are encoded in the TLA$^+$ theories we have implemented. They include usual correctness properties for workflows as well as ones (proposed more recently [5]) that are more specific to BPMN. Termination of verification for some of the properties (see Sect. 5.5) is ensured only for BPMN models with a finite state semantic model. If it is not the case, one can use constraints (see Sect. 6.4.1) to perform verification on a subset of the semantic model.

Both our fbpmn tool and the TLC tool are open source and freely available online. Furthermore, the models we have used for evaluation in Section 5 are also available online. To get the tools and the models, please see the fbpmn repository [6].

This work is based on the paper "*A First-Order Logic Semantics for Communication-Parametric BPMN Collaborations*" [11], extended as follows:

1. We have enriched the subset of BPMN taken into account and the formalization of its execution semantics into first-order logic to support time-related events and (message and time) boundary events, either interrupting or non-interrupting. We propose two different semantics for time and compare their benefits.
2. We have extended the six possible communication models to be used as parameters in the verification of a collaboration with a new one, relative to the causality of message emission. Furthermore, on top of these now seven generic communication models that are applied on a collaboration as a whole, we now support the definition and the use of ad-hoc communication models (a specific model built by assembling micro communication models that provide different constraints on sending and receiving messages).
3. Accordingly, we have extended our verification tool, fbpmn, to support these new elements and communication models. We have also extended fbpmn into a more comprehensive tool suite with a counter-example animator and a containerized Web application where process designers can model and verify BPMN processes or collaborations directly from the browser. This tool suite is now presented, including with reference to its extension as far as verification constraints, new properties, and new communication models are concerned.
4. The presentation is enriched with well-formedness rules, greater details on the semantics rules, the formalization in first-order logic of the communication models and of the properties that are verified, the presentation of the use of fairness for verification in presence of loops and alternatives, and an extended state of the art.

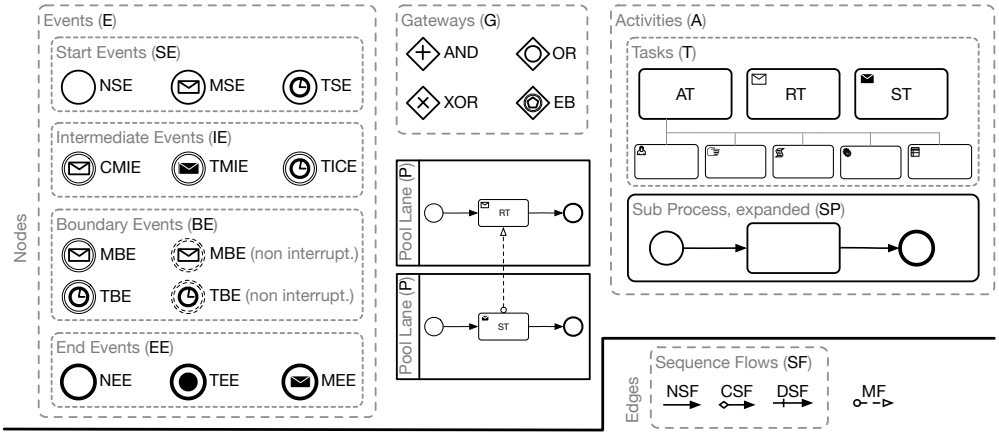

Figure 2: BPMN subset being supported in our work, with types used in the formalization (*e.g.*, MSE stands for Message Start Events and SE for Start Events).

5. We discuss the relation between the FOL semantics and its TLA$^+$ implementation.
6. We have a new and more complex and realistic case study, with the new BPMN constructs taken into account.

Overview. The formal part of the paper is developed in Section 2, with Sections 2.1 and 2.2, respectively addressing the presentation of the model underlying the semantics, and then the semantics itself, including seven possible communication models. Section 3 is devoted to taking into account time, with two possible semantics, either non-deterministic abstraction of time or based on explicit time, presented respectively in Sections 3.2 and 3.3. The properties of interest for the process designer are presented in Section 4. The implementation of the semantics in TLA$^+$, verification, and evaluation are presented in Section 5. This section also includes a short introduction to the TLA$^+$ language and verification framework, and a discussion on the use of fairness to deal with loops and alternatives in models. Section 6 is devoted to our tool suite, fbpmn, its architecture, general principles, and the way it can be easily extended. Related work is discussed in Section 7, and we end with conclusions and perspectives in Section 8.

2. Formal Semantics

In this section, we first present the model on which we base the definition of the communication-parametric semantics for BPMN collaborations. This model is used to represent collaborations as typed graphs. In a second step, we present the semantics itself. It follows the "token game" of the standard [1, Ch. 13], with a notion of state that evolves with activation and completion of graph nodes.

2.1. A Typed Graph Representation of BPMN Collaborations

In our work, a BPMN model is seen as a typed graph (Def. 2.1), where types corresponding to the BPMN syntax (see Fig. 2) are associated to nodes and edges.

- **For Nodes:**
  - The task type (T) groups the abstract task (AT), the receive task (RT), and the send task (ST) types. Formally: $T = \{AT, RT, ST\}$.
  - The activity type (A) groups the task and the sub-process (SP) types. Formally: $A = T \cup \{SP\}$.
  - The gateway type (G) groups the parallel (AND), the inclusive (OR), the exclusive (XOR), and the event-based (EB) gateway types. Formally: $G = \{AND, OR, XOR, EB\}$.
  - The start event type (SE) groups the none start event (NSE), the message start event (MSE), and the timer start event (TSE) types. Formally: $SE = \{NSE, MSE, TSE\}$.
  - The intermediate event type (IE) groups the catch message intermediate event ($CMIE$), the throw message intermediate event ($TMIE$), and the timer intermediate catch event ($TICE$) types. Formally: $IE = \{CMIE, TMIE, TICE\}$.
  - The boundary event type (BE) groups the message boundary event (MBE) and the timer boundary event (TBE) types. Formally: $BE = \{MBE, TBE\}$. Both indeed regroup interrupting and non-interrupting versions. A function, *isInterrupt* (Def. 2.1), is used to make the difference.
  - The end event type (EE) groups the none end event (NEE), the terminate end event (TEE), and the message end event (MEE) types. Formally: $EE = \{NEE, TEE, MEE\}$.
  - The event type (E) is the set of all event types. Formally: $E = SE \cup IE \cup BE \cup EE$.
- **For Edges:**
  - The sequence flow type (SF) groups the normal sequence flow (NSF), the conditional sequence flow (CSF), and the default sequence flow (DSF) types. Formally: $SF = \{NSF, CSF, DSF\}$.
  - The message flow type (MF), is used to denote message flows.

T_{Nodes} denotes the set of all node types, with an added type, P, to denote processes. Formally: $T_{Nodes} = A \cup G \cup E \cup \{P\}$. T_{Edges} denotes the set of all edge types. Formally: $T_{Edges} = SF \cup \{MF\}$. The hierarchical structure of collaborations, with processes P and sub-processes SP is dealt with by using specific types for nodes, and a relation, R, denoting containment. From our example, Figure 1, we would then have two nodes of type P (say n_1 for Customer, n_2 for Travel Agency), two nodes of type SP (say n_3 for Offer SP, n_4 for Exchange SP), $n_3 \in R(n_2)$, $n_4 \in R(n_2)$, and $n_5 \in R(n_3)$ with n_5 being the node of type ST for Make Travel Offer.

Notation (Restriction to types). Given a set of types T, a set X that whose elements can be typed using a function $cat_X : X \to T$, and a subset T' of T, we note $X^{T'} = \{x \in X \mid cat_X(x) \in T'\}$. By abuse of notation, we may write X^t, for some t in T, instead of $X^{\{t\}}$. In the sequel, we are mainly interested into two sets that are typed, N (nodes) typed using T_{Nodes}, and E (edges) typed using T_{Edges}. Accordingly, we will use N^T (resp. E^T) to denote the subset of nodes (resp. edges) of type T.

Definition 2.1 (BPMN Graph)**.** A BPMN graph is a tuple $\widehat{G} = (\ N,\ E,\ \mathbb{M},\ cat_N,\ cat_E,\ source,\ target,\ R,\ msg,\ attachedTo,\ isInterrupt)$ where:

- N, is the set of nodes,
- E $(N \cap E = \emptyset)$, is the set of edges,
- $\mathbb{M}$, is the set of message types,
- $cat_N : N \to T_{Nodes}$, gives the type of a node,
- $cat_E : E \to T_{Edges}$, gives the type of an edge,
- $source/target : E \to N$, give the source/target of an edge,
- $R : N^{\{P,SP\}} \to 2^{N \cup E}$, gives the set of nodes and edges which are directly contained in a container (process or sub-process),
- $msg : E^{MF} \to \mathbb{M}$ gives the message associated to a message flow,
- $attachedTo : N^{BE} \to N^{A}$, gives the activity to which a boundary event node is attached,
- $isInterrupt : N^{BE} \to Bool$, denotes whether a boundary event node is interrupting or not,

Notation. We note R^+ the transitive closure of R.

Auxiliary functions. For a graph $\widehat{G} = (\ N,\ E,\ \mathbb{M},\ cat_N,\ cat_E,\ source,\ target,\ R,\ msg_t,\ attachedTo$, we define the following auxiliary functions:

- $in/out : N \to 2^E$ give the incoming/outgoing edges of a node:

$$in(n) = \{e \in E \mid target(e) = n\}$$

$$out(n) = \{e \in E \mid source(e) = n\}$$

- a family of functions in^T (resp. out^T) : $N \to 2^E$ is used to get incoming/outgoing edges of a selected type T, $in^T(n) = in(n) \cap E^T$ and $out^T(n) = out(n) \cap E^T$.
- $procOf : N \to N^P$ gives the container process of a given node, $procOf(n) = p$ if and only if $n \in R^+(p)$.

It is desirable to enforce that models respect some well-formedness rules before performing verification. We therefore define well-formed BPMN graphs using well-formed condition rules.

Well-formed BPMN graph. A well-formed BPMN graph satisfies the following conditions. These rules are extracted from the standard [1]:

- (C1) No incoming sequence flow edges for start events: $\forall n \in N^{SE}, in^{SF}(n) = \emptyset$
- (C2) No outgoing sequence flow edges for end events: $\forall n \in N^{EE}, out^{SF}(n) = \emptyset$
- (C3) A sub-process contains exactly one None Start Event and no other start event types: $\forall n \in N^{SP}, |R(n) \cap N^{NSE}| = 1 \wedge R(n) \cap N^{\{MSE,TSE\}} = \emptyset$
- (C4) A sub-process node cannot contain a process node: $\forall n \in N^{SP}, R(n) \cap N^{P} = \emptyset$
- (C5) For each process node, we require that:
  - it contains at least one initial node: $\forall n \in N^{P}, R(n) \cap N^{SE} \neq \emptyset$;
  - it contains at least one end node: $\forall n \in N^{P}, R(n) \cap N^{EE} \neq \emptyset$.
- (C6) No looping edges: $\forall e \in E, source(e) \neq target(e)$
- (C7) No node isolation: $\forall n \notin N^{P} \implies in(n) \neq \emptyset \vee out(n) \neq \emptyset$
- (C8) A gateway that has a conditional edge must have a unique default edge:

$$\forall n \in N^{\{AND,XOR\}}, out^{CSF}(n) \neq \emptyset \implies |out^{DSF}(n)| = 1$$

- (C9) No incoming message flow for send task, message end event, throw message intermediate event: $\forall n \in N^{\{ST,MEE,TMIE\}}, in^{MF}(n) = \emptyset$
- (C10) No outgoing message flow for receive tasks, message start event, catch message intermediate event, boundary message intermediate event:

$$\forall n \in N^{\{RT,MSE,CMIE,MBE\}}, out^{MF}(n) = \emptyset$$

- (C11) A message flow edge connects two nodes of different processes:

$$\forall e \in E^{MF}, procOf(source(e)) \neq procOf(target(e))$$

- (C12) An event-based gateway must have two or more outgoing edges:

$$\forall n \in N^{EB}, |out^{SF}(n)| \geq 2$$

- (C13) Parallel and event-based gateways cannot have an outgoing edge of conditional sequence flow type: $\forall n \in N^{\{AND,EB\}}, (out^{SF}(n) \cap E^{CSF} = \emptyset)$
- (C14) Elements that follow an event-based gateway can only be catching intermediate message events, receive tasks, or timer intermediate catch events. Additionally, one cannot have both receive tasks and intermediate message events (see for example Fig. 31):

$$\begin{aligned} \forall n \in N^{EB} \quad & (\forall e \in out^{SF}(n), target(e) \in N^{\{CMIE,RT,TICE\}}) \\ & \wedge \begin{pmatrix} (\{e \in out^{SF}(n) \mid target(e) \in N^{RT}\} = \emptyset) \\ \vee\ (\{e \in out^{SF}(n) \mid target(e) \in N^{CMIE}\} = \emptyset) \end{pmatrix} \end{aligned}$$

- (C15) The outgoing edges of an inclusive or an exclusive gateway must be a combination between default sequence flows and conditional sequence flows, or all are of the normal sequence flow type:

$$\forall n \in N^{\{XOR,OR\}}, (\forall e \in out^{SF}(n), e \in E^{\{CSF,DSF\}}) \vee (\forall e \in out^{SF}(n), e \in E^{NSF})$$

- (C16) Message flows connect the throwing elements (send task, message end event, throw message intermediate event) with catching elements (receive task, message start event, catch message intermediate event, message boundary intermediate event):

$$\forall e \in E^{MF}, source(e) \in N^{\{ST,MEE,TMIE\}} \wedge target(e) \in N^{\{RT,MSE,CMIE,MBE\}}$$

- (C17) Message catching elements must have at least one incoming message flow edge: $\forall n \in N^{\{RT,MSE,CMIE,MBE\}},\ |in^{MF}(n)| \geq 1$

- (C18) Message throwing elements must have at least one outgoing message flow edge: $\forall n \in N^{\{ST,MEE,TMIE\}},\ |out^{MF}(n)| \geq 1$

It should be noted that, but for these rules, we do not require a specific structure of the BPMN graph. For example, we do not require these graphs to be well-balanced (when for each splitting gateway of a given type, there is a corresponding merging gateway of the same type). One can use a splitting exclusive gateway and merge its branches using a parallel gateway. Verification will be able to detect this is a erroneous model.

2.2. A FOL Semantics for BPMN Collaborations

In order to maintain traceability with the standard, we use a token-based approach to define the semantics. The movement of tokens is based on node types and on two predicates, starting predicate St and completing predicate Ct defined for each node type. These predicates correspond to the enabling of the node to start its execution, and the enabling of the node to complete its execution, respectively. Some nodes only have a start transition (*e.g.*, end events), and others only have a completion transition (*e.g.*, gateways). When a node defines only one of the two predicates, the other one is considered to be false. The semantics (Sect. 2.2.3) relies on a notion of *state* of the BPMN graph (Sect. 2.2.1) to define the St and Ct predicates. Further, the semantics is parameterized by a type T_{net} that encapsulates the properties of the communication network using an initialization function, *initnet*, and two predicates, *send* and *receive* (see Sect. 2.2.2 for the abstract definition and Sect. 2.2.4 for realizations of T_{net} corresponding to seven communication models).

2.2.1. State

A state of a BPMN graph gives a marking for the nodes and the edges, together with a state of the communication network.

Definition 2.2 (State)**.** The state of a BPMN graph is a tuple $s = (mn, me, mnet)$ such that:

- $mn : N \to \mathbb{N}$, is a function assigning a natural number marking to each node.

- $me : E \to \mathbb{N}$, is a function assigning a natural number marking to each edge.
- $mnet : T_{net}$, is the state of the communication network.

The set of all states of a BPMN graph is denoted by *States*.

Definition 2.3 (Initial state)**.** The initial state of a BPMN graph, denoted by $s_o = (mn_0, me_0, mnet_0)$, is such that:

- the start nodes of the processes hold a token, all other nodes are unmarked:

$$\forall n \in N, mn_0(n) = \begin{cases} 1 & if\ \exists p \in N^P, n \in N^{SE} \cap R(p) \\ 0 & otherwise. \end{cases}$$

- all edges are unmarked: $\forall e \in E, me_0(e) = 0$
- the network is empty: $mnet_0 = initnet$

2.2.2. Communication

The properties of communication between two participants (process nodes) for a given type of message are abstracted with an initialization function, *initnet*, and two transition predicates, *send* and *receive*. *initnet* is used to give the initial state of *mnet*. *send* and *receive* specify when a communication action is enabled and what effect it has on *mnet*. The value of *mnet* describes the state of the network in terms of what messages are sent but not yet received, as the network evolves through time. In essence we are modeling the pool of messages that have been sent but not yet received, possibly using a single universal queue, or by using channel-by-channel queues, or some other structures that carry information to allow and order send and receive events.

Several communication models are formally described in Section 2.2.4. For instance, with the Fifo All asynchronous communication model, messages must be delivered in the order they were sent. In this model, $send(p_1, p_2, m)$ is always enabled, and adds m to *mnet* ; $receive(p_1, p_2, m)$ is true only if m is the oldest message and thus the next one to be delivered, and the new state of *mnet* is its previous value minus m.

Definition 2.4 (Communication Model)**.** The communication model is characterized by a function $initnet : T_{net}$ and two predicates $send/receive : N^P \times N^P \times \mathbb{M}$.

2.2.3. Semantics

We define here the execution semantics of BPMN based on the above-mentioned St and Ct predicates, for each type of node in the BPMN graph, and based on its notion of state. The presentation of time-related constructs is deferred to Section 3 where we present two different semantics for time. In the semantics, let $s = (mn, me, mnet)$ and $s' = (mn', me', mnet')$ denote two states. Additionally, we consider the predicate $\triangle$ that denotes marking equality but for nodes and edges given as parameter. Hence, $\triangle(X)$ means "*nothing changes except for* X":

$$\triangle(X) \stackrel{def}{\equiv} \forall n \in N \setminus X, mn'(n) = mn(n) \wedge \forall e \in E \setminus X, me'(e) = me(e)$$

Similarly, Ξ denotes that the state of the network does not change: $\Xi \stackrel{def}{\equiv} mnet' = mnet$.

Start nodes. There are two starting node types for the instantiation of the process: the none start event (NSE), and the message start event (MSE).

The behavior of an NSE is defined only by a completing predicate. It consumes its token and generates one token on all of its outgoing sequence flow edges. If it is the initial node of a process p, it activates p by generating a token on it. When an NSE is defined within a sub-process p, its activation is conditioned by the activation of p.

Formally.

$$\forall n \in N^{NSE}, Ct(n) \stackrel{def}{\equiv} \begin{array}{l} (mn(n) = 1) \wedge (mn'(n) = mn(n) - 1) \\ \wedge\ \forall e \in out^{SF}(n), (me'(e) = me(e) + 1) \\ \wedge \left(\begin{array}{c} (\exists p \in N^{P}, (n \in R(p)) \wedge (mn(p) = 0) \\ \wedge\ (mn'(p) = 1) \wedge \triangle(\{n, p\} \cup out^{SF}(n)) \wedge \Xi) \\ \vee\ (\exists p \in N^{SP}, (n \in R(p)) \wedge \triangle(\{n\} \cup out^{SF}(n)) \wedge \Xi) \end{array} \right) \end{array}$$

Example. Figure 3 presents a process p with a none start event (nse), an abstract task ($task$), and a none end event (nee). It shows that there is a token within nse node which is represented by a green token. nse completes by consuming this token and produces one on P and its outgoing sequence flow ($e1$).

Figure 3: Completing behavior of a None Start Event. Before (left) and after (right) application of the Ct rule.

The behavior of a message start event (MSE) is defined by a completing predicate. An MSE is enabled if it has a token and there is a message offer on one of its incoming sequence flow edges. It completes by consuming the message, generating one token on all of its outgoing sequence flow edges, and activating the process p by generating a token on it.

Formally.

$$\begin{array}{l} \forall n \in N^{MSE}, Ct(n) \stackrel{def}{\equiv} (mn(n) = 1) \wedge (mn'(n) = mn(n) - 1) \\ \qquad \wedge\ \forall e \in out^{SF}(n), (me'(e) = me(e) + 1) \\ \qquad \wedge\ \exists em \in in^{MF}(n), (me(em) \geq 1) \wedge (me'(em) = me(em) - 1) \\ \qquad\quad \wedge\ receive(procOf(source(em)), procOf(n), msg(em)) \\ \qquad\quad \wedge\ \exists p \in N^{P}, n \in R(p) \wedge (mn(p) = 0) \wedge (mn'(p) = 1) \\ \qquad\qquad \wedge\ \triangle\ (\{n, p, em\} \cup out^{SF}(n)) \end{array}$$

Example. Consider again the example of Figure 3. By modifying the none start event to a message start event (mse), we get the model in Figure 4. It represents the complete execution semantics behavior. The left hand-side of Figure 4 shows that there is a token on the start node and a message offer ($m1$) on the incoming message flow edge of mse. This latter completes by consuming the message according to the chosen communication model, and producing a token on the process and on all its outgoing edges.

Figure 4: Completing behavior of a Message Start Event. Before (left) and after (right) application of the Ct rule.

