Classification of BPMN Collaborations

Flavio Corradini, Chiara Muzi, Barbara Re, and Francesco Tiezzi
School of Science and Technology, University of Camerino, Italy
name.surname@unicam.it

Abstract. BPMN has a huge uptake in modelling business process collaborations in both academia and industry. It results that providing a solid ground to enable BPMN designers to understand their models in a consistent way is becoming more and more important. In our investigation we define and exploit a formal characterisation of the collaborations’ semantics, specifically and directly given for BPMN models, to provide a classification of BPMN collaborations. In particular, we refer to collaborations involving processes with arbitrary topology, thus overcoming the well-structuredness limitations. The proposed classification is based on some of the most important correctness properties, namely safeness and soundness. We prove, with a uniform formal framework, some conjectured and expected results and, most of all, we achieve novel results for BPMN collaborations concerning the relationships between safeness and soundness, and their compositionality, that represent major advances in the state-of-the-art.

1 Introduction

Nowadays, BPMN [1] is the standard language for business process modelling. In particular, the collaboration model is used to describe distributed and complex scenarios where multiple parties interact via message exchange. Even if widely accepted, BPMN major drawback is related to the possible misunderstanding of its execution semantics defined by means of natural text descriptions, sometimes containing misleading information [2]. To fill this gap, much effort has been devoted to reconsider in the business process domain studies related to formal properties on Petri Nets [4] [5], Workflow Nets [6], Condition Event System [7] and Elementary Nets [8]. However, none of them considers BPMN specificities such as different abstraction levels (i.e., collaboration, process, and sub-process levels), asynchronous communication model, and the notion of completeness considering both end and terminate events.

Our investigation is based on a formal characterisation of the semantics of BPMN collaboration models, used to formally define a classification of these models according to relevant properties of the business process domain. It is worth noticing that we do not aim at providing a generic classification approach suitable to different kinds of workflow models, but we specifically focuses on the BPMN notation. To this aim, our formal semantics is directly defined on BPMN elements rather than as an encoding on other formalisms. Our intention is to introduce a solid ground to enable BPMN designers to understand their models, and the properties they enjoy, in a consistent way. Relying on this classification, systematic methodological advices for modelling BPMN diagrams in a correct way can be provided, thus avoiding errors during their enactment.
As a reference formal framework, the proposed classification relies on a direct formalisation of the BPMN semantics we defined. As a distinctive aspect, it supports models with arbitrary topology, including of course also well-structured ones. This is necessary to enable the classification of both structured and unstructured models with respect to the considered correctness properties. Moreover, considering unstructured models is motivated by the fact that model structuredness can only be achieved at the expense of increased model size [9], or it cannot be applicable at all [10] [11], and most of all it would anyway limit BPMN designer freedom [12]. The use of unstructured models in practice, especially when the model size increases, is also confirmed by a study we conducted in a public repository of BPMN models.

Our classification study is based on well-known and relevant properties: well-structuredness [13], safeness [6], and soundness [14] [15]. Despite the large body of work on BPMN, no formal definition of such properties directly given on BPMN is provided, neither for processes nor for collaborations. We believe instead that the direct formal characterisation we provide for such properties is crucial, as it does not leave any room for ambiguity, and increases the potential for formal reasoning. In this respect, a further contribution is the introduction of a novel property, named message-disregarding soundness, that relaxes the soundness notion by considering sound also those collaborations in which asynchronously sent messages are not handled by the receiver (hence, disregarding possible pending messages when the execution completes).

The provided classification relies on the considered properties and their relationship. We demonstrate that a well-structured collaboration is always safe, but the vice versa does not always hold. We also prove that well-structuredness implies soundness only at process level, while there are well-structured collaborations that are not sound. Regarding the relationship of soundness and safeness we prove the existence of unsafe models that are sound. Moreover, we study safeness and soundness compositionality. Finally, we show how the use of some specific BPMN constructs, namely terminate end event and sub-processes, crucially affects the satisfaction of some properties, thus ‘moving’ models from one class to another.