Ending nodes. We have three ending node types for the termination of a process: none end event (NEE), terminate end event (EE), and message end event (MEE).

The behavior of a none end event (NEE) node is defined only by a starting predicate: it is enabled if it has at least one token on one of its incoming edges. It starts by consuming this token and adding one to itself.
Formally.

$$\forall n \in N^{NEE}, St(n) \stackrel{def}{\equiv} \exists e \in in^{SF}(n), (me(e) \geq 1) \wedge (me'(e) = me(e) - 1) \\ \wedge\ (mn'(n) = mn(n) + 1) \wedge \triangle(\{n, e\}) \wedge Xi$$

Example. Figure 5 presents the execution semantics of an end node (nee). The left-hand side of the figure shows the enabling of nee by the presence of a token on its incoming edge ($e2$). The right-hand side of the figure shows the starting behavior. It consumes the token from $e2$ and generates a token on it.

Figure 5: Starting behavior of None End Event. Before (left) and after (right) application of the St rule.

A terminate end event (TEE) node is defined only by a starting predicate: it is enabled if it has at least one token on one of its incoming edges. It behaves like a none end event by consuming a token from one of its incoming sequence flows and generates a token on itself. Besides, it does the additional work of dropping down all the remaining tokens of the process or sub-processes to which it belongs.
Formally.

$$\forall n \in N^{TEE}, St(n) \stackrel{def}{\equiv} \exists e \in in^{SF}(n), (me(e) \geq 1) \wedge (mn'(n) = 1) \\ \wedge\ \exists p \in N^{\{P,SP\}}, (n \in R(p)) \\ \wedge\ \forall nn \in ((R^{+}(p) \cap N) \setminus \{n\}), (mn'(nn) = 0) \\ \wedge\ \forall ee \in (R^{+}(p) \cap E), (me'(ee) = 0) \\ \wedge\ \triangle\,(R^{+}(p)) \wedge \Xi$$

Example. The left-hand side of Figure 6 shows the enabling of a terminate end node (tee) by the presence of a token on its incoming edge ($e7$). The right-hand side of the figure shows the starting execution of the node. It consumes the token from $e7$ and

generates a token on itself. Besides, it affects all the existing executions in parallel (here $e3$ and $e4$) by dropping down all their tokens.

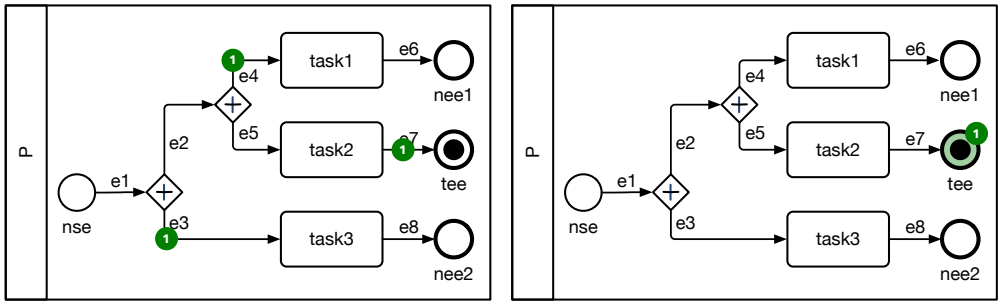

Figure 6: Starting behavior of a Terminate End Event. Before (left) and after (right) application of the St rule.

A message end event (MEE) is defined only by a starting predicate. It is enabled to start if it has a token on one of its incoming edges. It starts by moving the token from one of its incoming edges to itself, and sending a message on the network according to the communication model.

Formally.

$$\begin{aligned}\forall n \in N^{MEE}, St(n) \stackrel{def}{\equiv}\ & \exists e \in in^{SF}(n), (me(e) \geq 1) \wedge (me'(e) = me(e) - 1) \\ & \wedge\ (mn'(n) = mn(n) + 1) \\ & \wedge\ \exists ee \in out^{MF}(n), (me'(ee) = me(ee) + 1) \\ & \quad \wedge\ send(procOf(n), procOf(target(ee)), msg(ee)) \\ & \quad \wedge \triangle(\{n, e, ee\})\end{aligned}$$

Example. Figure 7 shows the starting behavior of the message end event (mee). It starts by consuming the token from the incoming edge ($e2$), producing a token on itself, and sending a message $m1$ on the network.

Figure 7: Starting behavior of a Message End Event. Before (left) and after (right) application of the St rule.

Activity nodes. Two kinds of activity nodes have to be taken into account: the abstract tasks (AT), and the sub-processes (SP).

The behavior of an abstract task node (AT) is defined by a starting and a completing predicate. An AT starts if at least one token is present on one of its incoming edges and it does not already own a token. It consumes a token from one of its incoming edges, and produces one on itself. An AT node is completed by consuming one token from itself,

and adding one token on each of its outgoing edges. Note that an abstract task may be ended by an interrupting boundary event (see *MBE* and *TBE* in pages 20 and 31).
Formally.

$$\forall n \in N^{AT}, St(n) \stackrel{def}{\equiv} \exists e \in in^{SF}(n), (me(e) \geq 1) \wedge (me'(e) = me(e) - 1) \\ \wedge\ (mn(n) = 0) \wedge (mn'(n) = mn(n) + 1) \\ \wedge \triangle\ (\{n, e\}) \wedge \Xi$$

$$\forall n \in N^{AT}, Ct(n) \stackrel{def}{\equiv} (mn(n) = 1) \wedge (mn'(n) = mn(n) - 1) \\ \wedge\ \forall e \in out^{SF}(n), (me'(e) = me(e) + 1) \\ \wedge \triangle\ (\{n\} \cup out^{SF}(n)) \wedge \Xi$$

Example. Figure 8 shows the starting behavior of the abstract task *task*. It starts by consuming the token from $e1$ and generating a token on itself.

Figure 8: Starting behavior of an Abstract Task activity. Before (left) and after (right) application of the St rule.

Figure 9 shows its completing behavior. It consumes the token from itself and generates one on its outgoing edge $e2$.

Figure 9: Completing behavior of an Abstract Task activity. Before (left) and after (right) application of the Ct rule.

The behavior of a sub-process activity *SP* node extends the one of an *AT* node with some additional conditions: when it is enabled, a sub-process adds a token to the start event it contains. It completes when at least one end event it contains has some tokens and none of its edges or non end event nodes are still active (*i.e.*, owning a token). Note that, alike an abstract task, a sub-process may also be ended by an interrupting boundary event (see *MBE* and *TBE* in pages 20 and 31).
Formally.

$$\forall n \in N^{SP}, St(n) \stackrel{def}{\equiv} \exists e \in in^{SF}(n), (me(e) \geq 1) \wedge (me'(e) = me(e) - 1) \\ \wedge\ (mn(n) = 0) \wedge (mn'(n) = mn(n) + 1) \\ \wedge\ \forall ns \in (N^{NSE} \cap R(n)), (mn'(ns) = mn(ns) + 1) \\ \wedge \triangle\ (\{e, n\} \cup (N^{NSE} \cap R(n))) \wedge \Xi$$

$$\forall n \in N^{SP}, Ct(n) \stackrel{def}{\equiv} (mn(n) = 1) \wedge (mn'(n) = mn(n) - 1)$$
$$\wedge\ \forall e \in R(n) \cap E, (me(e) = 0)$$
$$\wedge\ \exists n_{ee} \in (N^{EE} \cap R(n)), (mn(n_{ee}) \geq 1)$$
$$\wedge\ \forall nn \in R(n) \cap N, (mn(nn) \geq 1 \Rightarrow nn \in N^{EE})$$
$$\wedge\ \forall nn \in (R(n) \cap N^{EE}), (mn'(nn) = 0)$$
$$\wedge\ \forall e \in out^{SF}(n), (me'(e) = me(e) + 1)$$
$$\wedge\ \triangle\ (\{n\} \cup (R(n) \cap N^{EE}) \cup out^{SF}(n)) \wedge \Xi$$

Example. Figure 10 shows the starting behavior of the sub-process activity (SP). It starts by consuming a token from its incoming edge ($e1$) and generating a token on itself and on its start event ($nse1$).

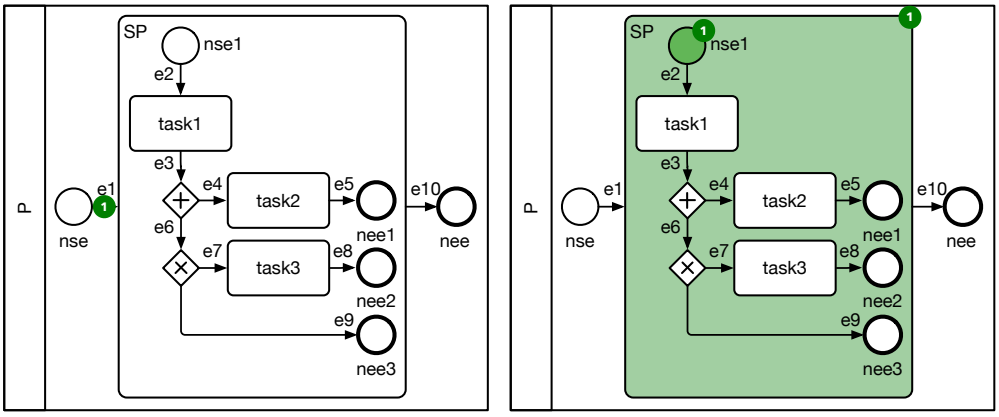

Figure 10: Starting behavior of a Sub-Process activity. Before (left) and after (right) application of the St rule.

Figure 11 shows that even if there is a token on one of the end events, here $nee3$, the sub-process can not execute its completing transition: to complete, SP must wait until the token on $e5$ has given place to one on $nee1$.

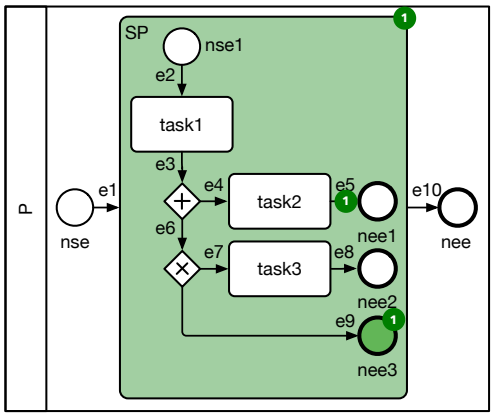

Figure 11: A Sub-Process activity not ready to complete: a token is still present on one of its edges.

Communication. The semantics for *MSE* and *MEE* have been presented above. The remaining communicating elements, *TMIE*, *CMIE*, *MBE*, *ST* and *RT* require additional conditions for starting and completing due to the presence of sending/reception predicates. All these elements require a token on one of their incoming edges to be enabled.

The *ST* and *RT* are enabled when they have a token on their incoming edges and they have no token on them. They start executing by moving this token inside. *ST* completes by sending a message on all its outgoing edges regarding the chosen communication model (which affects the network state) and producing a token on all its outgoing edges.

A *ST* is not necessarily instantaneous as a send may block, for instance with synchronous communication, RSC or a bounded size network. Another important point is that a boundary event, such as a timeout, can be attached to a receive or send task and not to an event. For all these reasons, we have chosen to make these tasks non-atomic. This allows to distinguish a send task from aThrowMessageIntermediateEvent, and a receive task from a CatchMessageIntermediateEvent. Note that the semantics in the standard is ambiguous, since there are two contradictory aspects: tasks in BPMN are non-atomic while the purpose of a send task is to only send a message, which is intrinsically atomic.

Formally.

$$\begin{aligned}\forall n \in N^{ST}, St(n) \stackrel{def}{\equiv}\ & \exists e \in in^{SF}(n), (me(e) \geq 1) \wedge (me'(e) = me(e) - 1) \\ & \wedge\ (mn(n) = 0) \wedge (mn'(n) = mn(n) + 1) \\ & \wedge \triangle(\{n, e\}) \wedge \Xi\end{aligned}$$

$$\begin{aligned}\forall n \in N^{ST}, Ct(n) \stackrel{def}{\equiv}\ & (mn(n) = 1) \wedge (mn'(n) = mn(n) - 1) \\ & \wedge\ \forall e \in out^{SF}(n), (me'(e) = me(e) + 1) \\ & \wedge\ \exists ee \in out^{MF}(n), (me'(ee) = me(ee) + 1) \\ & \wedge\ send(procOf(n), procOf(target(ee)), msg(ee)) \\ & \wedge \triangle(\{n, ee\} \cup out^{SF}(n))\end{aligned}$$

Example. Figure 12 presents the starting behavior of a send task activity (*task*). It shows the same starting behavior as the one presented in Figure 8.

Figure 12: Starting behavior of a Send Task activity. Before (left) and after (right) application of the *St* rule.

Fig. 13 shows the completing behavior, *task* completes by consuming one token from it and by generating a token on its outgoing edge $e2$, and producing a message $m1$ on the network according to the chosen communication model.

Figure 13: Completing behavior of a Send Task activity. Before (left) and after (right) application of the Ct rule.

A receive task (RT) has a complementary behavior to the send task (ST). It is enabled to complete only if it is active (*i.e.*, it has a token) and it has a message on one of its incoming message flows. RT completes by consuming the message offer, updating the network state, and producing a token on all its outgoing edges.

Formally.

$$\forall n \in N^{RT}, St(n) \stackrel{def}{\equiv} \exists e \in in^{SF}(n), (me(e) \geq 1) \wedge (me'(e) = me(e) - 1) \\ \wedge\ (mn(n) = 0) \wedge (mn'(n) = mn(n) + 1) \\ \wedge \triangle(\{n, e\}) \wedge \Xi$$

$$\forall n \in N^{RT}, Ct(n) \stackrel{def}{\equiv} (mn(n) = 1) \wedge (mn'(n) = mn(n) - 1) \\ \wedge\ \forall e \in out^{SF}(n), (me'(e) = me(e) + 1) \\ \wedge\ \exists ee \in in^{MF}(n), (me(ee) \geq 1) \wedge (me'(ee) = me(ee) - 1) \\ \wedge\ receive(procOf(source(ee)), procOf(n), msg(ee)) \\ \wedge \triangle(\{n, ee\} \cup out^{SF}(n))$$

Example. The starting of the receive task activity is similar to the one presented for the abstract task in Figure 8. Fig. 14 shows that the receive task activity ($task$) can complete if it has a token on itself and a message $m1$ on its incoming message flow. It completes by consuming its token, receiving the message from the network, and producing a token on its outgoing edge ($e2$).

Figure 14: Completing behavior of a Receive Task activity. Before (left) and after (right) application of the Ct rule.

A throw message intermediate event ($TMIE$) defines only the starting behavior. It is enabled to start if it has a token on one of its incoming edges. It starts by consuming the token from this incoming edge, sending a message on the network according to the chosen communication model, and producing a token on all its outgoing edges.

Formally.

$$\forall n \in N^{TMIE}, St(n) \stackrel{def}{\equiv} \exists ein \in in^{SF}(n), (me(ein) \geq 1) \wedge (me'(ein) = me(ein) - 1)$$
$$\wedge\ \forall e \in out^{SF}(n), (me'(e) = me(e) + 1)$$
$$\wedge\ \exists eout \in out^{MF}(n), (me'(eout) = me(eout) + 1)$$
$$\wedge\ send(procOf(n), procOf(target(eout)), msg(eout))$$
$$\wedge\ \triangle\ (\{ein, eout\} \cup out^{SF}(n))$$

Example. Figure 15 shows the starting behavior of a throw message intermediate event (*Send Notif.*). It starts by consuming the token from its incoming edge ($e1$), producing a token on its outgoing edge ($e2$), and sending a message $m1$ on the network.

Figure 15: Starting behavior of a Throw Message Intermediate Event. Before (left) and after (right) application of the St rule.

A catching message intermediate event ($CMIE$) is an instantaneous event with only a starting transition. It is enabled if it has a message offer on one of its incoming message flow edges and a token on one of its incoming sequential flow edges. It starts by consuming the token from this incoming edge, receiving the message from the incoming message flow according to chosen the communication model, and producing a token on all its outgoing edges.

Formally.

$$\forall n \in N^{CMIE}, St(n) \stackrel{def}{\equiv} \exists e_1 \in in^{SF}(n), (me(e_1) \geq 1) \wedge (me'(e_1) = me(e_1) - 1)$$
$$\wedge\ \forall e_2 \in out^{SF}(n), (me'(e_2) = me(e_2) + 1)$$
$$\wedge\ \exists ein \in in^{MF}(n), (me(ein) \geq 1) \wedge (me'(ein) = me(ein) - 1)$$
$$\wedge\ receive(procOf(source(ein)), procOf(n), msg(ein))$$
$$\wedge\ \triangle\ (\{e_1, ein\} \cup out^{SF}(n))$$

Example. Figure 16 shows the starting behavior of a catching message intermediate event (*Receive Notif.*). It starts by consuming the token from $e1$, receiving the message $m1$ from the medium, and producing a token on its outgoing edge, $e2$.

Figure 16: Starting behavior of a Catching Message Intermediate Event. Before (left) and after (right) application of the St rule.

Boundary Events. A message boundary event (*MBE*) defines only the starting behavior. An *MBE* is ready to start if it has a message offer on one of its incoming message flows, and if the activity on which it is attached has a token. An *MBE* may have either an interrupting behavior or a non-interrupting one. In the latter case, the *MBE* starts by receiving a message and generating a token on all its outgoing edges. For an interrupting behavior, the *MBE* starts also by cancelling the activity to which it is attached, which is possible only if this activity is not a sub-process in a completing step. This is checked using the *mayComplete* predicate that is formally defined below. Cancelling an activity involves dropping all its tokens. After that, the *MBE* produces a token on each of its outgoing edges.

Formally.

Auxiliary functions. To formalize the semantics of a message boundary event, we define an auxiliary function.

- $mayComplete(n) : N^{SP} \rightarrow Bool$, returns true if the subprocess may complete, *i.e.*, if there are no tokens on its elements except for its end event nodes where there is at least one that holds some tokens.

$$\begin{aligned}\forall n \in N^{SP}, mayComplete(n) \stackrel{def}{\equiv}\ & (mn(n) \geq 1) \\ & \wedge\ \forall e \in (R(n) \cap E), (me(e) = 0) \\ & \wedge\ \exists nn \in R(n) \cap N^{EE}, (mn(nn) \geq 1) \\ & \wedge\ \forall x \in R(n) \cap (N \setminus N^{EE}), (mn(x) = 0)\end{aligned}$$

$$\begin{aligned}St_{interrupting}(n, act, ein) \stackrel{def}{\equiv}\ & (act \notin N^{SP} \wedge (mn'(act) = 0) \wedge \triangle(\{act, ein\} \cup out^{SF}(n)) \ ) \\ & \vee \left(\begin{array}{l} act \in N^{SP} \wedge \neg mayComplete(act) \wedge (mn'(act) = 0) \\ \wedge\ \forall nn \in R(act) \cap N, (mn'(nn) = 0) \\ \wedge\ \forall ee \in R(act) \cap E, (me'(ee) = 0) \\ \wedge\ \triangle\ (\{act, ein\} \cup R(act) \cup out^{SF}(n)) \end{array} \right)\end{aligned}$$

$$\begin{aligned}\forall n \in N^{MBE}, St(n) \stackrel{def}{\equiv}\ & \exists act \in N^{A}, (act = attachedTo(n)) \wedge (mn(act) = 1) \\ & \wedge\ \exists ein \in in^{MF}(n), (me(ein) \geq 1) \\ & \quad \wedge\ receive(procOf(source(ein)), procOf(n), msg(ein)) \\ & \quad \wedge\ (me'(ein) = me(ein) - 1) \\ & \quad \wedge\ \forall eo \in out^{SF}(n), (me'(eo) = me(eo) + 1) \\ & \quad \wedge \left(\begin{array}{l} (isInterrupt(n) \wedge St_{interrupting}(n, act, ein)) \\ \vee\ (\neg isInterrupt(n) \wedge \triangle(\{ein\} \cup out^{SF}(n))) \end{array} \right)\end{aligned}$$

Example. Figure 17 presents a part of a process with a none start event *nse*, an abstract task $task1$ with an outgoing sequence flow edge $e2$, an interrupting boundary event (*interrupt*) with an outgoing sequence flow edge $e4$, and two abstract tasks $task2$, and $task3$. The *interrupt* boundary node starts by consuming a token from the activity it is attached to, receiving a message $m1$ from the network, and producing a token on its outgoing edge $e4$.

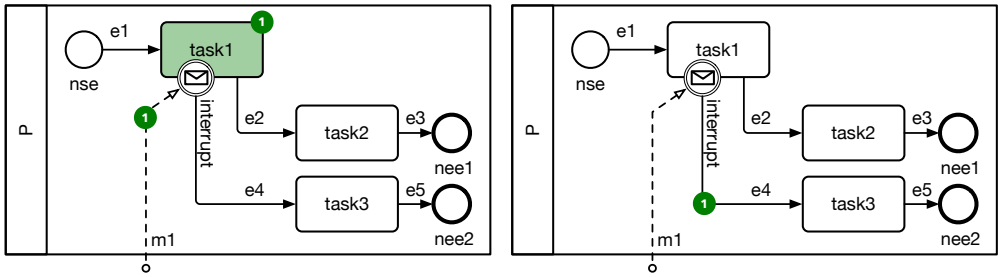

Figure 17: Starting behavior of an interrupting Message Boundary Event (task case). Before (left) and after (right) application of the *St* rule.

Figure 18 shows an interrupting boundary event *interrupt* attached to a sub-process *SP*. The *interrupt* event interrupts the execution of *SP* when it receives a message $m1$. It cancels the execution of the sub-process by removing all its token (the token on it and the token on node $nse1$), and generates a token on its outgoing sequence flow edge, $e4$.

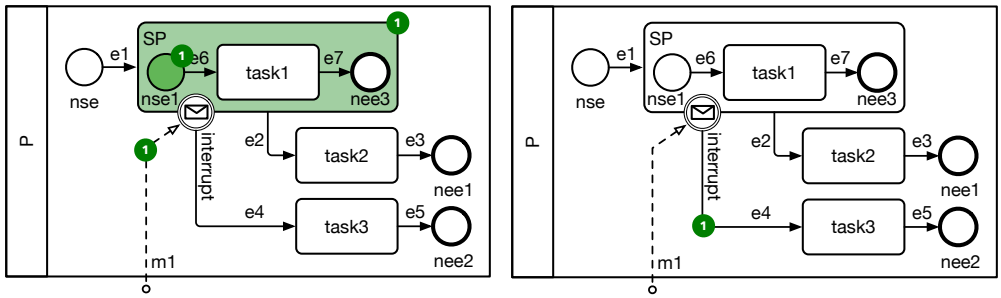

Figure 18: Starting behavior of an interrupting Message Boundary Event (sub-process case). Before (left) and after (right) application of the *St* rule.

Gateways. Gateways are atomic and define only the completing behavior.

A parallel gateway (AND) is ready to complete if it has at least one token on all its incoming edges. It completes by removing one token on each of these edges, and producing one on all its outgoing edges.

Formally.

$$\begin{aligned}\forall n \in N^{AND}, Ct(n) \stackrel{def}{\equiv}\ & \forall ei \in in^{SF}(n), (me(ei) \geq 1) \wedge (me'(ei) = me(ei) - 1) \\ & \wedge\ \forall eo \in out^{SF}(n), (me'(eo) = me(eo) + 1) \\ & \wedge \triangle\,(in^{SF}(n) \cup out^{SF}(n)) \wedge \Xi\end{aligned}$$

Example. Figure 19 shows that the parallel gateway $AND1$ completes by consuming a token from its incoming edge ($e1$) and producing a token on all its outgoing edges ($e2$ and $e3$).

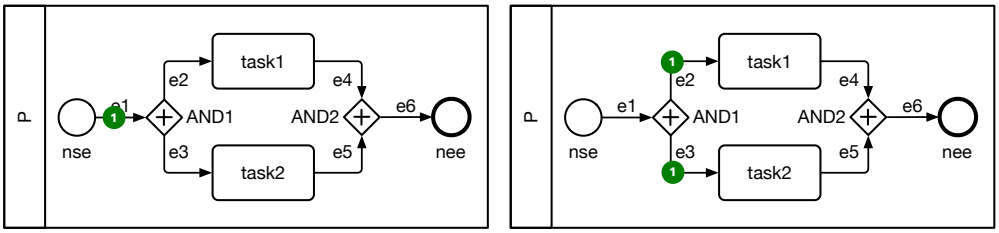

Figure 19: Completing behavior of a splitting Parallel Gateway. Before (left) and after (right) application of the Ct rule.

In Figure 20, the parallel gateway $AND2$ completes only if all its incoming sequence flows edges ($e4$ and $e5$) are synchronized (*i.e.*, own at least a token).

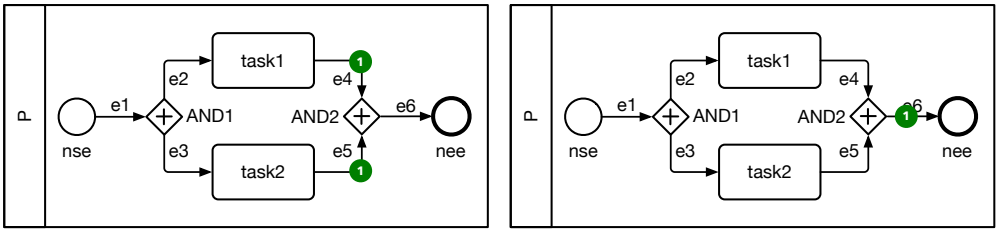

Figure 20: Completing behavior of a merging Parallel Gateway. Before (left) and after (right) application of the Ct rule.

An exclusive gateway (XOR) is ready to complete if it has at least one token on one of its incoming edges. It completes by removing this token, and producing one on one of its outgoing edges, depending on conditions. Since we abstract away from data, the concerned edge is non-deterministically chosen.
Formally.

$$\begin{aligned}\forall n \in N^{XOR}, Ct(n) \stackrel{def}{\equiv}\ & \exists ei \in in^{SF}(n), (me(ei) \geq 1) \wedge (me'(ei) = me(ei) - 1) \\ & \wedge\ \exists eo \in out^{SF}(n), (me'(eo) = me(eo) + 1) \\ & \wedge\ \triangle(\{ei, eo\}) \wedge \Xi\end{aligned}$$

Example. Figure 21 shows that the exclusive gateway ($XOR1$) completes by consuming a token from its incoming edge ($e1$), and producing a token on one of its outgoing edges ($e2$ in the example).

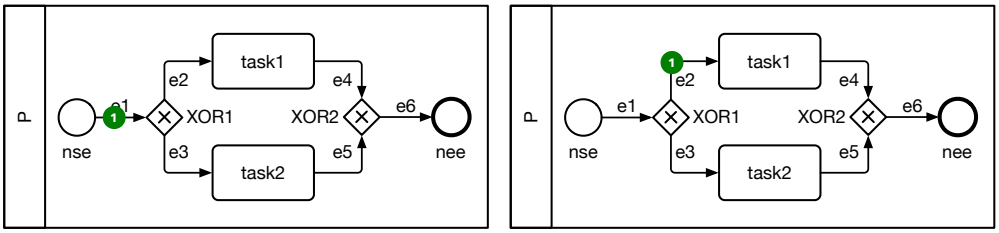

Figure 21: Completing behavior of an Exclusive Gateway. Before (left) and after (right) application of the Ct rule.

As described in the standard [1], an event-based gateway (EB) is always followed by communication elements, either receive tasks (RT) or intermediate catching message events ($CMIE$)[2]. The firing of an event-based gateway relies on the enabling of one of these elements. Hence, an event-based gateway completes by consuming a token from one of its incoming edges, and producing a token on its outgoing edge on which the event is enabled.

Formally.

$$\begin{aligned}\forall n \in N^{EB}, Ct(n) \stackrel{def}{\equiv}\ & \exists ei \in in^{SF}(n), (me(ei) \geq 1) \wedge (me'(ei) = me(ei) - 1) \\ & \wedge \exists eo \in out^{SF}(n), target(eo) \in N^{\{RT,CMIE\}} \\ & \quad \wedge \exists em \in in^{MF}(target(eo)), (me(em) \geq 1) \\ & \quad \wedge (me'(eo) = me(eo) + 1) \\ & \quad \wedge \triangle(\{ei, eo\}) \wedge \Xi\end{aligned}$$

Example. Figure 22 shows a process that contains an event-based gateway (EBG), and two receive tasks ($Rec.1$ and $Rec.2$). The left-hand side of the figure shows that the event-based gateway is enabled because it has a token on its incoming edge ($e1$), and an incoming message for at least one of the two receive tasks. Actually, this is true for both receive tasks here. Hence, EBG completes by consuming the token from $e1$, and generating a token on one of its two outgoing edges (arbitrarily chosen, here $e2$).

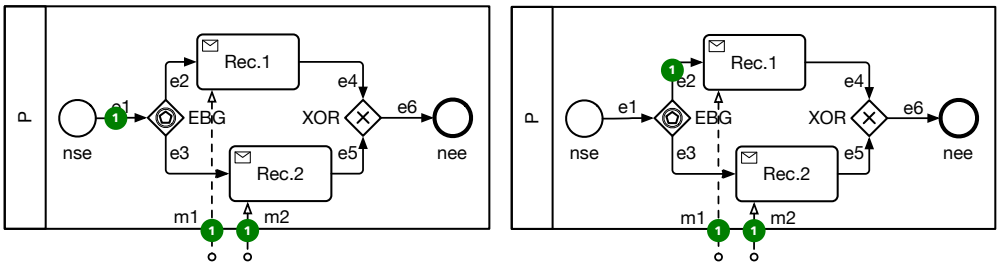

Figure 22: Completing behavior of an Event-Based Gateway. Before (left) and after (right) application of the Ct rule.

[2]We recall that the semantics for time-related constructs is the focus of Section 3, hence we extend the definition of event-based gateways to support time-related events there.

An inclusive gateway (OR) behaves differently from the other gateways. The activation of an OR gateway g is more complex [1, Chap. 13]. It can be activated only if:

- (1) it has at least one token on one of its incoming edges, and
- (2) for each marked node or edge x such that there is a path – that does not pass through g – from x to an unmarked incoming edge of g, there must be also a path – that does not pass through g – from x to a marked incoming edge of g.

The OR gateway completes by adding a token either to the outgoing edges whose conditions are true, otherwise to its default sequence flow edge. Since we abstract from data, we chose non-deterministically to add a token either to a combination (1 or more) of the outgoing non-default edges, or to the default edge.

Formally.

Auxiliary functions. To formalize the semantics of an OR gateway, we define some auxiliary functions.

- $Pre_N : N \times E \to 2^N$ gives the predecessor nodes of an edge such that n^{pre} is in $Pre_N(n, e)$ if there is a path from n^{pre} to e that never visits n. Accordingly, $Pre_E : N \times E \to 2^E$ gives predecessor edges. These two sets can be structurally computed from the BPMN graph structure, hence can be taken as constants (for a given BPMN model).
- $InMinus : N \to E$ gives the unmarked incoming edges of a node:

$$InMinus(n) = \{e \in in^{SF}(n) \mid me(e) = 0\}$$

- $InPlus : N \to E$ gives the marked incoming edges of a node:

$$InPlus(n) = \{e \in in^{SF}(n) \mid me(e) \geq 1\}$$

- $ignore_E : N \to 2^E$ gives the predecessor edges of the marked incoming edges of a given node:

$$ignore_E(n) \stackrel{def}{=} \bigcup_{e \in InPlus(n)} Pre_E(n, e)$$

- $ignore_N : N \to 2^N$ gives the predecessor nodes of the marked incoming edges of a given node:

$$ignore_N(n) = \bigcup_{e \in InPlus(n)} Pre_N(n, e)$$

$$\forall n \in N^{OR}, Ct(n) \stackrel{def}{\equiv} (InPlus(n) \neq \emptyset)$$
$$\wedge\ \forall e \in InPlus(n), (me'(e) = me(e) - 1)$$
$$\wedge\ \forall ez \in InMinus(n), \forall ee \in (Pre_E(n, ez) \setminus ignore_E(n)), (me(ee) = 0)$$
$$\wedge\ (\forall nn \in (Pre_N(n, ez) \setminus ignore_N(n)), (mn(nn) = 0))$$
$$\wedge \left(\begin{array}{ll} & \left(\begin{array}{l} \exists Outs \subset (out^{SF}(n) \cap E^{\{NSF, CSF\}}), (Outs \neq \emptyset) \\ \quad \wedge\ \forall e \in Outs, (me'(e) = me(e) + 1) \\ \quad \wedge\ \triangle\ (InPlus(n) \cup Outs) \wedge \Xi \end{array} \right) \\ \vee & \left(\begin{array}{l} \exists e \in out^{DSF}(n), (me'(e) = me(e) + 1) \\ \quad \wedge\ \triangle\ (InPlus(n) \cup \{e\}) \wedge \Xi \end{array} \right) \end{array} \right)$$

Example. Figure 23 illustrates the case when an OR getaway cannot be activated, despite a marked incoming edge, here $e3$: there is a path from the marked edge $e2$ to an unmarked incoming edge of OR ($e6$ or $e7$) but no path from $e2$ to a marked incoming edge of OR. If the token on $e2$ had been on $e1$, the OR gateway could have been activated.

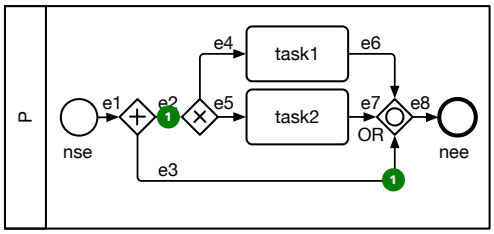

Figure 23: Non activable Inclusive Gateway. It has to wait for the token on $e2$ which is in $Pre_E(OR, e6)$ and $Pre_E(OR, e7)$.

2.2.4. Communication Models

Generic Communication Models. We define seven communication models which differ in the order the messages can be sent or received, and are all the possible point-to-point models when considering local ordering (per process), causal ordering, and global ordering (absolute time) [12]. In the following we define the structure of each model as a type called T_{net} where $T_{net} \in \{bag, pair, inbox\ , outbox, causal, fifoall, RSC\}$.

The models are formally defined in Table 1 and are explained below. To simplify notation, we include sequences of terms ($Seq[T]$) and bags ($Bag[T]$) as a part of the usual definition of ground terms in *first order logic*. We also assume some standard definitions and operations on terms:

- $\langle\rangle$ is the empty sequence
- $head : Seq[T] \to T$: head of a sequence
- $tail : Seq[T] \to Seq[T]$: tail of a sequence
- $append : Seq[T] \times T \to Seq[T]$: append an element at the end of a sequence
- $\oplus, \ominus$: $Bag[T] \times Bag[T] \to Bag[T]$: union and difference of bags

Table 1: Encoding of the network models in first-order logic.

Network model	Definition
Bag	$initnet \stackrel{def}{\equiv} \emptyset$ $send(from, to, m) \stackrel{def}{\equiv} mnet' = mnet \oplus \{\langle from, to, m\rangle\}$ $receive(from, to, m) \stackrel{def}{\equiv} \langle from, to, m\rangle \in mnet \wedge mnet' = mnet \ominus \{\langle from, to, m\rangle\}$
Fifo Pair	$initnet \stackrel{def}{\equiv} [p, q \in N^P \mapsto \langle\rangle]$ $send(from, to, m) \stackrel{def}{\equiv} mnet'(from, to) = append(mnet(from, to), m)$ $receive(from, to, m) \stackrel{def}{\equiv} m = head(mnet(from, to)) \wedge mnet'(from, to) = tail(mnet(from, to))$
Fifo Inbox	$initnet \stackrel{def}{\equiv} [p \in N^P \mapsto \langle\rangle]$ $send(from, to, m) \stackrel{def}{\equiv} mnet'(to) = append(mnet(to), \langle from, m\rangle)$ $receive(from, to, m) \stackrel{def}{\equiv} \langle from, m\rangle = head(mnet(to)) \wedge mnet'(to) = tail(mnet(to))$
Fifo Outbox	$initnet \stackrel{def}{\equiv} [p \in N^P \mapsto \langle\rangle]$ $send(from, to, m) \stackrel{def}{\equiv} mnet'(from) = append(mnet(from), \langle to, m\rangle)$ $receive(from, to, m) \stackrel{def}{\equiv} \langle to, m\rangle = head(mnet(from)) \wedge mnet'(from) = tail(mnet(from))$
Fifo All	$initnet \stackrel{def}{\equiv} \langle\rangle$ $send(from, to, m) \stackrel{def}{\equiv} mnet' = append(mnet, \langle from, to, m\rangle)$ $receive(from, to, m) \stackrel{def}{\equiv} \langle from, to, m\rangle = head(mnet) \wedge mnet' = tail(mnet)$
Causal	$initnet \stackrel{def}{\equiv} \emptyset \times [p \in N^P \mapsto [q \in N^P \mapsto 0]]$ $send(from, to, m) \stackrel{def}{\equiv}$ $mnet'[1] = mnet[1] \cup \{\langle from, to, m, vc[from]\rangle\}$ $\wedge\ mnet'[2] = vc$ with $vc \stackrel{def}{\equiv} [p \in N^P \to [q \in N^P \to$ if $p = from \wedge q = from$ then $mnet[2][p][q] + 1$ else $mnet[2][p][q]\ ]]$ $receive(from, to, m) \stackrel{def}{\equiv}$ $\exists msg \in mnet[1], msg[1] = from \wedge msg[2] = to \wedge msg[3] = m$ $\wedge \neg(\exists msg_2, msg_2 \neq msg \wedge msg_2[2] = to \wedge\ \forall p \in N^P, msg_2[4][p] \leq msg[4][p])$ $\wedge\ mnet'[1] = mnet[1] \setminus \{msg\}$ $\wedge\ mnet'[2] = [p \in N^P \mapsto$ if $p = to$ then $Sup(mnet[2][p], m[4])$ else $mnet[2][p]]$
RSC	$initnet \stackrel{def}{\equiv} \emptyset$ $send(from, to, m) \stackrel{def}{\equiv} mnet = \emptyset \wedge mnet' = \{\langle from, to, m\rangle\}$ $receive(from, to, m) \stackrel{def}{\equiv} \langle from, to, m\rangle \in mnet \wedge mnet' = \emptyset$

- **Bag** is a multiset of messages. Formally:

$$bag \stackrel{def}{\equiv} Bag[N^P \times N^P \times \mathbb{M}]$$

No order on message reception is imposed. Messages can overtake each other or be arbitrarily delayed. It is usually modeled by a bag, or a set if messages are unique.

- **Fifo pair** is a queue of messages attached to each couple of processes. Formally:

$$pair \stackrel{def}{\equiv} N^P \times N^P \to Seq[\mathbb{M}]$$

Messages between a couple of processes are received in their send order. Messages from or to different processes are independently received. More precisely, if a process P_1 sends a message m_1 to process P_2, and later a message m_2 to this same process, then m_2 cannot be received before m_1.

- **Fifo inbox** is an input queue attached to each process, where senders put messages. Formally:

$$inbox \stackrel{def}{\equiv} N^P \to Seq[N^P \times \mathbb{M}]$$

Each process has its own unique input queue, and senders add messages to this queue, without blocking. Messages are consumed from this queue in their insertion order. This means that if a process P_1 sends a message m_1 to P_0, and later (but independently) a process P_2 sends a message m_2 to P_0, then m_2 cannot be received before m_1. This model is stricter than Fifo pair as it requires a global order on the emission events.

- **Fifo outbox** is an output queue attached to each process where messages are retrieved. Formally:

$$outbox \stackrel{def}{\equiv} N^P \to Seq[N^P \times \mathbb{M}]$$

Messages from a same process are received in their send order. If a process P sends a message m_1 and later a message m_2 (to the same process or to another one), then m_2 cannot be received before m_1, even if the receptions occurs on distinct processes.

- **Fifo All** is a unique shared queue. Formally:

$$fifoall \stackrel{def}{\equiv} Seq[N^P \times N^P \times \mathbb{M}]$$

Messages are globally ordered, independently of their sender or receiver process, and are received in the global emission order.

- **Causal.** Messages are received according to the causality of their emission [13]. Formally:

$$VC \stackrel{def}{\equiv} [N^P \to \mathbb{N}] \quad \text{-- } a \textit{ vector clock}$$
$$causal \in Set[N^P \times N^P \times \mathbb{M} \times VC] \times [N^P \to VC]$$

If a message m_1 is causally sent before message m_2 (there exists a causal path from the emission of m_1 to the emission of m_2), then a process cannot receive m_2 before

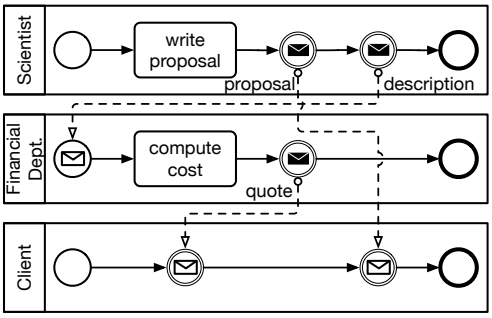

Figure 24: A Non-Causal Execution. The emission of the proposal causally precedes the emission of the description which causally precedes the emission of the quote. Then the quote cannot be consumed before the proposal on the client in the causal communication model.

m_1. Figure 24 presents a model that deadlocks with causal communication. In this example, a scientist writes a proposal, sends the proposal to the client, and sends its description to its financial department. Based on the description, the financial department computes a quote and sends it to the client. Without causal communication, the quote can be received by the client before the proposal; with causal communication, the quote cannot be delivered because it causally depends on the proposal that must be received first. A usual implementation of this model uses causal histories [14, 15] or vector/matrix logical clocks [16] as presented here[3]. The state of the network *mnet* holds a couple: the set of messages in transit (with their associated vector clock), and the vector clocks of each process.