To sum up, the contribution of this paper is to prove the above results using a uniform formal framework. Some of them were only conjectured for BPMN models, or at least expected, as they hold for other workflow languages. Moreover, as we clarify in the next section, other results are completely new. Therefore, the classification we provide for BPMN collaborations significantly advances the state-of-the-art.

The rest of the paper is organised as follows. Sec. 2 provides a first insight into the obtained results and justifies their impact into practices. Sec. 3 provides a running example, while Sec. 4 introduces the proposed formal framework. Sec. 5 provides the definition of properties, and Sec. 6 makes clearer the relationships between these properties. Sec. 7 presents the study on safeness and soundness compositionality. Finally, Sec. 8 discusses related works, and Sec. 9 concludes the paper.

## 2 Classification Results

In this section, we show how BPMN diagrams can be classified according to well-structuredness, safeness, soundness and message disregarding soundness. Differently
from other classifications, typically referred in the business process domain but pro-
poped for Petri Nets [16] and Workflow Nets [17], our study directly addresses BPMN
collaboration models and this has led to novel results. The obtained results are synthe-
ized in the Euler diagram in Fig. 1 showing that:

(i) all well-structured collaborations are safe, but the vice versa does not hold;
(ii) there are well-structured collaborations that are neither sound nor message
disregarding-sound;
(iii) there are sound and message disregarding-sound collaborations that are not safe.

In the following we first discuss how our classification advances the related state of
the art. Then, we make clear the practical relevance of the work.

**Advances with respect to already available classifications.** Result (i) demonstrates
that well-structured collaborations represent a subclass of safe collaborations. The clas-
sification relaxes the existing results on Workflow Nets, where a model to be safe has
to be not only well-structured, but also sound [17]. We formally prove the relationship
between well-structured and safeness directly on BPMN collaborations.

Result (ii) shows that there are well-structured collaborations that are not sound.
This result is completely novel, since the classification differs when passing from the
process to the collaboration level. While for processes it holds that well-structuredness
implies soundness, we formally prove that this result cannot be extended to collabora-
tions. This is a new outcome compared to the one obtained in Petri Nets [16], where
relaxed soundness and well-structuredness together imply soundness, and the one for
workflow processes, where well-structuredness implies soundness [18]. Together with
result (i), this confirms the limits of well-structuredness as a correctness criterion. In-
deed, on the one hand, well-structuredness is too strict, as some safe and sound models
are discarded right from the start; on the other hand, at collaboration level even it fails
to ensure both soundness and message-disregarding soundness.

Also result (iii) is a novel contribution strictly related to the expressiveness of
BPMN and its differences with respect to other workflow languages. In fact, Van der
Aalst shows that soundness of a Workflow Net is equivalent to liveness and bounded-
ness of the corresponding short-circuited Petri Net [19]. Similarly, in workflow graphs
and, equivalently, free-choice Petri Nets, soundness can be characterized in terms of
two types of local errors, viz. deadlock and lack of synchronization: a workflow graph
is sound if it contains neither a deadlock nor a lack of synchronization [20]. Thus, a
sound workflow is always safe. In BPMN instead there are unsafe processes that are
sound and/or message disregarding sound, due to the effects of the terminate end event
and sub-processes. These elements, in fact, impact on the relationship between safeness
and soundness, as shown in the following.
Advance in Classifying BPMN Models. Our BPMN formalisation considers as first-class citizens BPMN characteristics such as the distinction between the collaboration and the process level, the asynchronous communication models, and the notion of completeness distinguishing the end event and the terminate event.