- **RSC (Realisable with Synchronous Communication)** is a shared 1-sized buffer for all processes. Formally:

$$net_{RSC} \stackrel{def}{\equiv} (N^P \times N^P \times \mathbb{M})$$

Send and receive events strictly alternate. If the couple (send event, reception event) is considered as atomic, this corresponds to synchronous communication [17].

Ad-hoc Communication Models. The previous communication models are monolithic and are applied on the whole collaboration. In Section 5.4, we show how a modular communication model can be specified.

2.2.5. Transition Relation and Executions

With the previously defined predicates, we are able now to express the complete transition relation (successor relation between states).

Definition 2.5 (Transition Relation)**.** Let s and s' be two states. We say that s' is a successor of s, iff the predicate $Next(s, s')$ holds:

$$Next(s, s') \stackrel{def}{\equiv} \bigvee_{n \in N} (St(n) \vee Ct(n))$$

[3] A vector clock is a logical clock that tracks the causal dependencies between send and receive events. They are not to be confused with the clocks used for timing constraints in Section 3.3.

We recall that states, here s and s', correspond to tuples of the form (me, mn, $mnet$) and (me', mn', $mnet'$), whose elements are used in the definitions of St and Ct.

An execution of the whole process is defined through the notion of *trace*.

Definition 2.6 (Trace)**.** A trace is a finite or infinite sequence of states such that $\sigma[0]$ is the initial state, and $\forall i \in 0..Len(\sigma) - 1, Next(\sigma[i], \sigma[i+1])$ (if the trace is finite) or $\forall i \in \mathbb{N}, Next(\sigma[i], \sigma[i+1])$ (if the trace is infinite), where $\sigma[i]$ denotes the i^{th} state of the trace. The set of all the traces of the collaboration is noted $Traces$.

Definition 2.7 (Execution)**.** An execution is a maximal trace, *i.e.*, a trace that goes as far as possible. Formally, an execution σ is either an infinite trace, or a finite trace such that $\nexists s', Next(\sigma[Len(\sigma)], s')$. The set of all the executions of the collaboration is noted $\mathcal{E}xec$.

2.2.6. Fair Executions

A BPMN model can include loops. In that case, an execution defined as a maximal trace can get stuck in a loop, where only this loop progresses, and the rest of the model doesn't progress at all. Moreover, when modeling actual business activities, loops are expected to finish at some point. To prevent these infinite loops, fairness is introduced to restrain the set of executions. We use two kinds of fairness: weak fairness and strong fairness. Informally, weak fairness ensures that a transition cannot be permanently enabled and never done. Strong fairness ensures that a transition cannot be infinitely often enabled and never done. Both are formally defined in Section 5.3.

Thus, fairness is a conjunction of two parts. The first part is the weak fairness on each start (St) and complete (Ct) transitions of every node: $\forall n \in Node : weakfair(step(n))$. This property ensures that any permanently enabled transition eventually occurs. This means that no process may progress forever while others are never allowed to do so if they can. This also means that if a process contains several loops that are simultaneously live, all loops will progress (not necessarily at the same speed, but no loop can be permanently halted while another run forever).

The second part is the strong fairness on each output edges of *XOR*, *OR*, and *EB* gateways. Strong fairness ensures that no choice is infinitely often ignored: if a *XOR*, *OR*, or *EB* gateway is included in a loop, the fairness forbids the infinite executions that never use some output edges. Either the loop finishes somehow, or all the choices are infinitely often taken. Consider Figure 25, left; as strong fairness is imposed on the two output edges of gateway *choice*, an execution cannot always ignore the edge ($e5$) leading to the ending node, and this model is sound. Consider Figure 25, right; as strong fairness is imposed on the output edges of gateway *choice*, both $task1$ and $task2$ are infinitely often chosen.

3. Temporal Support for BPMN

The goal of our FOL semantics is to give a meaning to the BPMN elements. In a sense, it defines which executions are possible. With regard to time, our semantics is asynchronous: enabled elements are fired at some point, but not necessarily at the instant they have become enabled; message transfer is asynchronous, which means that a message is available for reception at some point after its sending, but there is no time constraint.

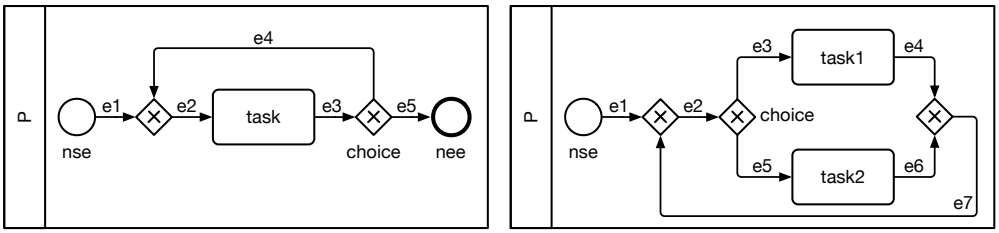

Figure 25: Use of strong fairness to avoid infinite loops (left) and starvation (right).

With these remarks in mind, we present two choices for handling time: in the first one, the semantics does not explicitly handle time constraints, but it reverts to non-determinism: if a time constraint exists, it *may* fire; in the second one, the semantics is defined with explicit time to verify duration properties.

3.1. BPMN Elements with Time

Events where time occurs are the following:

1. Timer Start Event for Top-level Processes. A process is started when the time constraint allows it.
2. Timer Intermediate Catch Event. The norm states that it acts as a delay.
3. Timer Boundary Event, which can be interrupting or non-interrupting. It can act as a timeout on a long running task (interrupting event) or create a secondary path when the time constraint specifies it (non-interrupting event).
4. An Event-Based Gateway can be followed by a Timer Intermediate Catch Event. Especially, it can be used as a timeout while waiting for a message amongst several links.

An event with time has an attribute specifying the time constraint. It is one of timeDate (a date and/or time, e.g., 2019-11-27 16:44), timeDuration (a time duration, e.g., PT05:15 for a duration of five hours and fifteen minutes), or timeCycle (a time interval, e.g., a start and end time, or a start time and a duration).

3.2. Non-deterministic Abstraction of Time

So far, our FOL semantics is asynchronous (an event or a task is fired at any point of time after it is enabled) and non-deterministic (e.g., for an *XOR* gateway, any of the output branches can be taken, and the verification ensures that all the cases are covered): the semantics arbitrarily fires an enabled transition. Consequently, time progress can be modeled with non-determinism.

1. Timer Start Event (*TSE*): as the semantics is asynchronous, it is exactly as a None Start Event (*NSE*), where the event will fire at some point.
2. Timer Intermediate Catch Event (*TICE*): it is indistinguishable from a gateway with one input and one output (it fires at some point).

$$\begin{aligned}\forall n \in N^{TICE}, Ct(n) \stackrel{def}{\equiv}\ & \exists ei \in in^{SF}(n), (me(ei) \geq 1) \wedge (me'(ei) = me(ei) - 1) \\ & \wedge \exists eo \in out^{SF}(n), (me'(eo) = me(eo) + 1) \\ & \wedge \triangle(\{ei, eo\}) \wedge \Xi\end{aligned}$$

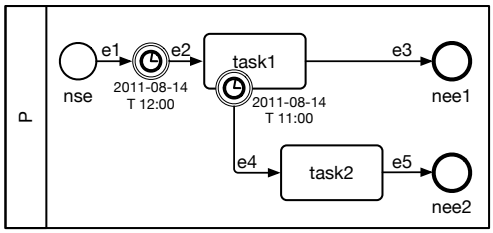

Figure 26: Timer Boundary Event with an impossible timeout. As the timeout on $task1$ is later than the date imposed by the timer intermediate catch event, this timeout should never fire and $task2$ should never be activated.

3. Timer Boundary Event (TBE): a TBE is non-deterministically activated, without fairness (it may never fire). Its semantics is close to a Message Boundary Event (MBE), without the constraint on the presence of a message.

$$St_{interrupting}(n, act) \stackrel{def}{\equiv} (act \notin N^{SP} \wedge (mn'(act) = 0) \wedge \triangle(\{act\} \cup out^{SF}(n))) \vee \left(\begin{array}{l} act \in N^{SP} \wedge \neg mayComplete(act) \wedge (mn'(act) = 0) \\ \wedge\ \forall nn \in R(act) \cap N, (mn'(nn) = 0) \\ \wedge\ \forall ee \in R(act) \cap E, (me'(ee) = 0) \\ \wedge \triangle(\{act\} \cup R(act) \cup out^{SF}(n)) \end{array} \right)$$

$$\forall n \in N^{TBE}, St(n) \stackrel{def}{\equiv} \exists act \in N^{A}, (act = attachedTo(n)) \wedge (mn(act) = 1) \wedge \forall eo \in out^{SF}(n), (me'(eo) = me(eo) + 1) \wedge \left(\begin{array}{l} (isInterrupt(n) \wedge St_{interrupting}(n, act)) \\ \vee\ (\neg isInterrupt(n) \wedge \triangle(out^{SF}(n))) \end{array} \right) \wedge \Xi$$

4. Event-Based Gateway followed by a Timer Intermediate Catch Event: as the event-based gateway can be fired at any time after it is enabled, a branch with a $TICE$ is indistinguishable from a Default branch followed by a $TICE$ modelled as above in item 2.

A limitation to non-deterministic time is when a TBE can actually never be fired in BPMN semantics (e.g., a date in the past): the non-deterministic semantics allows it to fire. This gives us an over-approximation: the non-deterministic semantics contains the same executions as with BPMN semantics, plus additional ones. Thus, if the verification states that a property is verified with the non-deterministic semantics, it is necessarily verified with BPMN semantics. The reverse is not true. For instance, in the example of Fig. 26, BPMN semantics states that task $task2$ should never be activated as its time constraint is always in the past. The non-deterministic semantics defines two executions: one where the timeout is not fired, and one where the timeout is fired and $task2$ is activated.

3.3. Explicit Time

In order to integrate the notion of explicit time, we extend the BPMN graph to deal with some time constraints: here, we treat only the *timeDuration* type for the different time events (i.e., *timeDate* and *timeCycle* types are not handled by our extension).

To simplify our definition, we introduce the set *Timer* that groups the time start (*TSE*), the timer intermediate catch (*TICE*) and the timer boundary (*TBE*) events: *Timer*= { *TSE*, *TICE*, *TBE*}. Hence, to express the *timeDuration* type for each timer node, we add to the graph definition (Def. 2.1) a *timing* function: $\widehat{G}$= (N, E, $\mathbb{M}$, cat_N, cat_E, $source, target$, R, msg, $attachedTo$, $isInterrupt$, $timing$). $timing$ maps a time duration t to each timer node.

$$timing : N^{Timer} \rightarrow \mathbb{N}$$

.

We extend the state definition with locals clocks (l_c). Hence, the definitions of state (Def. 2.2) and initial state (Def. 2.3) change as follows.

Definition 3.1 (State)**.** $s = (mn, me, mnet, l_c)$ such that:

- $l_c : N^{Timer} \rightarrow \mathbb{N}$, is the local discrete clock representing the time spent on a given timer node.

Definition 3.2 (Initial state)**.** The initial state of a BPMN graph, denoted by $s_o = (mn_0, me_0, mnet_0, l_{c_0})$.

- Local clocks are initialized to zero: $\forall n \in N^{Timer}, l_{c_0}(n) = 0$.
- mn_0, me_0, and $mnet_0$ are initialized as above (Def. 2.3).

Based on these changes, we define the execution semantics of the timer event nodes as follows.

- A start timer event (*TSE*) is defined by a completing predicate. *TSE* is enabled to complete only if it has a token and the time of its local clock has elapsed. It completes by consuming its token, producing a token on each of its outgoing edges and on the process, and resetting its local clock.

$$\begin{aligned}\forall n \in N^{TSE}, Ct(n) \stackrel{def}{\equiv}\ & (l_c(n) \geq Timing(n)) \wedge (l'_c(n) = 0) \\ & \wedge\ (mn(n) = 1) \wedge (mn'(n) = mn(n) - 1) \\ & \wedge\ \forall eo \in out^{SF}(n), (me'(eo) = me(eo) + 1) \\ & \wedge\ \exists p \in N^P, n \in R(p), (mn(p) = 0) \wedge (mn'(p) = mn(p) + 1) \\ & \quad \wedge \triangle\ (\{n, p\} \cup out^{SF}(n)) \wedge \Xi\end{aligned}$$

Example. Consider the process of Figure 27. Here, the start event node (tse) is associated with a timing duration of 5 units. The figure presents the completing behavior of tse. The starting node has a token and its local clock meets the deadline. So the completing transition can fire. The right-hand side of the figure shows the result of this operation: the local clock is reset, the process is started, and a token is added to the outgoing edge ($e1$).

- A timer intermediate catch event (*TICE*) is defined by a starting predicate. A *TICE* is enabled to start if one of its incoming edges has a token, and the time of its local clock has met its deadline. It starts by consuming a token from one of

Figure 27: Completing behavior of a Timer Start Event.

its marked incoming edges, resetting its local clock, and generating a token on its outgoing edges.

Formally.

$$\forall n \in N^{TICE}, St(n) \stackrel{def}{\equiv} (l_c(n) \geq Timing(n)) \wedge (l'_c(n) = 0) \\ \wedge\ \exists ei \in in^{SF}(n), (me(ei) \geq 1) \wedge (me'(ei) = me(ei) - 1) \\ \wedge\ \forall eo \in out^{SF}(n), (me'(eo) = me(eo) + 1) \\ \wedge\ \triangle\ (\{ei\} \cup out^{SF}(n)) \wedge \Xi$$

685 **Example.** Let us consider a claims process. To support his claim, a client should register and provide some documents. Then, the claim is analysed to take the necessary actions and inform the client. The action to be taken is executed after a certain time. This delay can be modelled by an intermediate catching time event. Figure 28 shows the starting behavior of a timer intermediate catching
690 event (*Waiting*). The upper part of the figure shows that *Waiting* is enabled to complete: it has a token on its incoming edge ($e4$), and its clock meets the deadline. The lower part of the figure shows that *Waiting* completes by moving the token on its outgoing sequence flow edge ($e5$), and resetting its local clock.

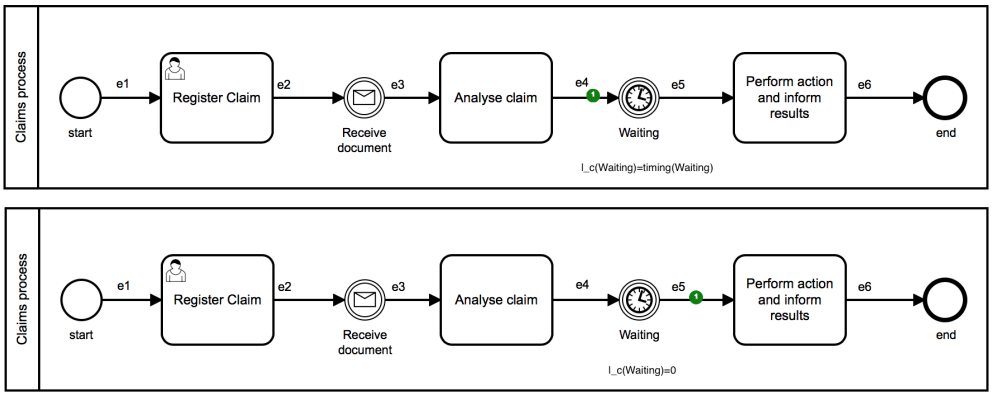

Figure 28: Starting behavior of a Timer Intermediate Catch Event.

- A timer boundary event (*TBE*) is defined by a starting predicate. *TBE* is ready
695 to start if its local clock meets its deadline and the activity attached to it is active (i.e., owns a token). In this case, we distinguish the interrupting behavior from non-interrupting one. In the latter, a *TBE* starts by generating a token on its outgoing

sequence flow edges and resetting its local clock. In the interrupting behavior, a *TBE* starts by cancelling the activity which it is attached to if this activity is not a sub-process in its completing step (i.e., it has at least one token on one of its elements other than its end events). If it is the case, the activity is cancelled by dropping all its tokens. The *TBE* resets then its local clock and produces a token on each of its outgoing edges.

Formally.

$$
\begin{aligned}
St_{interrupting}(n, act) \overset{def}{\equiv}\ & (act \notin N^{SP} \wedge (mn'(act) = 0) \wedge \triangle(\{act\} \cup out^{SF}(n))) \\
& \vee \left(\begin{array}{l} act \in N^{SP} \wedge \neg mayComplete(act) \wedge (mn'(act) = 0) \\ \wedge\ \forall nn \in R(act) \cap N, (mn'(nn) = 0) \\ \wedge\ \forall ee \in R(act) \cap E, (me'(ee) = 0) \\ \wedge\ \triangle\ (\{act\} \cup R(act) \cup out^{SF}(n)) \end{array} \right) \\
\forall n \in N^{TBE}, St(n) \overset{def}{\equiv}\ & \exists act \in N^{A}, (act = attachedTo(n)) \wedge (mn(act) = 1) \\
& \wedge\ (l_c(n) \geq Timing(n)) \wedge (l'_c(n) = 0) \\
& \wedge\ \forall eo \in out^{SF}(n), (me'(eo) = me(eo) + 1) \\
& \wedge \left(\begin{array}{l} (isInterrupt(n) \wedge St_{interrupting}(n, act)) \\ \vee\ (\neg isInterrupt(n) \wedge \triangle(out^{SF}(n))) \end{array} \right) \wedge \Xi
\end{aligned}
$$

Example. Consider an example of an on-line payment process: when the payment page is opened, the user must enter the credit card number, this task is stopped if a timeout (e.g., 10 minutes) is triggered. Besides, the user receives a notification of the remaining time before the deadline (e.g., at minute 7, to say that it remains 3 minutes). The timeout can be modelled by an interrupting boundary time event (*timeOut*) and the reminder by a non-interrupting boundary event (*timeReminder*).

Figure 29 presents the starting behavior of *timeOut*. The left-hand side of the figure shows that *timeOut* is ready to start (i.e., the task *CB_payment* has a token and the local clock of *timeOut* meets its deadline). The right-hand side of the figure shows that *timeOut* starts by cancelling the *CB_payment* task and generating a token on its outgoing edge ($e4$).

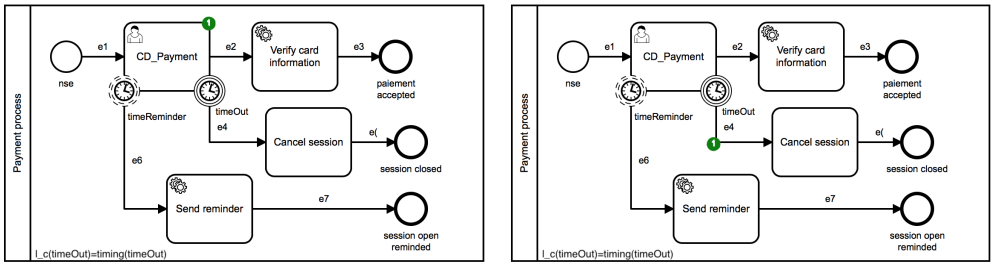

Figure 29: Starting behavior of an interrupting Timer Boundary Event.

Figure 30 presents the starting behavior of *timeReminder*. The left-hand side of the figure shows that *timeReminder* is ready to complete (i.e., *CB_payment* has a token and the local clock of *timeReminder* meets its deadline). The right-hand

side of the figure shows that *timeReminder* completes by generating a token on its outgoing edge ($e6$) without interrupting the behavior of *CB_payment* (it still owns a token).

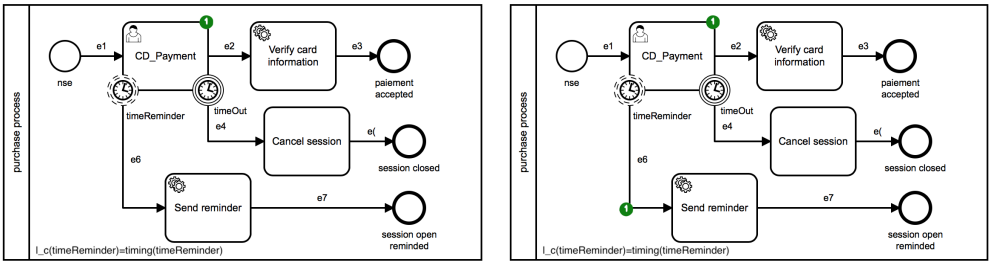

Figure 30: Starting behavior of a non-interrupting Timer Boundary Event.

- Event-Based Gateway. Due to the provided semantics for the timer events, the semantics of an event-based gateway must be changed. In the standard, the execution semantics of an event-based gateway is defined as a branching point where exactly one of its outgoing edges is activated, depending on which event is triggered. Then, the path to that event will be used (a token will be sent down the outgoing sequence flows of the event) [1].

  In a BPMN model, an event-based gateway is followed by a receive task (RT) or an intermediate catching message event ($CMIE$), combined with a timer intermediate catch event ($TICE$): the activation of one of the outgoing edges depends on the enabling of these elements (i.e., the reception of a message, or a specific time event is triggered). So, to handle time, we adapt the semantics of an event-based gateway as follows.

  An Event-based gateway (EB) is defined only by a completing predicate. It is ready to complete if one of its incoming edges has a token and one of its target events is enabled (i.e., the target of an outgoing edge is an RT or a $CMIE$ that has an offer on one of its incoming message edges, or the target of an outgoing edge is a timer intermediate event that meets its deadline). The EB completes by consuming the token from one of its incoming edge and producing a token on the outgoing edge on which the event is enabled.

  Formally.

$$\forall n \in N^{EB}, Ct(n) \stackrel{def}{\equiv} \exists\, ei \in in^{SF}(n), (me(ei) \geq 1) \wedge (me'(ei) = me(ei) - 1)$$
$$\wedge \left(\begin{array}{l} \left(\begin{array}{l} \exists eo \in out^{SF}(n), target(eo) \in N^{\{RT, CMIE\}} \\ \quad \wedge\, \exists em \in in^{MF}(target(eo)), (me(em) \geq 1) \\ \quad \wedge\, (me'(eo) = me(eo) + 1) \wedge \triangle(\{ei, eo\}) \end{array} \right) \\ \vee \left(\begin{array}{l} \exists eo \in out^{SF}(n), (target(eo) \in N^{TICE}) \\ \quad \wedge\, (l_c(target(eo)) \geq timing(target(eo))) \\ \quad \wedge\, (me'(eo) = me(eo) + 1) \wedge \triangle(\{ei, eo\}) \end{array} \right) \end{array} \right)$$
$$\wedge\, \Xi$$

Example. Let us consider an extension of the claim process presented in Figure 28.

If the documentation is not provided by the client within a certain period, the claim will fail. This should be included in the diagram (see Fig. 32). After the registration of the claim, an event-based gateway waits for the first event to occur: a *Receive document* or a *TimeOut*. The figure shows that after a maximum period of time (defined through the deadline of *TimeOut*), if the client does not present the documents, *TimeOut* is activated and then the event-based gateway generates a token on its outgoing edge leading to *TimeOut* (see lower part of Fig. 32). The lower part of Figure 31 shows the case where the documents are provided in time, and the event-based gateway generates a token on the corresponding outgoing edge ($e3$).

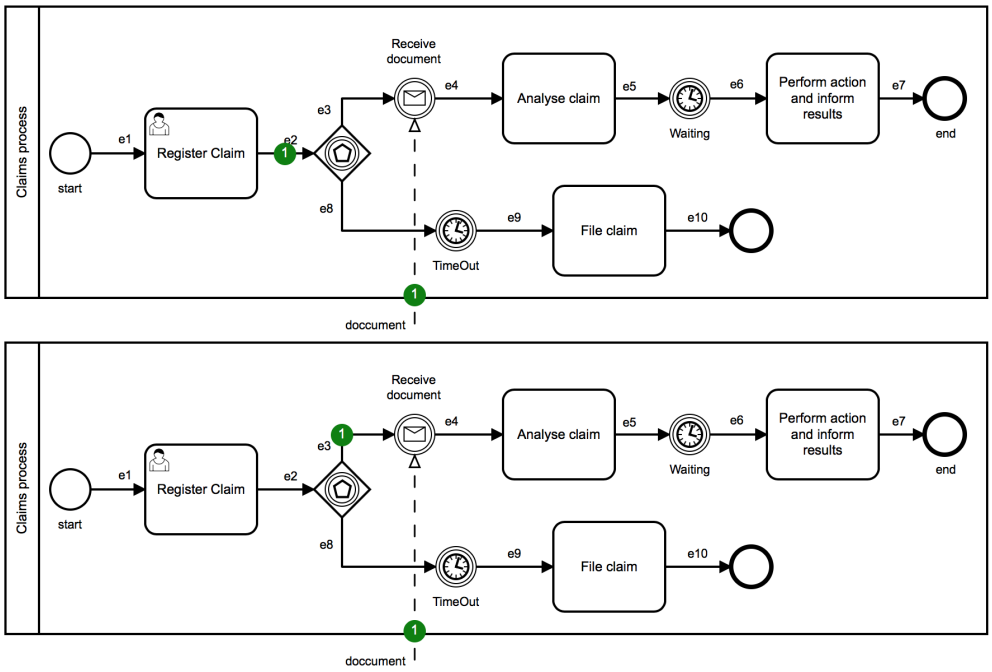

Figure 31: Completing behavior of an Event-Based Gateway: a message is ready to be received.

Transition Relation with Time. Based on the latter, we change the definition of the transition relation mentioned in Section 2.5 by introducing a set of predicates and subsets of nodes:

Let us consider a subset of timer nodes, called S, that groups the nodes that satisfy one of the following conditions: (i) all starting timer nodes that have a token and their local clocks are not active, (ii) all intermediate timer nodes that have an inactive local clock and have a marked incoming edge, or if they follow an event based gateway and the latter has a marked incoming edge, (iii) all boundary timer nodes attached to an active activity and their local clock is not active, or (vi) all active timer nodes (i.e., their local

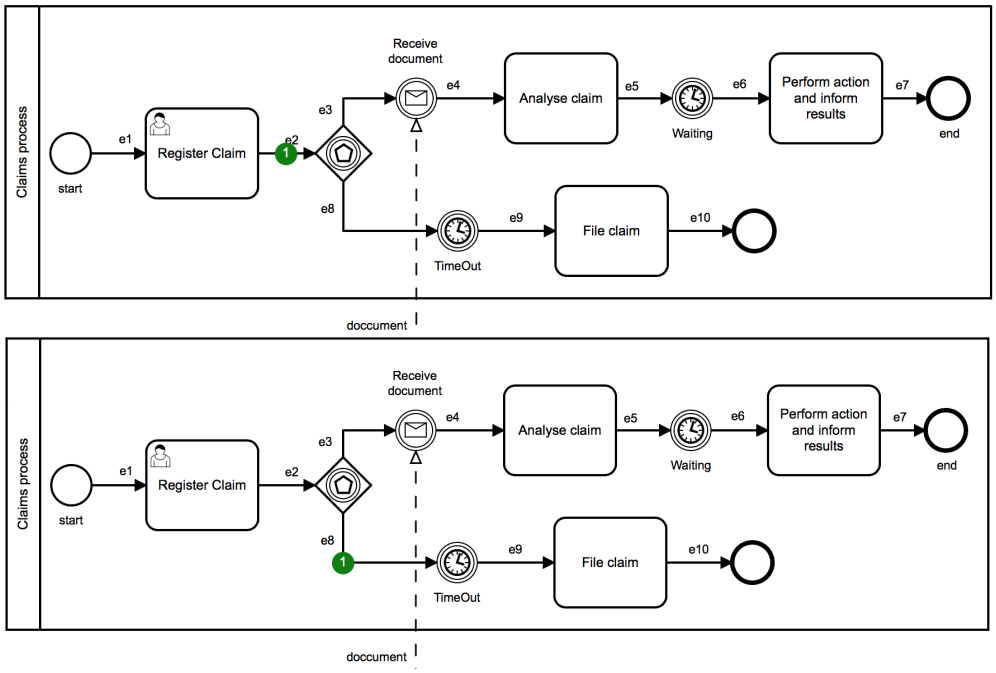

Figure 32: Completing behavior of an Event-Based Gateway: the timer is ready to fire.

clocks are greater than 0 and they didn't reach their timing limits) as follows:

$$
\begin{aligned}
S \stackrel{def}{=} \quad & \{n \in N^{TSE} \mid (l_c(n) = 0) \wedge (mn(n) = 1)\} \\
& \cup \{n \in N^{TICE} \mid \exists e \in in^{SF}(n), (l_c(n) = 0) \wedge (me(e) > 0)\} \\
& \cup \{n \in N^{TICE} \mid \exists e \in in^{SF}(source(in^{SF}(n))), (l_c(n) = 0) \wedge (me(e) > 0)\} \\
& \cup \{n \in N^{TBE} \mid (l_c(n) = 0) \wedge (mn(attachedTo(n)) > 0)\} \\
& \cup \{n \in N^{Timer} \mid timing(n) > l_c(n) > 0\}
\end{aligned}
$$

Let Y be the subset of timer nodes in the BPMN graph that are ready to fire:

$$Y \stackrel{def}{=} \{y \in N^{Timer} \mid l_c(y) \geq Timing(y)\}$$

To facilitate the reading of the transition relation, we define the following predicates:

- *step* defines a step of execution for a given node:

$$step(n) \stackrel{def}{=} St(n) \vee Ct(n)$$

- *stepT* increases the local clock of each enabled timer node or currently executed:

$$stepT \stackrel{def}{=} \forall n \in S, l'_c(n) = l_c(n) + 1$$

- $fztime$ denotes time equality for the local clock of all timer nodes given as parameter:

$$fztime(Z) \stackrel{def}{\equiv} \forall z \in Z, l'_c(z) = l_c(z)$$

The transition relation distinguishes two cases. If at least a timer is ready to fire ($Y \neq \emptyset$), then a timer fires (it does a step) or an event-based gateway that precedes a firable timer does a step. Time does not advance, and other timers with the same expiration time can then fire. If no timer is ready to fire, all timers increase ($stepT$) and non-deterministically, a step can occur ($\exists n, step(n)$) or no step is done ($\triangle(\emptyset) \wedge \Xi$).

Definition 3.3 (Transition Relation with time)**.** Let s and s' be two states. We say that s' is a successor of s, iff the predicate $Next(s, s')$ holds:

$$Next(s,s') \stackrel{def}{\equiv} (Y \neq \emptyset \wedge \left(\begin{array}{l} (\exists n \in Y : step(n) \wedge fztime(N^{Timer} \setminus \{n\})) \\ \vee \left(\begin{array}{l} \exists n \in N^{EB}, \exists eo \in out^{SF}(n), \\ \quad (target(eo) \in Y) \wedge step(n) \wedge fztime(N^{Timer}) \end{array} \right) \end{array} \right) \\ \vee (Y = \emptyset \wedge stepT \wedge fztime(N^{Timer} \setminus S) \wedge ((\exists n \in N : step(n)) \vee (\triangle(\emptyset) \wedge \Xi))$$

3.4. Discussion

The advantage of abstracting time with non-deterministic transitions is its simplicity, and its efficiency in model-checking once translated in TLA$^+$. The first limitation is that it can not recognize impossible-to-meet time constraints, e.g. a time necessarily in the past. As noted above, this is safe with regard to LTL properties, which are the majority of the useful properties in Section 4. The *no dead activities* property is the only one that cannot be checked, as it translates to a CTL existential property. The second limitation is that we cannot derive any temporal information on a collaboration, for instance how long it takes to complete. With explicit time, on the other hand, it is possible to compute the minimal number of ticks a process or a collaboration need to complete. This requires adding history variables [18, p. 270] to record time information such as the start time of a process, without altering the semantics [19]. Our semantics use discrete time. As shown in [20], discrete time allows specifications to be checked with ordinary model checkers, and our semantics do not require continuous time model checkers such as Uppaal.

4. Verification Properties

Verifying a model involves checking the correctness of its properties. In the context of process modelling, properties are classified into two main classes: *structural* and *behavioural*. The *structural properties* relate to the type of elements and how they are connected. Such properties could be checked using a standard process modelling tool which can enforce that the model is correctly designed. The *behavioural properties* relate to the sequences of execution as defined by the process model. We further classify the behavioural properties into general and specific ones. The specific properties are unique to business process models, while the general properties are used in other types of models.

General properties. *deadlocks* and *livelocks* are common examples of general properties. Fig. 33 shows a simple BPMN model with an *XOR* and an *AND* gateway. The *XOR* gateway produces a token either on its outgoing edge $e2$ or $e3$ but not on both. As a consequence, the *AND* gateway will never be enabled (its incoming edges $e4, e5$ will never be synchronized). Hence, this model suffers from a deadlock situation. While in a deadlock the involved activities can never be executed and the process can never be completed, in a livelock situation a set of activities are executed indefinitely.

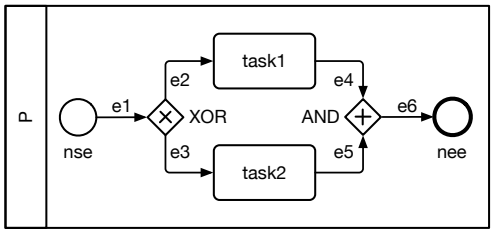

Figure 33: BPMN diagram with deadlock.

Specific properties. *Soundness* is the main property that can be checked after a process model is executed.

Soundness property for business processes was developed by Wil van der Aalst in the context of workflow nets [21]. A workflow net has a unique terminal place and the authors defined the soundness of the WF-net by the satisfaction of the three following requirements: "*(1) Option to complete: from any reachable state, it is possible to reach a state with marks on the terminal place, (2) proper completion: if the terminal place is marked, all other places are empty, (3) no dead transitions: it should be possible to execute an arbitrary activity by following an appropriate route through the WF-net*". In the case of Petri nets in workflow verification, [22] and [23] prove that a workflow net is sound if and only if the corresponding short-circuited Petri net is live and bounded. In addition, they define the safeness of a WF-net by ensuring that each place cannot hold multiple tokens at the same time.

In the book [24], [p.186, ch.5], Dumas et.*al* define informally the soundness of the BPMN process model by the satisfaction of the following three properties: "*(1) Option to complete: any running process instance must eventually complete, (2) Proper completion: at the moment of completion, each token of the process instance must be in a different end event, (3) No dead activities: any activity can be executed in at least one process instance*".

In [5], authors address BPMN collaboration models. They introduce a formal definition of safeness and soundness properties by focussing on the specificities of BPMN. "*A process is safe if during its execution no more than one token occurs along the same sequence edge*". The authors extend the safeness property for processes collaboration which require that "*each of all the processes that involved in the overall collaboration execution is safe*". On the other hand, they define the soundness of the process as follows: "*A BPMN process is sound, if it can complete its execution without leaving active elements and all the model elements can be activated in at least one of the execution traces*". In addition, they extend the latter to define the soundness of the whole collaboration model.

820 Based on those definitions [24, 5], we provide a set of properties that can be formally specified as follows. We recall here that a state in our formalisation of BPMN is $s = (mn, me, mnet)$. For an execution $\sigma = s_1 s_2 ...$ and a node x, we note $\sigma[i].mn(x)$ the value of the marking of x at step i in σ, and $domain(\sigma) = \{1, ..., |\sigma|\}$.

Definition 4.1 (Option to complete)**.** Any running process must eventually complete. A process is complete in a state if markings occur only on end events.

$$Completed(p, s) \stackrel{def}{\equiv} \forall n \in R(p) \cap N, (s.mn(n) = 0) \vee (s.mn(n) = 1 \wedge n \in N^{EE}) \\ \wedge (\forall e \in R(p) \cap E, (s.me(e) = 0))$$

$$OptionToComplete \stackrel{def}{\equiv} \forall p \in N^P, \forall \sigma \in \mathcal{E}xec, \forall i \in domain(\sigma), \sigma[i].mn(p) > 0 \Rightarrow \\ \exists j \in domain(\sigma), j \geq i \wedge Completed(p, \sigma[j])$$

Definition 4.2 (Proper Completion)**.** At the moment of completion, each token of the process instance must be in a different end event.

$$ProperCompletion \stackrel{def}{\equiv} \forall p \in N^P, \forall \sigma \in \mathcal{E}xec, \forall i \in domain(\sigma), \\ Completed(p, \sigma[i]) \Rightarrow \forall n \in R(p) \cap N^{EE}, \sigma[i].mn(n) = 1$$

Definition 4.3 (No dead activities)**.** An activity is dead if no execution activates it.

$$NoDeadActivities \stackrel{def}{\equiv} \forall a \in N^A, \exists \sigma \in \mathcal{E}xec, \exists i \in domain(\sigma), \sigma[i].mn(a) \neq 0$$

Definition 4.4 (Undelivered messages)**.** No messages are left in transit.

$$NoUndeliveredMessages \stackrel{def}{\equiv} \forall \sigma \in \mathcal{E}xec, \exists i \in domain(\sigma), \forall j \in domain(sigma), \\ j \geq i \implies \forall e \in MF, \sigma[j].me(e) = 0$$

Definition 4.5 (Safe process)**.** A process is safe if and only if all its sequence flow edges never hold more than one token during their execution.

$$\text{For } p \in N^P, SafePr(p) \stackrel{def}{\equiv} \forall \sigma \in \mathcal{E}xec, \forall i \in domain(\sigma), \forall e \in R(p) \cap E, (\sigma[i].me(e) \leq 1)$$

Definition 4.6 (Safe Collaboration)**.** A collaboration is safe all its processes are safe.

$$Safe \stackrel{def}{\equiv} \forall p \in N^P, SafePr(p)$$

Definition 4.7 (Process soundness)**.** A process is sound in a state if only its end events hold at most one token, and all the other nodes (ignoring start events) and all the edges are unmarked. Formally, process $p \in N^P$ is sound in state s if and only if the following predicate is true :

$$soundPr(p, s) \stackrel{def}{\equiv} \forall n \in R(p) \cap N, (s.mn(n) = 0) \vee (s.mn(n) = 1 \wedge n \in (N^{EE} \cup N^{SE})) \\ \wedge \forall e \in R(p) \cap E, s.me(e) = 0$$

Definition 4.8 (Message-relaxed sound collaboration)**.** A collaboration is message-relaxed sound if eventually all the processes are sound and it is stable:

$$msgSoundCol \stackrel{def}{\equiv} \forall \sigma \in \mathcal{E}xec, \exists i \in domain(\sigma), \forall j \in domain(\sigma), \\ j \geq i \implies \forall p \in N^P, soundPr(p, \sigma[j])$$

Definition 4.9 (Collaboration soundness)**.** A collaboration is sound if and only if, for all executions, eventually, all the processes involved in the collaboration are sound and all the message flow edges are unmarked.

$$soundCol \stackrel{def}{\equiv} \forall \sigma \in \mathcal{E}xec, \exists i \in domain(\sigma), \forall j \in domain(\sigma), \\ j \geq i \implies \forall p \in N^P, soundPr(p, \sigma[j]) \wedge \forall e \in E^{MF}, (\sigma[j].me(e) = 0)$$

With reference to other definitions of soundness [21, 24, 5], we consider a form of soundness under fairness assumptions, that could be called *fair soundness*.

5. Implementation & Verification

In this section, we present our encoding of the FOL semantics in TLA$^+$. This allows one to easily parameter the properties of the communication, and to benefit from the efficient TLC model checker to automatically verify collaborations.

5.1. The TLA$^+$ Specification Language and Verification Framework

TLA$^+$ [9] is a formal specification language based on untyped Zermelo-Fraenkel set theory for specifying data structures, and on the temporal logic of actions (TLA) for specifying dynamic behaviors. TLA$^+$ allows one to specify symbolic transition systems with variables and *actions*. An action is a transition predicate between a state and a successor state. It is an arbitrary first-order predicate with quantifiers, set and arithmetic operators, and functions. In an action, x denotes the value of a variable x in the origin state, and x' denotes its value in the next state. Functions are primitive objects in TLA$^+$. The application of function f to an expression e is written as $f[e]$. The expression $[x \in X \mapsto e]$ denotes the function with domain X that maps any $x \in X$ to e. The expression $[f \text{ EXCEPT } ![e_1] = e_2]$ is a function that is equal to the function f except at point e_1, where its value is e_2. A system specification is usually a disjunction of actions. Fairness, usually expressed as a conjunction of weak or strong fairness on actions, or more generally as an LTL (Linear Temporal Logic) property, ensures progression. Weak fairness on an action states that this action cannot be continuously enabled without being fired, and strong fairness on an action states that this action cannot be infinitely often enabled without being fired. The TLA$^+$ toolbox, freely available at http://lamport.azurewebsites.net/tla/tla.html, contains an editor, a pretty-printer, the TLC model checker, and the TLAPS proof assistant.

5.2. Encoding of FOL Semantics in TLA$^+$

The expression and action fragment of TLA$^+$ contains FOL, and the encoding of the semantics in TLA$^+$ is straightforward (459 lines of TLA$^+$ formulae).

Actually, the given FOL formalisation captures the behavior of each of the components of BPMN (nodes which can be events, activities, and gateways), and thus the behavior of the whole BPMN model. This behavior is defined using the concept of tokens which move from nodes to edges (and vice-versa) when specific conditions are fulfilled. The distribution of those tokens (marking) on the elements of the BPMN model describes its state. Hence, the whole behavior is seen as a set of states reachable when specific transitions are fired.

As extensively described before, the idea here is to associate to each node (n) two predicates[4]: a first predicate, St(n), which states if the node can start its execution, and thus changing its current marking before its execution (mn (n)) to another marking after its execution (mn' (n)). The second predicate, Ct(n), that states if the node can finish its execution and so change the current marking (mn (n)) before its termination to another marking (mn' (n)) after its termination.

In this context, the TLA$^+$ specification of the semantics of node n is nothing but a direct translation of the St(n) and Ct(n) predicates of node n into the TLA$^+$ syntax.

Example.