When considering collaboration models, we can observe at the collaboration level pools that exchange message tokens, while at process level the execution is rendered by the movements of the sequence flow tokens. In this setting, there is a clear difference between the notion of safeness directly defined on BPMN collaborations with respect to that defined on Petri Nets and applied to the Petri Nets resulting from the translation of BPMN collaborations (as in [5]). In fact, safeness of a BPMN collaboration only refers to tokens on the sequence edges of the involved processes, while in its Petri Nets translation refers to both message and sequence edges. Indeed, such distinction is not considered in the mapping, because a message is rendered as a (standard) token in a place. Hence, a safe BPMN collaboration where the same message is sent more than once (e.g., via a loop), it is erroneously considered unsafe by relying on the Petri Nets notion (i.e., 1-boundedness), because enqueued messages are rendered as a place with more than one token. Therefore, the notion of safeness defined for Petri Nets cannot be safely applied as it is to collaboration models. Similarly, regarding the soundness property, we are able to consider different notions of soundness according to the requirements we impose on message queues (e.g., ignoring or not pending messages). Again, due to lack of distinction between message and sequence edges, these fine-grained reasoning are precluded using the current translations from BPMN to Petri Nets.

The study of BPMN models via the Petri Nets framework has another limitation concerning the management of the terminate event. In the mapping provided in [5], terminate events are treated as a special type of error events, which however occur mainly on sub-processes, whose translation assumes safeness. This does not allow reasoning on most models including the terminate event, and more in general on all models including unsafe sub-processes. Anyway, given the local nature of Petri Nets transitions, such cancellation patterns are difficult to handle. This is confirmed in [21], stating that modelling a vacuum cleaner, i.e., a construct to remove all the tokens from a given fragment of a net, is possible but results in a spaghetti-like diagram.

The ability of our formal framework to properly classify BPMN models both at process and collaboration level, together with our treatment of the terminate event and sub-processes without any of the restrictions mentioned above, has led us to further novel results, synthesised by the Euler diagrams in Fig. 2(a) and Fig. 2(b). In particular, Fig. 2(a) underlines reasoning that can be done at process level on soundness. Here it emerges how the terminate event can affect model soundness, as using it in place of an
end event may render sound a model that was unsound. For example, let us consider the process in Fig. 3; it is a simple process that first runs in parallel Task A and Task B, then resulting in two possible executions of Task C, and finally completes with an end event. According to the proposed classification the model is unsound, due to the presence of more than one token possibly reaching the end event at different time, that is when the first token arrives there is still a token on the way. Now, let us consider another model, obtained from the one in Fig. 3 by replacing the end event with a terminate event. The resulting model is sound. In fact, from any reachable configuration of the process it is possible to arrive in a (completed) configuration where all marked end events are marked exactly by a single token and all sequence edges are unmarked. This is due to the behaviour of the terminate event that, when reached, removes all tokens in the process. We underline that, although the two models are quite similar, in terms of our classification they result to be significantly different. Also the use of sub-processes can impact on the satisfaction of the soundness property. Fig. 4 shows a simple process model where the unsound process in Fig. 3 is included in the behaviour of a sub-process. Notably, a sub-process is not syntactic sugar that can be removed via a sort of macro expansion. Indeed, according to the BPMN standard, a sub-process completes only when all the internal tokens are consumed, and then just one token is propagated along the including process [1]. Thus, it results that the model in Fig. 4 is sound. Its behaviour would not correspond to that of the process obtained by flattening it, as the resulting model could be unsound. Notice, this reasoning cannot be extended to collaborations. In fact, as we discuss in Sec. 7 when we compose two sound processes the resulting collaboration could be either sound or unsound.

Interesting situations also arise when focussing on the collaboration level, as observed in Fig. 2(b). Worth to notice is the possibility to transform with a small change an unsound collaboration into a sound one. In Fig. 5 and 6 we report a simple example showing the impact of sub-processes on message exchange. Also in this case the two models are rather similar, but according to our classification they result completely different. In particular, the collaboration in Fig. 5 is unsound, but the use of the sub-process mitigates the causes. In fact, at completion time, there will be a pending message, since task C sends two messages and only one will be consumed. However, apart for the token in the end event of ORG A, no other pending sequence token needs to be processed. This means that the collaboration is message-disregarding sound. Differently, Fig. 6 shows that enclosing within a sub-process the part of the model generating multiple tokens it has also in this case a positive effect on the soundness of the model. The collaboration is sound, since only one message is sent and there are no pending messages. Message-disregarding soundness is also satisfied since it is implied by soundness.