Let us reconsider the semantics of a "none start event" defined through the predicate Ct(n):

$$\begin{aligned}\forall n \in N^{NSE}, Ct(n) \stackrel{def}{\equiv}\ & (mn(n) = 1) \wedge (mn'(n) = mn(n) - 1) \\ & \wedge \forall e \in out^{SF}(n), (me'(e) = me(e) + 1) \\ & \wedge \left(\begin{array}{l} (\exists p \in N^{P}, (n \in R(p)) \wedge (mn(p) = 0) \\ \qquad \wedge (mn'(p) = 1) \wedge \triangle(\{n, p\} \cup out^{SF}(n)) \wedge \Xi) \\ \vee\ (\exists p \in N^{SP}, (n \in R(p)) \wedge \triangle(\{n\} \cup out^{SF}(n)) \wedge \Xi) \end{array} \right)\end{aligned}$$

This FOL semantics is translated into a TLA$^+$ specification as follows.

$$\begin{aligned} nonestart\_complete(n) \triangleq\ & \wedge\ CatN[n] = NoneStartEvent \\ & \wedge\ nodemarks[n] >= 1 \\ & \wedge\ \text{LET } p == ContainRelInv(n) \text{ IN} \\ & \quad \vee\ \wedge CatN[p] = Process \\ & \qquad \wedge\ nodemarks[p] = 0 \\ & \qquad \wedge\ nodemarks' = [nodemarks \text{ EXCEPT } ![n] = @ - 1, ![p] = 1] \\ & \quad \vee\ \wedge CatN[p] = SubProcess \\ & \qquad \wedge\ nodemarks' = [nodemarks \text{ EXCEPT } ![n] = @ - 1] \\ & \wedge\ edgemarks' = [e \in \text{DOMAIN } edgemarks \mapsto \\ & \qquad \text{IF } e \in outtype(SeqFlowType, n) \text{ THEN } edgemarks[e] + 1 \\ & \qquad \text{ELSE } edgemarks[e]] \\ & \wedge\ Network!unchanged \end{aligned}$$

Intuitively, the translation is done syntactically as shown in Table 2. Here, we can easily observe that the translation is straightforward, and this holds for all the elements of BPMN.

The resulting theories, for the translation of the whole elements, are available at [6] under `theories/tla`.

Module `PWSTypes` defines the abstract constants that correspond to the node and edge types. Module `PWSDefs` specifies the constants that describe a BPMN graph (Definition 2.1): `Node` (for N), `Edge` (for E), `Message` (for $\mathbb{M}$), `CatN` (for cat_N), `CatE` (for

[4]In some cases, the behavior is described using only one of the two predicates.

Table 2: Translation between FOL and TLA$^+$ (*NSE* example).

FOL Expression		TLA$^+$ expression	
ϕ_1	$cat_N(n) = NSE$	ϕ'_1	$CatN[n] = NoneStartEvent$
ϕ_2	$mn(n) \geq 1$	ϕ'_2	$nodemarks[n] >= 1$
ϕ_3	$\forall e \in out^{SF}(n),$ $(me'(e) = me(e) + 1)$	ϕ'_3	$edgemarks' = [e \in$ DOMAIN $edgemarks \mapsto$ IF $e \in outtype(SeqFlowType, n)$ THEN $edgemarks[e] + 1$ ELSE $edgemarks[e]$]
ϕ_4	$\exists p \in N^P, (n \in R(p))$ $\wedge\ (mn(p) = 0)$ $\wedge\ (mn'(n) = mn(n) - 1)$ $\wedge\ (mn'(p) = 1)$	ϕ'_4	LET $p == ContainRelInv(n)$ IN $\wedge\ CatN[p] = Process$ $\wedge\ nodemarks[p] = 0$ $\wedge\ nodemarks' =$ $[nodemarks$ EXCEPT $![n] = @ - 1, ![p] = 1]$
ϕ_5	$\exists p \in N^{SP}, (n \in R(p))$ $\wedge\ (mn'(n) = mn(n) - 1)$	ϕ'_5	LET $p == ContainRelInv(n)$ IN $\wedge\ CatN[p] = SubProcess$ $\wedge\ nodemarks' =$ $[nodemarks$ EXCEPT $![n] = @ - 1]$
ϕ_6	Ξ	ϕ'_6	$Network!unchanged$

cat_E), `ContainRel` (for R)... This module also defines auxiliary functions such as in^T, defined in TLA$^+$ as the operator $intype(type, n) \triangleq \{e \in \{ee \in Edge : target(ee) = n\} : CatE[e] \in type\}$.

Module `PWSWellFormed` encodes the well-formedness predicates for BPMN graphs. For instance, the rule C3 (a sub-process has a unique start event node) becomes:

$$C3\_SubProcessUniqueStart \triangleq \forall n \in Node : CatN[n] = SubProcess \Rightarrow Cardinality(ContainRel[n] \cap \{nn \in Node : CatN[nn] \in StartEventType\}) = 1$$

Last, module `PWSSemantics` contains the semantics. It defines the variables for the marking: `nodemarks` ($\in [Node \to Nat]$), `edgemarks` ($\in [Edge \to Nat]$), and `net` (whose type depends on the selected communication model). Then it contains a translation of the FOL formulas, where each rule yields one TLA$^+$ action, translated from the FOL semantics as explained above.

The *Next* predicate specifies a possible transition between a starting state and a successor state. It is a disjunction of all the actions. The full specification is then, as usual in TLA$^+$, $Init \wedge \Box[Next]_{var} \wedge Fairness$, where *Init* specifies the initial state (Def. 2.3), and $\Box[Next]$ specifies that *Next* (or stuttering) is verified along all the execution steps.

5.3. Fairness in Loops and Alternatives

The restriction to fair executions (Section 2.2.6) is naturally translated in TLA$^+$. TLA$^+$ supports weak and strong fairness, defined as below for an action A:

$$WF_e(\mathcal{A}) \stackrel{def}{\equiv} \Box\Diamond\neg(\text{ENABLED}\langle\mathcal{A}\rangle_e) \vee \Box\Diamond\langle\mathcal{A}\rangle_e$$
$$SF_e(\mathcal{A}) \stackrel{def}{\equiv} \Diamond\Box\neg(\text{ENABLED}\langle\mathcal{A}\rangle_e) \vee \Box\Diamond\langle\mathcal{A}\rangle_e$$

Fairness is then a conjunction of weak fairness on all actions ($\forall n \in Node : WF_{var}(step(n))$), and of strong fairness on XOR, OR and EB transitions.

5.4. Communication as a Parameter

One of the objectives of our FOL semantics is to be able to specify the communication behavior as a parameter of the verification. To achieve this, all operations related to communication are isolated in a `Network` module. This module is a proxy for several implementations that correspond to communication models with different properties, such as their delivery order.

Generic Communication Models. We provide seven communication models which differ in the order messages can be sent or received, and are all the possible point-to-point models when considering local ordering (per process), causal ordering, and global ordering (absolute time). Their formal description is provided above in section 2.2.4, and their formal analysis and comparison are in [12].

The state of the communication model is specified with a variable `net`, whose content depends on the communication model. The communication actions are two transition predicates `send` and `receive` which are true when the action is enabled. These actions take three parameters, the sender process, the destination process and the type of message. Their specification depends on the communication model and is a direct translation in TLA$^+$ of the FOL formula of table 1. For instance, `NetworkFifo` specifies a communication model where the delivery order is globally first-in first-out: messages are delivered in the order they have been sent. Its realization is a queue and the two predicates are:

$$\begin{aligned} send(from, to, m) &\triangleq net' = Append(net, \langle from, to, m\rangle) \\ receive(from, to, m) &\triangleq net \neq \langle\rangle \wedge \langle from, to, m\rangle = Head(net) \wedge net' = Tail(net) \end{aligned}$$

Ad-hoc Communication Models. The communication models described in 2.2.4 are all monolithic. This means that all the communication interactions are handled by the same communication model, and that it restricts the receptions in the same way for all communication channels. In some cases, one needs to have different properties in different parts of a model. For instance, a set of processes can require Fifo All communication for their interactions, while another set does not have any constraint. Using a modular communication framework based on micromodels [25], we offer the possibility to implement the *send* and *receive* predicates with a combination of micromodels that are applied to subsets of the channels in the BPMN collaboration. The available micromodels are : the seven ordering model as above, that order the receptions with regard to the emission events; a micromodel where priorities are assigned to channels; a message cap micromodel that limits the number of messages in transit; a bounded micromodel that limits the total number of messages that a set of channels can transport in an execution.

Consider the example in Figure 1. This example has an infinite number of states as the travel agency can send an arbitrary number of offer. Moreover, it is required that the `NoMore` message is received after all the `Offer` messages. With the monolithic communication models, this can be handled by using the Fifo Pair (or Fifo All) communication model. However, observe that the `Confirmation` and `Ticket` messages are expected by the customer in the reverse order of their emissions. Imposing fifo ordering means that the ticket cannot be delivered before the confirmation, and this collaboration with a Fifo communication model is unsound: the customer process deadlocks without reaching the final state.

Using micromodels, we state that all the channels are point-to-point (micromodel "p2p"), and we impose fifo ordering only on `Offer` and `NoMore` (micromodel "fifo11").

Moreover, we limit the overall number of offers by limiting the number of messages sent on `Offer` (micromodel "voting"). Lastly, we can also bound the number of messages in transit (micromodel "message_cap"). This ad-hoc communication model is described as below:

```
CHANNELS == {"Offer", "NoMore", "Travel", "Payment", "Abort",
             "Ticket", "Confirmation"}
COMMODELS == {[name ↦ "p2p", params ↦ [chan ↦ CHANNELS ] ],
              [name ↦ "fifo11", params ↦ [chan ↦ {"Offer","NoMore"} ] ],
              [name ↦ "voting", params ↦ [chan ↦ {"Offer"}, bound ↦ 2 ] ],
              [name ↦ "message_cap", params ↦ [chan ↦ CHANNELS, bound ↦ 4 ] ] }
COM == INSTANCE multicom WITH
              PEERS ← {"Customer", "Travel Agency"}
              COM ← COMMODELS,
              CHANNEL ← CHANNELS
```

5.5. Mechanized Verification

A specific BPMN diagram is described by instantiating the constants in `PWSDefs` (`Node`, `Edge`...) from the BPMN collaboration. This is automated using our `fbpmn` tool. Regarding the well-formedness of the BPMN diagram, the predicates from `PWSWellFormed` are *assumed* in the model. Before checking a model, The TLA$^+$ model checker checks these assumptions with the instantiated constants that describe the diagram, and reports an error if an assumption is violated. Otherwise, this proves that the diagram is well-formed.

The TLA$^+$ model checker, TLC, is an explicit-state model checker that checks both safety and liveness properties specified in LTL. This logic includes operators $\Box$ and $\Diamond$ that respectively denote that, in all executions, a property F must always hold ($\Box F$) or that it must hold at some instant in the future ($\Diamond F$). TLC builds and explores the full state space of the diagram to verify if the given properties are verified. These properties are generic properties related to any Business Process diagram, or specific properties for a given diagram. Some generic properties are safe collaboration, sound collaboration and message-relaxed sound collaboration [5].

A collaboration is safe if no sequence flow holds more than one token:

$$\Box(\forall e \in E^{SF}, me(e) \leq 1) \tag{1}$$

A collaboration is sound if all the processes are sound and there are no undelivered messages. A process is sound if it is in a stable state where there are no tokens on its inside edges, and no tokens on its nodes, except possibly for start and end events.

$$\begin{aligned} SoundProc(p) \stackrel{def}{=} & \forall e \in R(p) \cap E^{SF}, me(e) = 0 \\ & \wedge \forall n \in R(p) \cap N, (mn(n) = 0 \vee (mn(n) = 1 \wedge n \in (N^{EE} \cup N^{SE}))) \\ Soundness \stackrel{def}{=} & \Diamond\Box(\forall p \in N^P, SoundProc(p) \wedge \forall e \in E^{MF}, me(e) = 0) \end{aligned} \tag{2}$$

A collaboration is message-relaxed sound if it is sound when ignoring messages in transit, *i.e.*, when ignoring the Message Flow edges.

$$MsgRelaxedSoundness \stackrel{def}{=} \Diamond\Box(\forall p \in N^P, SoundProc(p)) \tag{3}$$

Other generic properties are available, such as the absence of undelivered message or the possible activation which states there does not exist a task node (Abstract Task, Send Task, Receive Task) that is never activated in any execution. From a business process point of view, it means that there are no tasks in the diagram that are never used. This is expressed as $\forall n \in N^T : \mathbf{EF}(mn(n) \neq 0)$ (TLC can check for the invalidity of the negation of this CTL formula).

Last, the user can also define business model properties concerning a specific diagram. For instance, one can check that the marking of a given node is bounded by a constant (*i.e.*, $\Box(nodemarks[\text{"}Confirm\ Booking\text{"}]) \leq 1$), or that the activation of one node necessarily leads to the activation of another node ($\Box(nodemarks[\text{"}Book\ Travel\text{"}] \neq 0 \Rightarrow \Diamond(nodemarks[\text{"}Offer\ Completed\text{"}] \neq 0)$).

Termination of the verification is ensured for a finite state model. When the model checker finds that a property is invalid, it outputs a counter-example trace that we animate on the BPM graphical model to help the user understand it. TLC uses a breadth-first algorithm, and this trace is minimal for safety properties. As any BPMN model is structurally finite, a model with an infinite state space is necessarily unsafe (in the sense of (1): some edges hold more than one tokens). This property is invalidated on a prefix of a trace. TLC incrementally checks invariants during the construction of the state space, and an unsafe model will be detected even if it would yield an infinite state space. TLC cannot check arbitrary properties on an infinite state model. Nevertheless, we can use constraints expressed on states or transitions to limit the state space (see Section 6.4.1).

5.6. Experiments

Experiments were conducted on a laptop with a 1.9 GHz (turbo 4.8 GHz) Intel Core i7 processor (quad core) with 32 GB of memory. Results are presented in Table 3. The first column is the reference of the example in our archive. The characteristics of a model are: number of participants, number of nodes (incl. gateways), number of flow edges (sequence or message flows), whether the model is well-balanced (for each gateway with n diverging branches we have a corresponding gateway with n converging branches) and whether it includes a loop. The communication model is asynchronous (bag), fifo-ordered between each couple of processes (fifo pair), globally fifo (fifo all), or synchronous-like (RSC). The results of the verification then follow. First, data on the resulting transition system are given: number of states, number of transitions, and depth (length of the longest sequence of transitions that the model checker had to explore). For each of the three correctness properties presented above, we indicate if the model satisfies it. Lastly, the accumulated time for the verification of the three properties is given. Our tool supports more verifications (see Table 6) and can be easily extended with new properties. We selected these three ones since they are more BPMN specific [5].

Table 3 presents the results for a selection from our repository [6] for a variety of gateways and activities. These illustrative examples include realistic business process models (001 and 002 two client-supplier models, 040 from Figure 1, 017 from [4], and 020 from [26]), and models dedicated to specific concerns: termination end events and sub processes (007–011 from [5]), inclusive gateways (003, 012, 013 and 018), exclusive and event-based gateways (015 and 016).

A first conclusion is that verification is rather fast: the verification of one property generally takes just a few seconds per model, the longest being for model 020 that takes

Table 3: Experimental Results.

ref.	Characteristics					Com.	LTS size			validity			total
	proc.	nodes (gw.)	SF/MF	B	L	model	states	trans.	depth	(1)	(2)	(3)	time
001	2	17 (2)	14/3	✓	×	bag	93	173	25	✓	✓	✓	3.60s
						fifo pair	85	161	21	✓	×	×	3.26s
						RSC	77	147	19	✓	×	×	3.66s
002	2	16 (2)	13/3	✓	×	bag	79	147	23	✓	✓	✓	3.57s
						fifo pair	71	135	19	✓	×	×	3.61s
						RSC	63	121	17	✓	×	×	3.56s
003	1	14 (6)	16/0	×	✓	none	41	59	15	✓	✓	✓	3.10s
006	2	20 (4)	18/5	×	✓	bag	470	966	43	×	×	✓	4.20s
						fifo all	522	932	40	×	×	×	4.87s
						RSC	247	420	38	×	×	×	4.13s
007	1	8 (2)	7/0	×	×	none	44	73	15	×	×	×	2.52s
008	1	11 (2)	9/0	×	×	none	48	77	19	×	✓	✓	2.70s
009	2	12 (2)	9/1	×	×	bag	170	395	19	×	×	×	3.70s
010	2	15 (2)	11/1	×	×	bag	186	423	23	×	×	✓	3.72s
011	2	15 (2)	11/1	×	×	bag	100	209	21	×	✓	✓	3.51s
012	1	15 (8)	17/0	✓	✓	none	71	137	15	✓	✓	✓	3.85s
013	1	17 (8)	21/0	✓	✓	none	407	1049	15	✓	✓	✓	5.93s
018	1	19 (8)	25/0	✓	✓	none	4631	15513	18	✓	✓	✓	30.23s
015	2	14 (2)	10/2	×	×	bag	68	117	11	✓	×	×	3.11s
016	2	14 (2)	10/2	×	×	bag	36	53	11	✓	✓	✓	3.17s
017	1	32 (12)	36/0	×	×	none	93	141	37	✓	✓	✓	4.03s
020	4	39 (6)	34/8	×	✓	bag	3558	11035	52	✓	✓	✓	20.76s
020						fifo all	2138	5654	52	✓	✓	✓	14.57s
020						RSC	1030	2695	52	✓	×	×	10.26s
040	2	29 (3)	23/7	✓	✓	bag	353	712	38	✓	×	✓	6.26s
040						fifo	297	616	35	✓	×	×	6.22s
040						ad-hoc p.45	657	1154	39	✓	×	✓	9.5s

up to 53s of accumulated time for the three properties (5s for the construction of the state space). Experiments also show the effect of the communication model on property satisfaction (models 1, 2, 6, 20), the use of TLA$^+$ fairness to avoid infinite loops (12, 13, 18, 20), and the use of terminate end events combined with model constraints (see Section 6.4.1) to deal with unsafety (6).

LTL verification is $O(M * 2^F)$, where M is the size of the state space, and F is the "size" of the formula (the number of involved temporal operators). F is mainly influenced by the number of fairness constraints. Regarding M, in practice, more than the size of the BPMN schemas, interleaving is the main cause of state explosion. Interleaving is directly linked to the number of processes. Thus, more than the number of nodes (which has a limited impact), the verification time is mainly impacted by the number of processes and the branching in them.

6. The fbpmn Tool Suite

Our FOL formal semantics for communication parametric BPMN collaboration, and the verification of properties, realized over TLA$^+$ and the TLC model checker, as presented in Section 5, are implemented in the fbpmn tool suite. This open source, freely available, suite is made up of :

- the fbpmn program and several accompanying scripts, to perform verification in one's command line and to graphically animate counter examples,
- a Web application version of the above, with a client-side front-end (for BPMN modelling and for giving communication and verification parameters) that runs in one's browser, and a server-side back-end verification engine, built around fbpmn and scripts, for which a Docker version is available.

The URL to get fbpmn is `https://github.com/pascalpoizat/fbpmn`.

6.1. Architecture and General Principles

The fbpmn tool suite is centered around a command, fbpmn, that is available for Linux, OSX, and Windows (binaries are available for the first two, the later requiring, for now, a compilation process). This command is used to transform a BPMN model into a TLA$^+$ representation of its BPMN Graph (Def. 2.1). fbpmn is also in charge of the computation of the Pre_N and Pre_E sets that are used in the semantics of the *OR* gateways, since these two sets can be structurally computed from the BPMN graph structure. This generated TLA$^+$ module is then passed, together with modules for TLA$^+$ implementation of our well-formedness rules (Sect. 2.1) and of semantics (Sect. 2.2), to the TLC model checker, as described in the bottom of Figure 34.

In case verification fails, TLC outputs a counter-example in the form of a trace that includes for each step the state of the markings and the communication network (Def. 2.2). In order to ease the interpretation of this by the process designer, fbpmn can also be used to generate an interactive animation of the counter-example, where one can see the marking over the BPMN model and navigate between the steps of the counter example (Fig. 4). The presentation layer for the counter-example animator has been achieved using the Camunda.io javascript library.