Relevance into Practice. To get a clearer idea of the impact of well-structuredness, safeness, soundness and message-disregarding soundness on the real-world modelling practice, we have analyzed the BPMN 2.0 collaboration models available in a well-
known, public, well-populated repository provided by the BPM Academic Initiative
(http://bpmai.org). From the raw dataset, to avoid uncompleted models and low
quality ones, we have selected only those with 100% of connectedness (i.e., all model
elements are connected). This results on 2.740 models suitable for our investigation.
Results of this study are reported in Figure 7, where the models are grouped in terms
of number of contained elements. From the technical point of view, well-structuredness
has been checked using the PromniCAT platform\footnote{https://github.com/tobiashoppe/promnicat}, while safeness and soundness have
been checked using the S\textsuperscript{3} tool\footnote{http://pros.unicam.it/s3/}.

We have found that 86% of models in the repository are well-structured. Anyway,
more interesting is the trend of the number of well-structured models with respect to
their size. It shows that in practice BPMN models starts to become unstructured when
their size grows. This means that structuredness should be regarded as a general guide-
line but one can deviate from it if necessary, especially in modelling complex scenarios.
The balancing between the two classes motivates, on the one hand, our design choice
of considering in our formalisation BPMN models with an arbitrary topology and, on
the other hand, the necessity of studying well-structuredness and the related properties.

Concerning safeness, it results that 2.689 models are safe. The classes that surely
cannot be neglected in our study, as they are suitable to model realistic scenarios, are
those with size 20-29, 30-39 and 40-49 including 156 models, of which only 3 are
unsafe. This makes evident that modelling safe models is part of the practice, and that
imposing well-structuredness is sometimes too restrictive, since there is a huge class of
collaboration models that are safe even if with an unstructured topology.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
Size & Dataset & WS & Non-WS & Safe & Sound & MD-Sound \\
\hline
0 - 9 & 1668 & 1551 (93\%) & 117 (7\%) & 1647 & 1077 & 1133 \\
10 - 19 & 910 & 692 (76\%) & 218 (24\%) & 883 & 462 & 487 \\
20 - 29 & 137 & 95 (69\%) & 42 (31\%) & 134 & 51 & 57 \\
30 - 39 & 13 & 4 (27\%) & 9 (73\%) & 13 & 4 & 4 \\
40 - 49 & 9 & 1 (14\%) & 8 (86\%) & 9 & 3 & 3 \\
50 - 59 & 1 & 0 (0\%) & 1 (100\%) & 1 & 0 & 0 \\
60 - 69 & 0 & 0 & 0 & 0 & 0 & 0 \\
70 - 79 & 2 & 0 (0\%) & 2 (100\%) & 2 & 0 & 0 \\
\hline
0 - 79 & 2740 & 2342 (86\%) & 398 (14\%) & 2689 & 1597 & 1684 \\
\hline
\end{tabular}
\caption{Classification of the models in the BPM Academic Initiative repository.}
\end{table}
Concerning soundness, it results that there are 1,597 sound models. The number of sound models is less than that of safe models, since soundness is more restrictive. However, it results that modelling in a sound way is a common practice, recognizing soundness as one of the most important correctness criteria. Moreover, the data show that there are well-structured models that are not sound. Concerning message disregarding-soundness, it results that the number of models satisfying this property is 87 more than the sound ones. This highlights a set of collaboration models, up to now, not considered.

3 Running Example

In this section we introduce a BPMN collaboration that combines the activities of a Customer and a Travel Agency (Fig. 8). It is used throughout the paper as a running example. This example is intentionally kept simple, as it just aims at illustrating how the language works in practice.

Running Example (1/9). The Travel Agency continuously offers travels to the Customer, until an offer is accepted. If the Customer is interested in one offer, she decides to book the travel and refuses all the other offers that the Travel Agency insistently proposes. As soon as the booking is received by the Travel Agency, it sends back a confirmation message, and asks for the payment of the travel. When this is completed the ticket is sent to the Customer, and the Travel Agency activities immediately end.

The processes of the Customer and of the Travel agency are represented inside two rectangles, called pools. These are used to represent participants or organisations involved in a collaboration, and provide details on internal process specifications and related elements. The flow of process elements in the same or different pools is given by connecting edges. In particular, sequence edges are used to specify the internal flow of the process, thus ordering elements in the same pool, while message edges are dashed connectors used to visualise communication flows between organisations.

Considering the Customer pool, from left to right, we have that as soon as the process starts, due to the presence of a start event (drawn as a circle with an outgoing sequence edge), the Customer checks the travel offer. This is done by executing a receiving task (drawn as a rectangle with rounded corners, an incoming message edge, and an incoming and an outgoing sequence edge). Then, she decides either to book

![BPMN diagram](image-url)

Fig. 8: BPMN collaboration diagram of a Travel Agency scenario.
the travel or to wait for other offers, by means of a cycle composed of two gateways: an XOR-Join (drawn as a diamond marked by \(\times\) with two incoming sequence edges and an outgoing one) that acts as a pass-through, meaning that it is activated each time the gateway is reached; and an XOR-Split (having two outgoing sequence edges and an incoming one), used after a decision to fork the flow into two branches. After the Customer finds the interesting offer, she books the travel, by sending a message to the Travel Agency by executing a sending task (having an outgoing message edge), and waits for the booking confirmation. As soon as she receives the booking confirmation, through an intermediate receiving event (drawn as a circle with an incoming message edge), she pays the travel, receives the ticket from the Agency and her specific works terminate by means of an end event (drawn as a thick circle).

Considering the work of the Travel Agency, as soon as its process starts, it makes travel offers to the Customer, by means of a sending task, until an offer is accepted. Thanks to the behaviour of the AND-Split (drawn as a diamond marked by \(\oplus\) with two outgoing sequence edges and an incoming one, used to enable parallel execution flows) combined with the XOR-Join in a cycle, it continuously make offers. At the same time, it proceeds in order to receive a booking via an intermediate receiving event. Then, it confirms the booking and sends a notification to the Customer. Finally, after receiving the payment, it orders and sends the ticket, thus completing its activities by means of a terminate end event (displayed by a thick circle with a darkened circle inside) which stops and aborts the running process.

4 Formal Framework

This section presents our formalisation of the BPMN semantics. Specifically, we first present the syntax and operational semantics we defined for a relevant subset of BPMN elements. In selecting the elements we followed a pragmatic approach as, even if we deal with a restricted number of elements, we focus on those regularly used to design process models in practice (corresponding to less than 20% of the BPMN vocabulary [22]). Anyway, extending the framework to include further elements is not particularly challenging (even a tricky element as the OR-join can be conveniently added to our formalisation, see [23]).
4.1 Syntax of BPMN Collaborations

To enable the formal treatment of collaborations’ semantics, we defined a BNF syntax of their model structure (Fig. 9). In the proposed grammar, the non-terminal symbols $C$ and $P$ represent **Collaborations Structure** and **Processes Structure**, respectively. The two syntactic categories directly refer to the corresponding notions in BPMN. The terminal symbols, denoted by the sans serif font, are the typical elements of a BPMN model, i.e., pools, events, tasks, sub-processes and gateways.

It is worth noticing that our syntax is too permissive with respect to the BPMN notation, as it allows to write collaborations that cannot be expressed in BPMN. Limiting such expressive power would require to extend the syntax (e.g., by imposing processes having at least one end event), thus complicating the definition of the formal semantics. However, this is not necessary in our work, as we are not proposing an alternative modelling notation, but we are only using a textual representation of BPMN models, which is more manageable for writing operational rules than the graphical notation. Therefore, in our analysis we will only consider terms of the syntax that are derived from BPMN models.