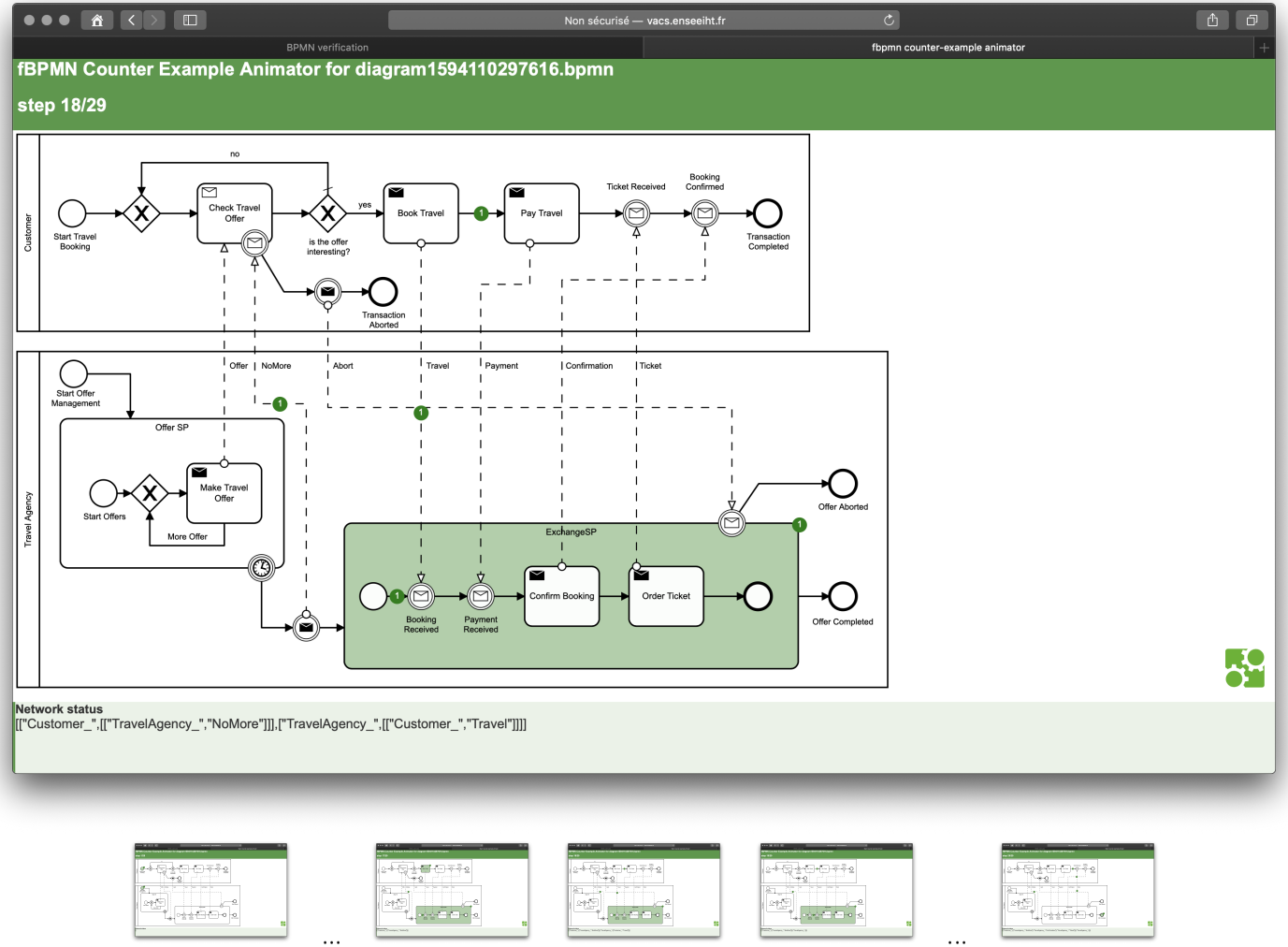

Table 4: Animation of a counter-example (model in Fig. 1, for soundness with Fifo Inbox mode) generated with fbpmn.

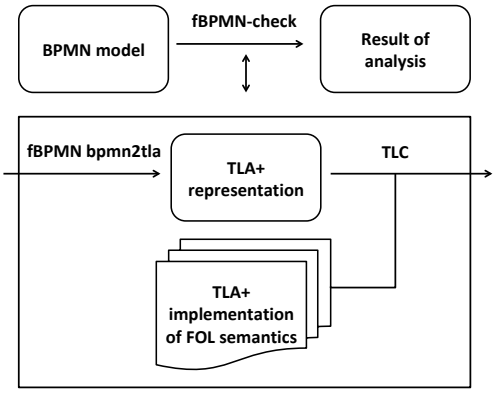

Figure 34: fbpmn approach for the verification of a BPMN collaboration.

6.2. Desktop Modelling and Verification

Since for a given model one may have different properties of interest (*e.g.*, safety, soundness, and message-relaxed soundness), and since several communication models are possible (*e.g.*, the seven ones presented in Sect. 2.2.4), it would be tedious to run fbpmn for each of the combinations. Hence, we provide the process designer with scripts (only under Linux or OSX) that ease verification.

When the designer launches the fbpmn-check script (upper part of Fig. 34), it reads a configuration directory and runs fbpmn based on the designer preferences. Let us suppose the configuration directory is as follows.

```
Network01Bag.tla       Network04Inbox.tla     Network07RSC.tla       Prop03Sound.cfg
Network02FifoPair.tla  Network05Outbox.tla    Prop01Type.cfg         Prop04MsgSound.cfg
Network03Causal.tla    Network06Fifo.tla      Prop02Safe.cfg
```

This will yield four different properties to be checked for seven different network models, generating at most 28 counter-example traces. Running the fbpmn-logs2html script on a working directory generated by fbpmn-check, finds out these counter-examples and generates an interactive animator for each of them. It is also possible to give fbpmn-check a number of cores to use, this value being passed to the TLC model checker.

6.3. Online Modelling and Verification

In order to ease the use of the fbpmn tool suite, we have implemented a Web application for it (Fig. 35).

There, the user can import, design, or export a BPMN model (this is achieved using the Camunda.io framework). Then verification parameters can be given: which properties to check, which communication models to check with, possibly model constraints (see below) for nodes and/or edges.

After retrieving the results (Fig. 36) the user has the possibility to see a textual version of counter-examples and/or to animate it on the model as presented in Sect. 6.1.

The fbpmn Web application is available at `http://vacs.enseeiht.fr/bpmn/` for demonstration purposes. Yet, if one is interested in it, we advocate the deployment of

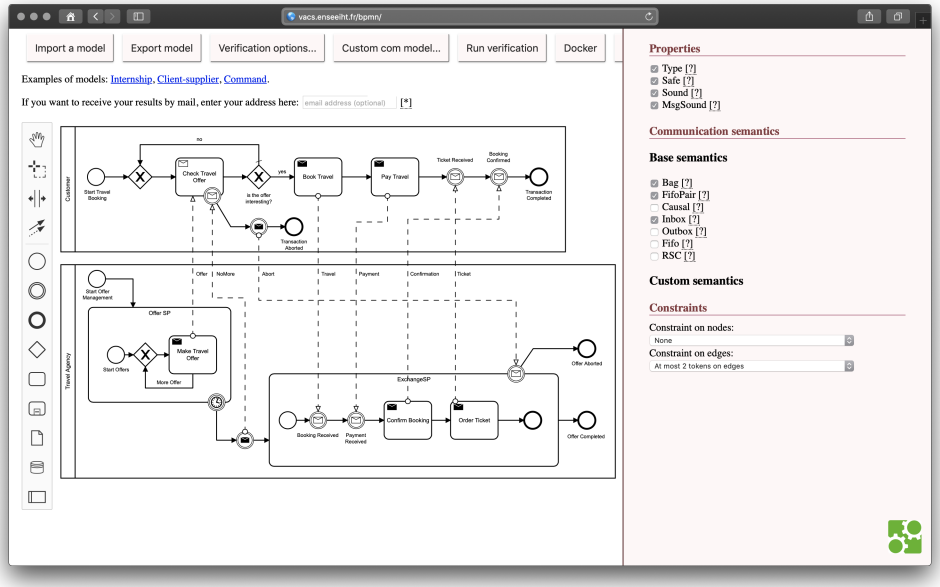

Figure 35: fbpmn Web application (modelling and verification panel).

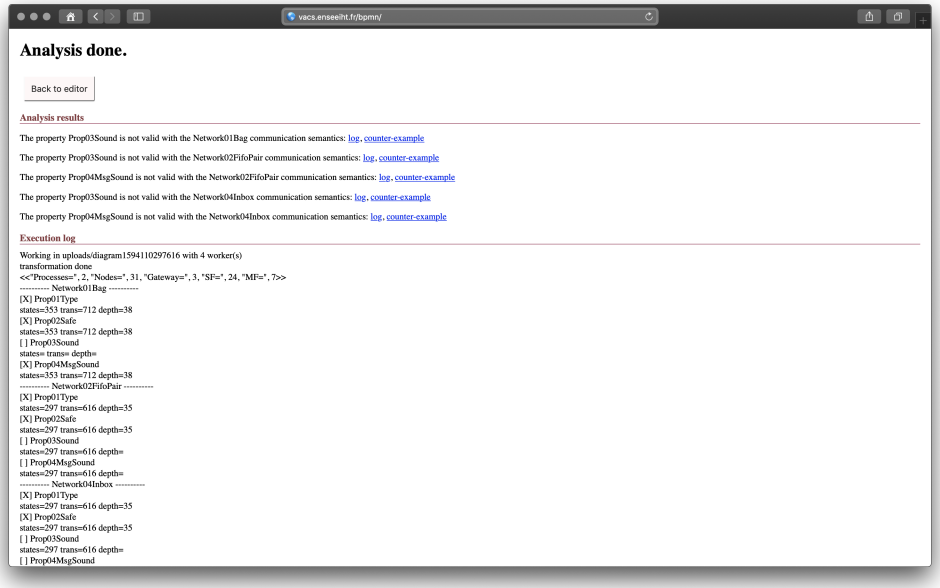

Figure 36: fbpmn Web application (verification results).

it on one's own machine or server. For this, we provide a Docker image, downloadable from our Web application.

6.4. Extensibility

Our framework can be extended as far as safeness constraints, properties to check, and communication models are concerned.

6.4.1. Model Constraints

Some models are unsafe, *i.e.*, the semantics can yield an infinite marking on some node(s) or edge(s). In such a case, one may rely on model constraints associated to the BPMN model to be verified. Given the model is model.bpmn, one just has to create a file model.constraint of the form:

```
CONSTANT ConstraintNode <- <ConstraintOnNodes>
         ConstraintEdge <- <ConstraintOnEdges>
         Constraint <- <Overall constraint in terms of ConstraintNode and ConstraintEdge>
```

Some node constraints and edge constraints are already defined in our TLA$^+$ library, *e.g.*, the one to state that an edge should have at most 2 tokens on it, MaxEdgeMarking2, or the one to limit the number of tokens only on message edges MaxMessageEdgeMarking2. The most usual constraint combinator is also already defined there, ConstraintNodeEdge, that is the conjunction of the user specified node and edge constraints. Using this, we may verify model 006 (as seen in Sect. 5.6), defined in file e006TravelAgency.bpmn, with a file e006TravelAgency.constraint:

```
CONSTANT ConstraintNode <- TRUE
         ConstraintEdge <- MaxEdgeMarking2
         Constraint <- ConstraintNodeEdge
```

The user is free to extend our constraint library by extending the PWSConstraints.tla TLA$^+$ module.

6.4.2. New Properties

We support several properties from the literature. However, it is possible to extend this set. To do so, one has to:

1. define a new property, say MyProperty, at the end of the main TLA$^+$ semantic module, PWSSemantics.tla
2. create a new file PropNNMyProperty.cfg in the fbpmn configuration directory, with NN being a number different from the existing properties there
3. in the contents of PropNNMyProperty.cfg refer to the property name given in step 1.

The definition of new properties has some limitations. First it must be possible to define them using LTL since this is the logic that is verified by TLC. Second, these properties must cope with our definition of state (Def. 2.2), *i.e.*, they can be defined in terms of node markings, edge markings, and/or network markings. Properties can also refer to the types of the nodes and edges, as demonstrated in Sect. 5.5 for the soundness property.

6.4.3. New Communication Models

As stated before, we support the most usual communication models to be used as parameters for the BPMN semantics. Still, one may define new models. In order to achieve this, one has to:

1. define the new communication model semantics, say MyNet, in a NetworkMyNet.tla file in the fbpmn TLA^+ theories directory
2. copy one of the files in the fbpmn configuration directory to a new file NetworkNN-MyNet.tla in the same directory, with NN being a number different from the existing communication models there
3. in the contents of NetworkNNMyNet.tla, change the line of the network implementation definition to refer to the new communication model as defined in step 1.

7. Related Work

The formalization of the BPMN execution semantics, and on a wider scale the formal study of business processes, is an active field of research. In this section, we discuss the most significant propositions in formalizing BPMN. We then compare our approach to them. In our selection, we focus on two criteria: (1) the use of BPMN as a modelling language for describing the business process; (2) the existence of tooling support for the developed approach.

Table 5: Verification tools in the literature.

Diagram	Features	Transformation artifact	References	Year	BPMN	Tool for transformation	Tool for verification
Process		PN	[27]	2011	1.0	InFlux	Woflan , LoLA
		Kripke structure	[28]	2013	2.0	BPMN2SPIN	EpiSpin
	Data	PN	[29]	2014	2.0	BPMN2PN prototype	LoLA
		LTL	[30]	2014	2.0	BPVSL	–
		Markov decision process	[31], [32]	2016	2.0	SBOAT	PRISM
		WF-nets	[33]	2016	2.0	InFlux	WFlan
		LNT (LOTOS)	[34][35]	2017	2.0	VBPMN	CADP
	Time	Maude source code	[36],[37]	2017-2018	2.0	BPMN-P	Maude Checker
		CPN	[38]	2019	2.0	GROOVE	CPN Tool
Collab.		PN	[39], [7]	2007-2008	1.0	Transformer	ProM
		YAWL	[40], [41]	2008-2010	1.0	BPMN2YAWL	–
	Time	CSP	[42, 43, 44], [45], [46]	2008-2012	2.0	BTRANSFORMER	FDR2
		LTL	[47]	2012	2.0	Prototype	–
		Graph rewriting rule	[48], [49]	2010-2013	2.0	GrGen	–
	Data	Maude source code	[50]	2014	2.0	Prototype	Maude checker
		RECATNets	[51], [52]	2016-2017	2.0	BPMNChecker	Maude
		LTS	[53], [54]	2017-2018	2.0	BProve	Maude checker
		CPN	[8], [55]	2018-2019	2.0	CP4BPMN	CPN Tool
	Time	TPN	[23]	2019	2.0	Transformer	–

In Table 5, column *Transformation artifact* shows the formalism in which the BPMN models are expressed, column *Tool for transformation* presents the supporting tool for applying the approach, and column *Tool for verification* presents the tool that is used for the verification. The upper part of the table lists approaches dealing only with process diagrams, while the lower part describes approaches dealing with collaboration diagrams.

In the upper part of the table, the discussed approaches formalize a minimal subset of BPMN 2.0: parallel and exclusive gateways, start and end events, and sequence flows. The inclusive gateway is treated only by a subset of these works. All but two approaches rely on a third-party formalism (*i.e.*, they transform the BPMN models to another language) to conduct the verification. Such an idea has the advantage to reuse

the existing results and tools with a minimal effort, but has the main drawback of being very restrictive with respect to the execution semantics of the target formalism.

From a time-related perspective, we consider the approach of [36, 37] to be very promising. First, it implements a pure semantics to BPMN without using any transformation. Second, it can deal with time constraints of the model using high level stochastic expressions. However, the approach extends the behavior of the standard BPMN elements (*e.g.*, timeouts for tasks and delays for sequence flow, probabilities to various forms of branching behavior in gateways) to treat the time constraints, and did not stick to the actual behaviors defined in the standard. With respect to our work, we provide a semantics for the timer BPMN elements as they are expressed in the standard.

The bottom part of the table presents work closer to ours with regard to the supported BPMN elements. Some works are based on a transformation approach, while others provide a direct formalization semantics.

Petri nets are used to formalize and verify correctness and soundness properties of BPMN in several works. Among them, the earliest work that we identified is [7]. The authors propose a transformation from BPMN 1.2 into Petri nets models. They have chosen the ILog BPMN Modeller as a graphical editor to create BPMN models. The authors apply the defined transformation approach on BPMN models and export the resulting Petri net in the form of a PNML file. Then, they use the resulting files as input for the verification tool to statically check the semantics of BPMN models. With the ProM framework, soundness is verified, as well as other properties such as dead transitions, deadlocks, and livelocks. Even if the work deals with collaboration elements, the formalization as Petri nets suffers from limitation for the presentation of the communication, the hierarchical relation between processes and sub-processes, and the verification of internal activities within them. Recently, the work in [23] has extended the approach of [7] to timed Petri nets to be able to express time constraints for activities, process regions, and timer events. Besides, they tackle the detection and management of constraint violations at run-time. The authors propose this formal model to analyze the soundness of the collaboration and of its temporal constraints. However, no formal verification of these properties is given.

In [8], the authors present a transformation approach of BPMN models into colored Petri nets (CPN). They have developed a tool called Coloured Petri net for BPMN design (CPN4BPMN) to automatize the transformation. Their approach is defined trough two phases: before applying the transformation, a BPMN design tool (such as the BPMN 2.0 Designer of the Eclipse IDE) is used to partition a BPMN model into sub-models. Then, they use the sub-models as input for the CPN4BPMN tool to generate the corresponding CPN model. Even if their work covers a large set of BPMN elements, (*e.g.*, the boundary events, sub-processes, communication elements), the message exchanges are not specified. Indeed, the partitioning technique does not support unstructured BPMN models.

In [52, 51], the authors present a formal verification approach for BPMN based on high-level Petri nets, called RECATNets. The authors provide a prototype to perform the automatic transformation from BPMN to RECATNets. Then the obtained RECATNets are translated into rewriting logic terms, and the Maude model-checker is used to verify proper termination and some other LTL properties. In the papers, no information about the communication model is given, and only a small set of BPMN elements is covered. Besides, no benchmarking is given, and the approach is illustrated through three simple examples.

Other transformations that have been proposed are those from BPMN to workflow models, *e.g.*, Yet Another Workflow language (YAWL) [40, 41]. YAWL is a language with a strictly defined execution semantics inspired by Petri Nets. In [40, 41], the author provide a formal semantics of BPMN models in terms of a transformation to YAWL nets. They use the ILog BPMN Modeler as a graphical editor tool to create BPMN models. They have implemented an open-source transformer plug-in called, BPMN2YAWL, and integrated it into the ProM platform. This tool transforms a BPMN model into a YAWL one and helps to export the YAWL model as an XML file. This XML file then serves as input to a YAWL-based verification tool. As a proof of concept, the tool has been tested using simple models.

In [49], the authors propose a formalization of BPMN 2.0 based on a set of in-place graph transformation rules. Then, they provide the implementation of their proposition in a framework called GrGen.

Process algebra have also been considered to formalize and verify BPMN processes [42, 43, 44]. The authors provide a formal semantic model for BPMN using Communicating Sequential Processes (CSP) followed by a refinement procedure for property checking. Two models are provided: the first one presents an un-timed model to facilitate the verification of safety and liveness properties. The second one, an extension of the first model, introduces relative timing information. This allows the modelling of concurrent activities under temporal constraints. The authors illustrate the advantages of their approach by means of case studies. Translated BPMN models can be checked with CSP checking tools such as FDR.

Another line of work aims at using symbolic encodings. In [47] for example, the authors give an execution semantics of BPMN elements expressed using linear temporal logic (LTL). The formalization is defined for a large set of elements of BPMN 1.2. This give an unambiguous definition of the execution semantics of BPMN diagrams, and could serve as a basis for the formal analysis of BPMN diagrams, but no tool is presented.

To overcome some issues related to the mapping of BPMN to other formal languages equipped with their own semantics (*e.g.*, non-local effects of BPMN elements such as termination), several recent works have been proposed for verifying BPMN processes based on a direct formalization of its execution semantics.

Rewriting logic and Maude have been proposed to formalize and analyze BPMN processes. In [50], the authors provide a formal semantics for a subset of BPMN models and encode it in Maude for verification purposes. They focus on data objects semantics and their use in database-related decision gateways. Besides, they focus on syntactic issues by introducing the notion of well-formed BPMN processes to avoid them. Unlike this work, we support all process models whether with a structured or non-structured topology.

In [56], the authors provide an operational semantics for a subset of BPMN models in terms of Labelled Transition Systems (LTS). The authors focused on the collaboration diagram elements. Then, in [54], they present a verification tool based on the Eclipse IDE, called Business Process Verifier (BProVe). This tool is based on their proposed operational semantics, implemented using Maude. They use the Maude Linear Temporal Logic (LTL) model-checker to verify properties, and support the verification of safety and soundness properties. In [26], the authors extend their formal framework to include multiple instances and data perspectives. They also provide an associated model animator, called MIDA. It may help the process designers to visualize the behavior of their

models and debug them.

From another perspective, the standard [1] states that a choreography model can be presented through a collaboration. In [57], the authors consider the interaction of business process with a choreography perspective. They introduce a new formalism called interaction Petri nets that aims at capturing the interaction in BPMN choreographies. This paper only considers synchronous communication while leaving asynchronous communication as an open question.

Discussion. Unlike the mentioned works, we have defined the execution semantics of BPMN models in terms of First-Order Logic, with the objective of defining a generic formal semantics that can be implemented into different languages and tools. Nevertheless, the above mentioned approaches have mainly focused on verifying the control flow of business process models against a correctness criterion. To the best of our knowledge, these approaches do not support verification of BPMN models under a specific, and parametric, communication model.