Intuitively, a BPMN collaboration model is rendered in our syntax as a collection of pools and each pool contains a process. More formally, a Collaboration $C$ is a composition, by means of operator $\mid$ of pools of the form $\text{pool}(p, P)$, where $p$ is the name that uniquely identifies the Pool; $P$ is the Process included in the specific pool, respectively.

In the following, $m \in \mathcal{M}$ denotes a *message edge*, enabling message exchanges between pairs of participants in the collaboration, while $M \in 2^\mathcal{M}$. Moreover, $m$ denotes names uniquely identifying a message edge. We also observe $e \in \mathcal{E}$ denotes a *sequence edge*, while $E \in 2^\mathcal{E}$ a set of edges; we require $|E| > 1$ when $E$ is used in joining and splitting gateways. Similarly, we require that an event-based gateway should contain at least two message events, i.e. $h > 1$ in each *eventBased* term. For the convenience of the reader, we refer with $e_i$ the edge incoming in an element and with $e_o$ the edge outgoing from an element. In the edge set $\mathcal{E}$ we also include spurious edges denoting the enabled status of start events and the complete status of end events, named *enabling* and *completing* edges, respectively. In particular, we use edge $e_{enb}$, incoming to a start event, to enable the activation of the process, while $e_{cmp}$ is an edge outgoing from the end events suitable to check the completeness of the process. They are needed to activate sub-processes as well as to check their completion. Moreover, as well as for the message edge we have that $e$ denotes names uniquely identifying a sequence edge.

The correspondence between the syntax used here to represent a *Process Structure* and the graphical notation of BPMN is as follows.

- $\text{start}(e_{enb}, e_o)$ represents a start event that can be activated by means of the enabling edge $e_{enb}$ and it has an outgoing edge $e_o$.
- $\text{end}(e_i, e_{cmp})$ represents an end event with an incoming edge $e_i$ and a completing edge $e_{cmp}$.
- $\text{startRcv}(e_{enb}, m, e_o)$ represents a start message event that can be activated by means of the enabling edge $e_{enb}$ as soon as a message $m$ is received and it has an outgoing edge $e_o$.
- $\text{endSnd}(e_i, m, e_{cmp})$ represents an end message event with an incoming edge $e_i$, a message $m$ to be sent, and a completing edge $e_{cmp}$.
- terminate(e_i) represents a terminate event with an incoming edge e_i.
- eventBased(e_i, (m_1, e_o1), . . . , (m_h, e_oh)) represents an event based gateway with incoming edge e_i and a list of possible (at least two) message edges, with the related outgoing edges that are enabled by message reception.
- andSplit(e_i, E_o) - resp. xorSplit(e_i, E_o) - represents an AND - resp. XOR - split gateway with incoming edge e_i and outgoing edges E_o.
- andJoin(E_i, e_o) - resp. xorJoin(E_i, e_o) - represents an AND - resp. XOR - join gateway with incoming edges E_i and outgoing edge e_o.
- task(e_i, e_o) represents a task with incoming edge e_i and outgoing edge e_o; we can also observe taskRcv(e_i, m, e_o) - resp. taskSnd(e_i, m, e_o) - to consider a task receiving - resp. sending - a message m.
- interRcv(e_i, m, e_o) (resp. interSnd(e_i, m, e_o)) represents an intermediate receiving - resp. sending - event with an incoming edge e_i and an outgoing edge e_o that are able to receive - resp. sending - a message m.
- subProc(e_i, P, e_o) represents a sub-process element with incoming edge e_i and outgoing edge e_o. When activated, the enclosed sub-process P behaves according to the elements it consists of, including nested sub-process elements (used to describe collaborations with a hierarchical structure).
- P_1 | P_2 represents a composition of elements in order to render a process structure in terms of a collection of elements.

Moreover, to simplify the definition of well-structured processes (given later), we include an empty task in our syntax. It permits to connect two gateways with a sequence flow without activities in the middle.

To achieve a compositional definition, each sequence (resp. message) edge of the BPMN model is split in two parts: the part outgoing from the source element and the part incoming into the target element. The two parts are correlated since edge names in the BPMN model are unique. To avoid malformed structure models, we only consider structures in which for each edge labeled by e there exists only one corresponding edge labeled by e, and vice versa.

Here in the following we define some auxiliary functions defined on the collaboration structure and the process structure. Considering BPMN collaborations they may include one or more participants; function participant(C) returns the process structures included in a given collaboration structure. Formally, it is defined as follows.

\[
\text{participant}(C | C_2) = \text{participant}(C_1) \cup \text{participant}(C_1) \\
\text{participant}(\text{pool}(p, P)) = P
\]

Since we also consider process including nested sub-processes to refer to the enabling edges of the start events of the current layer we resort to functions start(P).

\[
\text{start}(P_1 | P_2) = \text{start}(P_1) \cup \text{start}(P_2) \\
\text{start}(\text{start}(e_{cnb}, e_o)) = \{e_{cnb}\} \\
\text{start}(\text{startRcv}(e_{cnb}, m, e_o)) = \{e_{cnb}\} \\
\text{start}(P) = \emptyset \text{ for any element } P \neq \text{start}(e_{cnb}, e_o) \text{ or } P \neq \text{startRcv}(e_{cnb}, m, e_o)
\]
Notably, we assume that each process/sub-processes in the collaboration has only one start event. Function start applied to $C$ will return as many enabling edges as the number of participants involved.

$$start(C_1 | C_2) = start(participant(C_1)) \cup start(participant(C_2))$$

$$start(pool(p, P)) = start(P)$$

We similarly define functions $end(P)$ and $end(C)$ on the structure of processes and collaborations in order to refer to end events in the current layer.

$$end(P_1 | P_2) = end(P_1) \cup end(P_2)$$

$$endSnd(e_i, m, e_{cmp}) \quad end(end(e_i, e_{cmp})) = \{e_{cmp}\}$$

$$end(P) = \emptyset$$ for any element $P \neq end(e_i, e_{cmp})$ or $P \neq endSnd(e_i, m, e_{cmp})$

The function $end(C)$ on the collaboration structure is defined as follow.

$$end(C_1 | C_2) = end(participant(C_1)) \cup end(participant(C_2))$$

$$end(pool(p, P)) = end(P)$$

We also define the function $edges(P)$ to refer the edges in the scope of $P$.

$$edges(P_1 | P_2) = edges(P_1) \cup edges(P_2)$$

$$edges(start(enb, e_o)) = \{enb, e_o\} \quad edges(end(e_i, e_{cmp})) = \{e_i, e_{cmp}\}$$

$$edges(startRcv(e_{enb}, m, e_o)) = \{enb, e_o\} \quad edges(endSnd(e_i, m, e_{cmp})) = \{e_i, e_{cmp}\}$$

$$edges(terminate(e_i)) = \{e_i\}$$

$$edges(eventBased(e_i, (m_1, e_{o1}), \ldots, (m_h, e_{oh}))) = \{e_i, e_{o1}, \ldots, e_{oh}\}$$

$$edges(andSplit(e_i, e_{o1}, \ldots, e_{oh})) = \{e_i, e_{o1}, \ldots, e_{oh}\}$$

$$edges(xorSplit(e_i, e_{o1}, \ldots, e_{oh})) = \{e_i, e_{o1}, \ldots, e_{oh}\}$$

$$edges(andJoin(e_{i1}, \ldots, e_{ih}, e_o)) = \{e_{i1}, \ldots, e_{ih}, e_o\}$$

$$edges(xorJoin(e_{i1}, \ldots, e_{ih}, e_o)) = \{e_{i1}, \ldots, e_{ih}, e_o\}$$

$$edges(task(e_i, e_o)) = \{e_i, e_o\}$$

$$edges(taskRcv(e_i, m, e_o)) = \{e_i, e_o\} \quad edges(taskSnd(e_i, m, e_o)) = \{e_i, e_o\}$$

$$edges(empty(e_i, e_o)) = \{e_i, e_o\} \quad edges(interRcv(e_i, m, e_o)) = \{e_i, e_o\}$$

$$edges(interSnd(e_i, m, e_o)) = \{e_i, e_o\} \quad edges(subProc(e_i, P, e_o)) = \{e_i, e_o\} \cup edges(P)$$