Table 6 complements Table 5 by looking more into detail the BPMN features and properties of interest that are supported by recent works that provide tool support for the verification of collaboration diagrams and communication features in BPMN. This table divides the approaches between those that rely on an intermediary model, and those that have the benefit of providing a direct link between BPMN constructs and the verification formalism. Our work follows this line. Further, the choice of FOL lets one implement the semantics in different tools, *e.g.*, TLA$^+$ as here or SMT solvers. As far as the BPMN coverage criteria is concerned, we can observe that we are among the approaches with a high coverage. The use of FOL to define the semantics, and its implementation in TLA$^+$, made it possible for us to implement quite easily correctness checks as temporal logic properties to be checked against the model.

8. Conclusion & Future Work

In this paper we have proposed a direct formalization in first-order logic for a subset of BPMN that includes sub-process, communication and time constructs. This semantics is parametric with reference to the properties of the communication model, which is of importance since, as seen for example in Table 3, it has an impact on the properties a BPMN model fulfils or not. Together with time, the support for distinct communication models is useful when it comes to use the BPMN standard in contexts such as the IoT [2, 3]. Based on a realization of the first-order logic semantics into TLA$^+$, and the use of the TLC model-checker, we provide business process designers with tool support in order to check domain-specific properties (dedicated either to workflow notations in general, or to BPMN in particular) and animate counter-examples in order to fix erroneous models.

Even if we have provided users with an integrated Web application where one can model, check, and debug business process models, a direct perspective of our work is to integrate it as a plug-in in more general purpose platforms for business processes, such as Apromore [58] or ProM [59].

An ongoing work is the replacement of parameterization using a global communication model (here one in a choice of seven) by a more fine-grained solution based on the communication framework we have developed in [25]. This would support, for example,

Table 6: Comparison of tool-supported approaches for the analysis of communication in BPMN.

	approach	transformation						direct					
	reference	[8]	[51]	[49]	[44]	[40]	[7]	[26]	[5]	[53]	[47]	[50]	**ours**
	year	2018	2017	2013	2011	2008	2008	2018			2012	2014	2020
	formalism	CPN	ECATNets	In-Place Graph	CSP	YAWL Nets	PN	LTS			LTL	Maude	FOL
	tool	not avail. CP4PBMN	not avail. BPMNChecker	yes GrGen	– –	yes BPMN2YAWL	yes transformer	yes MIDA	– –	yes BProVe	– –	not avail. prototype	yes fBPMN
supported elements	*AND* gateway	●	●	●	●	●	●	●	●	●	●	●	●
	XOR gateway	●	●	●	●	●	●	●	●	●	●	●	●
	OR gateway	●	–	●	●	●	●	●	●	●	●	●	●
	EB gateway	●	●	●	–	●	●	●	●	●	●	–	●
	ST	●	●	●	●	●	●	●	●	●	●	●	●
	RT	●	●	●	●	●	●	●	●	●	●	●	●
	MSE	●	●	●	–	●	●	●	●	●	●	–	●
	TMIE	●	–	●	–	–	●	●	●	●	●	–	●
	CMIE	●	–	●	–	–	●	●	●	●	●	–	●
	MBE	–	–	–	●	–	–	–	–	–	●	–	●
	TBE	–	–	–	●	●	–	–	–	–	●	–	●
	TICE	–	●	–	●	●	●	–	–	–	●	–	●
	TSE	–	–	–	–	●	–	–	–	–	●	–	●
	MEE	●	●	●	–	–	–	●	●	●	●	–	●
	TEE	–	–	●	●	–	–	●	●	●	●	–	●
	sub-processes	●	●	●	–	●	●	●	–	–	●	●	●
	data	●	●	●	–	–	–	●	–	–	–	●	–
	multi instance (pools)	–	–	●	–	–	–	●	–	–	–	–	–
	multi instance(activities)	●	●	●	●	●	●	●	–	–	–	–	–
verification	option to complete	–	●	–	–	●	–	–	–	●	–	●	●
	proper completion	–	–	–	–	●	●	–	–	●	–	●	●
	no dead activity	–	–	–	–	●	●	–	–	●	–	●	●
	safety	–	–	–	●	–	–	–	●	●	●	–	●
	collaboration soundness	–	–	–	–	–	–	–	●	–	–	–	●
	msg relaxed soundness	–	–	–	–	–	–	–	●	–	–	–	●
	undelivered messages	–	–	–	–	–	–	–	–	–	–	–	●
	general-purpose	●	●	●	●	●	●	–	●	●	●	●	●
	communication models	–	–	–	–	–	–	–	–	–	–	–	●

the definition of communication policies per participant, per couples of communicating participants, or using delivering priorities over the messages in transit.

For now, time is dealt with fixed values in the model (*e.g.*, the duration of task `t` is 5 units of time, or after a timeout of 15 units of time run some specific subprocess). The extension of parameterization is also interesting as far as time is concerned. This would mean replacing the fixed information about time with intervals (bounded values) or variables (free values). Some collaborations may fulfil properties depending on constraints over the time at which processes are run, or over durations of tasks and timeouts. Retrieving such constraints from the models one verifies, in a synthesis approach, would have an added-value for the business process designers.

Some features that play a role in full-fledged executable collaborations have been discarded here. This is the case of the data (data objects, data stores, assignments, and message payloads) and multi-instance (for activities and pool lanes) constructs.

As far as data are concerned, a direct (and usual) solution is to extend the notion of state with a substitution from variables to values, indexed by process types or process identifiers in case of multi-instance support. This is indeed what we already did for the communication medium (the "substitution" in this case being limited to a single variable, $mnet$). Lets say that the substitution for data is σ. An assignment such as $var := term$ in a process p would then be supported by adding a clause $\sigma'_p = \sigma_p \triangleleft var \mapsto eval_{\sigma_p}(term)$ in the completion rule for tasks, with $\sigma \triangleleft var \mapsto val$ denoting the update of a substitution σ associating value val to variable var, and $eval_\sigma(term)$ the evaluation of $term$ wrt. σ (that gives values to the free variables in $term$). Accordingly, the conditions in exclusive and inclusive gateways can also be treated by requiring that their evaluation yields $true$, *i.e.*, for a condition c, $eval_{\sigma_p}(c) = true$. The treatment for unbound data (*e.g.*, if one wants to verify a process whatever the initialization of the data objects is, or with data stores whose content is unknown), is however much more complicated. This could be tackled using approaches based on symbolic verification techniques [60, 61, 62, 63, 64].

The support for multi-instance constructs requires also a specific treatment. There, the tokens would have to carry process identifiers and our marking functions (mn and me, yielding a value in $\mathbb{N}$) would have to yield a value in a bag (multi-set) of the type for process identifiers. This would be reflected in the semantic rules for the BPMN constructs, *e.g.*, for a merging parallel gateway, one would no longer require that each input edge has at least one token ($me(ei) \geq 1$ in the rule for *AND*, page 21) but instead that each input edge has at least one token for a process identifier of interest. Further, the support for multi-instance BPMN features not only for processes (pool lanes) but also for activities (sub-processes and tasks) would possibly require that the type for process identifiers has a specific indexed structure.

The interplay between data and multi-instance constructs also adds a degree to the complexity of the semantics, as, *e.g.*, data are used in message payloads and a part of these payloads are used as a correlation mechanism. Few approaches are able to deal with data, multi-instance activities and multi-instance pool lanes [49, 26].

Finally, a last perspective, that goes in the direction of spreading the use of lightweight formal verification for BPMN models, is related to the termination of verification. As we have seen, with our proposal, for some models and some properties, one has to bound the (construction of the) semantic state space using constraints. A perspective would be to identify fragments of the formal model for which termination is ensured, and ones for which changes in the BPMN model or bounding constraints are required. Work in

the field of combining control and data perspectives in business process modelling, and performing verification on such expressive models, could be an inspiration here, following what has been done for Data-Aware BPMN [63] and for Catalog Nets [64].

References

[1] OMG Group, Business Process Modeling Notation, http://www.omg.org/spec/BPMN/2.0.2/ (2013).

[2] F. Casati, et al., Towards business processes orchestrating the physical enterprise with wireless sensor networks, in: 34th International Conference on Software Engineering, 2012, pp. 1357–1360.

[3] S. Meyer, A. Ruppen, L. M. Hilty, The things of the Internet of Things in BPMN, in: Advanced Information Systems Engineering Workshops - CAiSE 2015, 2015, pp. 285–297. doi:10.1007/978-3-319-19243-7_27.

[4] A. Krishna, P. Poizat, G. Salaün, Checking business process evolution, Science of Computer Programming 170 (2019) 1–26.

[5] F. Corradini, et al., A classification of BPMN collaborations based on safeness and soundness notions, in: Proceedings of the 25th International Workshop on Expressiveness in Concurrency and of the 15th Workshop on Structural Operational Semantics, Vol. 276 of Electronic Proceedings in Theoretical Computer Science, Open Publishing Association EXPRESS/SOS, 2018, pp. 37–52. doi:10.4204/EPTCS.276.5.

[6] P. Poizat, et al., fbpmn repository, "https://github.com/pascalpoizat/fbpmn", v0.3.4 (2020).

[7] R. M. Dijkman, M. Dumas, C. Ouyang, Semantics and analysis of business process models in BPMN, Information and Software technology 50 (12) (2008) 1281–1294.

[8] C. Dechsupa, W. Vatanawood, A. Thongtak, Transformation of the BPMN design model into a colored Petri net using the partitioning approach, IEEE Access 6 (2018) 38421–38436.

[9] L. Lamport, Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers, Addison Wesley, 2002.

[10] D. Fahland, et al., Instantaneous soundness checking of industrial business process models, in: 7th International Conference on Business Process Management BPM 2009, 2009, pp. 278–293. doi:10.1007/978-3-642-03848-8_19.

[11] S. Houhou, S. Baarir, P. Poizat, P. Quéinnec, A First-Order Logic semantics for communication-parametric BPMN collaborations, in: 17th International Conference on Business Process Management BPM 2019, 2019, pp. 52–68. doi:10.1007/978-3-030-26619-6_6.

[12] F. Chevrou, A. Hurault, P. Quéinnec, On the diversity of asynchronous communication, Formal Aspects of Computing 28 (5) (2016) 847–879. doi:10.1007/s00165-016-0379-x.

[13] L. Lamport, Time, clocks and the ordering of events in a distributed system, Communications of the ACM 21 (7) (1978) 558–565.

[14] R. Schwarz, F. Mattern, Detecting causal relationships in distributed computations: In search of the holy grail, Distributed Computing 7 (3) (1994) 149–174.

[15] A. D. Kshemkalyani, M. Singhal, Necessary and sufficient conditions on information for causal message ordering and their optimal implementation, Distributed Computing 11 (2) (1998) 91–111.

[16] M. Raynal, A. Schiper, S. Toueg, The causal ordering abstraction and a simple way to implement it, Information Processing Letters 39 (1991) 343–350.

[17] B. Charron-Bost, F. Mattern, G. Tel, Synchronous, asynchronous, and causally ordered communication, Distributed Computing 9 (4) (1996) 173–191. doi:10.1007/s004460050018.

[18] M. Abadi, L. Lamport, The existence of refinement mappings, Theoretical Computer Science 82 (2) (1991) 253–284. doi:10.1016/0304-3975(91)90224-P.

[19] M. Abadi, L. Lamport, An old-fashioned recipe for real-time, ACM Transactions on Programming Languages and Systems 16 (5) (1994) 1543–1571. doi:10.1145/186025.186058.

[20] L. Lamport, Real-time model checking is really simple, in: Correct Hardware Design and Verification Methods, CHARME 2005, Vol. 3725 of Lecture Notes in Computer Science, Springer, 2005, pp. 162–175. doi:10.1007/11560548_14.

[21] W. M. P. van der Aalst, Verification of workflow nets, in: International Conference on Application and Theory of Petri Nets ICATPN, Vol. 1248 of Lecture Notes in Computer Science, Springer, 1997, pp. 407–426. doi:10.1007/3-540-63139-9_48.

[22] W. M. Van Der Aalst, K. M. Van Hee, A. H. Ter Hofstede, N. Sidorova, H. Verbeek, M. Voorhoeve, M. T. Wynn, Soundness of workflow nets: classification, decidability, and analysis, Formal Aspects of Computing 23 (3) (2011) 333–363.

[23] C. Combi, B. Oliboni, F. Zerbato, A modular approach to the specification and management of time duration constraints in BPMN, Information Systems 84 (2019) 111–144.
[24] M. Dumas, M. L. Rosa, J. Mendling, H. A. Reijers, Fundamentals of Business Process Management, Second Edition, Springer, 2018. doi:10.1007/978-3-662-56509-4.
[25] F. Chevrou, A. Hurault, P. Quéinnec, A modular framework for verifying versatile distributed systems, Journal of Logical and Algebraic Methods in Programming 108 (2019) 24–46. doi:10.1016/j.jlamp.2019.05.008.
[26] F. Corradini, et al., Animating multiple instances in BPMN collaborations: From formal semantics to tool support, in: 16th International Conference on Business Process Management BPM 2018, 2018, pp. 83–101. doi:10.1007/978-3-319-98648-7_6.
[27] A. Bastias, S. Bihary, S. Roy, An automated analysis of errors for BPM processes modeled using an in-house infosys tool, in: 2011 18th Asia-Pacific Software Engineering Conference, IEEE, 2011, pp. 97–105.
[28] O. M. Kherbouche, A. Ahmad, H. Basson, Using model checking to control the structural errors in BPMN models, in: IEEE 7th International Conference on Research Challenges in Information Science, RCIS, 2013, pp. 1–12. doi:10.1109/RCIS.2013.6577723.
[29] S. von Stackelberg, S. Putze, J. A. Mülle, K. Böhm, Detecting data-flow errors in BPMN 2.0, Open Journal of Information Systems (OJIS) 1 (2) (2014) 1–19.
URL http://nbn-resolving.de/urn:nbn:de:101:1-2017052611934
[30] O. E. Hichami, B. E. E. Mohajir, M. A. Achhab, I. Berrada, R. Oucheikh, Towards formal verification of business process using a graphical specification, in: Third IEEE International Colloquium in Information Science and Technology, CIST, 2014, pp. 12–17. doi:10.1109/CIST.2014.7016587.
URL https://doi.org/10.1109/CIST.2014.7016587
[31] L. T. Herbert, Z. Hansen, P. Jacobsen, SBAT: a stochastic BPMN analysis tool, in: ASME 2014 12th Biennial Conference on Engineering Systems Design and Analysis, American Society of Mechanical Engineers Digital Collection, 2014.
[32] L. T. Herbert, Z. N. L. Hansen, Restructuring of workflows to minimise errors via stochastic model checking: An automated evolutionary approach, Rel. Eng. & Sys. Safety 145 (2016) 351–365. doi:10.1016/j.ress.2015.07.002.
[33] S. Roy, A. S. M. Sajeev, A formal framework for diagnostic analysis for errors of business processes, Transactions on Petri Nets and Other Models of Concurrency 11 (2016) 226–261. doi:10.1007/978-3-662-53401-4_11.
[34] P. Poizat, G. Salaün, A. Krishna, Checking business process evolution, in: International Workshop on Formal Aspects of Component Software, Springer, 2016, pp. 36–53.
[35] A. Krishna, P. Poizat, G. Salaün, VBPMN: Automated verification of BPMN processes (tool paper), in: 13th International Conference on integrated Formal Methods (iFM 2017), Vol. 10510 of Lecture Notes in Computer Science, Springer, 2017, pp. 323–331. doi:10.1007/978-3-319-66845-1_21.
[36] F. Durán, G. Salaün, Verifying timed BPMN processes using maude, in: 19th IFIP International Conference on Coordination Models and Languages, 2017, pp. 219–236. doi:10.1007/978-3-319-59746-1_12.
[37] F. Durán, C. Rocha, G. Salaün, Stochastic analysis of BPMN with time in rewriting logic, Science of Computer Programming 168 (2018) 1–17. doi:10.1016/j.scico.2018.08.007.
[38] S. Meghzili, A. Chaoui, M. Strecker, E. Kerkouche, An approach for the transformation and verification of BPMN models to colored Petri nets models, International Journal of Software Innovation 8 (1) (2020) 17–49. doi:10.4018/IJSI.2020010102.
[39] R. M. Dijkman, M. Dumas, C. Ouyang, Formal semantics and automated analysis of BPMN process models, Preprint, (7115).
[40] J. Ye, S. Sun, W. Song, L. Wen, Formal semantics of BPMN process models using YAWL, in: Second International Symposium on Intelligent Information Technology Application, 2008, pp. 70–74.
[41] J. Ye, W. Song, Transformation of BPMN diagrams to YAWL nets, Journal of Software 5 (4) (2010) 396–404. doi:10.4304/jsw.5.4.396-404.
[42] P. Y. Wong, J. Gibbons, A process semantics for BPMN, in: International Conference on Formal Engineering Methods, Springer, 2008, pp. 355–374.
[43] P. Y. Wong, J. Gibbons, A relative timed semantics for BPMN, Electronic Notes in Theoretical Computer Science 229 (2) (2009) 59–75.
[44] P. Y. Wong, J. Gibbons, Formalisations and applications of BPMN, Science of Computer Programming 76 (8) (2011) 633–650.
[45] M. I. Capel, L. E. Mendoza, Automating the transformation from BPMN models to CSP+ T specifications, in: 35th IEEE Software Engineering Workshop (SEW), IEEE, 2012, pp. 100–109.

[46] A. González, L. E. M. Morales, M. I. Capel, M. A. Pérez, E. M. Méndez, K. Domínguez, BTRANSFORMER - A tool for BPMN to CSP+T transformation, in: 13th International Conference on Enterprise Information Systems ICEIS, 2011, pp. 363–366.

[47] V. S. Lam, A precise execution semantics for BPMN, International Journal of Computer Science 39 (1) (2012) 20–33.

[48] R. Dijkman, P. Van Gorp, BPMN 2.0 execution semantics formalized as graph rewrite rules, in: International Workshop on Business Process Modeling Notation, Springer, 2010, pp. 16–30.

[49] P. Van Gorp, R. Dijkman, A visual token-based formalization of BPMN 2.0 based on in-place transformations, Information and Software Technology 55 (2) (2013) 365–394.

[50] N. El-Saber, A. Boronat, BPMN formalization and verification using Maude, in: Workshop on Behaviour Modelling – Foundations and Applications, BM-FA, 2014.

[51] A. Kheldoun, K. Barkaoui, M. Ioualalen, Formal verification of complex business processes based on high-level Petri nets, Information Sciences 385 (2017) 39–54.

[52] A. Kheldoun, K. Barkaoui, M. Ioualalen, Specification and verification of complex business processes – a high-level Petri net-based approach, in: International Conference on Business Process Management, Springer, 2016, pp. 55–71.

[53] F. Corradini, et al., A formal approach to modeling and verification of business process collaborations, Science of Computer Programming 166 (2018) 35–70.

[54] F. Corradini, F. Fornari, A. Polini, B. Re, F. Tiezzi, A. Vandin, BProVe: Tool support for business process verification, in: International Conference on Automated Software Engineering, IEEE Computer Society, 2017, pp. 217–228. doi:10.1109/ASE.2017.8115635.

[55] C. Dechsupa, W. Vatanawood, A. Thongtak, Hierarchical verification for the BPMN design model using state space analysis, IEEE Access 7 (2019) 16795–16815.

[56] F. Corradini, A. Polini, B. Re, F. Tiezzi, An operational semantics of BPMN collaboration, in: International Workshop on Formal Aspects of Component Software, Springer, 2015, pp. 161–180.

[57] G. Decker, M. Weske, Local enforceability in interaction Petri nets, in: 5th International Conference on Business Process Management BPM 2007, 2007, pp. 305–319.

[58] Apromore, https://apromore.org, accessed: 2020-01-30.

[59] Process Mining Group, Eindhoven University of Technology, Process mining framework (ProM), http://www.processmining.org/prom/start, accessed: 2020-01-30.

[60] H. N. Nguyen, P. Poizat, F. Zaïdi, A symbolic framework for the conformance checking of value-passing choreographies, in: 10th International Conference on Service-Oriented Computing ICSOC 2012, 2012, pp. 525–532. doi:10.1007/978-3-642-34321-6_36.

[61] Y. Li, A. Deutsch, V. Vianu, VERIFAS: A practical verifier for artifact systems, Proc. VLDB Endow. 11 (3) (2017) 283–296. doi:10.14778/3157794.3157798.

[62] F. Durán, C. Rocha, G. Salaün, Symbolic specification and verification of data-aware BPMN processes using rewriting modulo SMT, in: International Workshop on Rewriting Logic and its Applications, Springer, 2018, pp. 76–97.

[63] D. Calvanese, S. Ghilardi, A. Gianola, M. Montali, A. Rivkin, Formal modeling and SMT-based parameterized verification of data-aware BPMN, in: 17th International Conference on Business Process Management BPM 2019, 2019, pp. 157–175. doi:10.1007/978-3-030-26619-6_12.

[64] S. Ghilardi, A. Gianola, M. Montali, A. Rivkin, Petri nets with parameterised data - modelling and verification, in: 18th International Conference on Business Process Management BPM 2020, 2020, pp. 55–74. doi:10.1007/978-3-030-58666-9_4.