Running Example (2/9). The BPMN model in Fig.8 is expressed in our syntax as the following collaboration structure (at an unspecified step of execution):

$$pool(Customer, P_{C}) \mid pool(TravelAgency, P_{TA})$$
with \( P_C \) expressed as follows (process structure \( P_{TA} \) is defined in a similar way) where for simplicity we identify the edges in progressive order \( e_i \) (with \( i = 0 .. 10 \):

\[
\text{start}(e_0, e_1) \mid \text{xorJoin}(\{e_1, e_3\}, e_2) \mid \text{taskRcv}(e_2, \text{Offer}, e_4) \mid \text{xorSplit}(e_4, \{e_3, e_5\}) \\
\text{taskSnd}(e_5, \text{Travel}, e_6) \mid \text{interRcv}(e_6, \text{Confirmation}, e_7) \mid \text{taskSnd}(e_7, \text{Payment}, e_8) \\
\text{interRcv}(e_8, \text{Ticket}, e_9) \mid \text{end}(e_9, e_{10})
\]

Moreover, considering functions we define on the structure we can observe the following: \( \text{participant}(\text{pool}(\text{Customer}, P_C) | \text{pool}(\text{TravelAgency}, P_{TA})) = \{P_C, P_{TA}\} \), \( \text{start}(P_C) = \{e_0\} \), and \( \text{end}(P_C) = \{e_{10}\} \). The others are defined in a similar way.

4.2 Semantics of BPMN Collaborations

The syntax presented so far permits to describe the mere structure of a collaboration and a process. To describe its semantics we need to enrich it with a notion of execution state, defining the current marking of sequence and message edges. We call \textit{collaboration configuration} and \textit{process configuration} these stateful descriptions.

Formally, a collaboration configuration has the form \( \langle C, \sigma, \delta \rangle \), where: \( C \) is a collaboration structure; \( \sigma \) is the part of the execution state at process level, storing for each sequence edge the current number of tokens marking it (notice it refers to the edges included in all the process of the collaboration), and \( \delta \) is the part of the execution state at collaboration level, storing for each message edge the current number of message tokens marking it. Moreover, a process configuration has the form \( \langle P, \sigma \rangle \), where: \( P \) is a process structure; and \( \sigma \) is the execution state at process level. Specifically, a state \( \sigma : E \rightarrow \mathbb{N} \) is a function mapping edges to numbers of tokens. The state obtained by updating in the state \( \sigma \) the number of tokens of the edge \( e \) to \( n \), written as \( \sigma \cdot \{e \rightarrow n\} \), is defined as follows: if \( \sigma(e') = e \), otherwise it returns \( \sigma(e') \). Moreover, a state \( \delta : M \rightarrow \mathbb{N} \) is a function mapping message edges to numbers of message tokens; so that \( \delta(m) = n \) means that there are \( n \) messages of type \( m \) sent by a participant to another that have not been received yet. Update for \( \delta \) are defined in a way similar to \( \sigma \)’s definitions.

Given the notion of configuration, a collaboration is in the \textit{initial state} when each process it includes is in the \textit{initial state}, meaning that the start event of each process must be enabled, i.e. it has a token in its enabling edge, while all other sequence edges (included the enabling edges for the activation of nested sub-processes), and messages edges must be unmarked.

\textbf{Definition 1 (Initial state of process)}. Let \( P \) be a process, the process configuration \( \langle P, \sigma \rangle \) is the initial one, i.e. predicate \( \text{isInit}(P, \sigma) \) holds, if \( \sigma(\text{start}(P)) = 1 \), and \( \forall \ e \in \text{edges}(P) \setminus \text{start}(P) . \sigma(e) = 0 \).

\textbf{Definition 2 (Initial state of collaboration)}. Let \( C \) be a collaboration, the collaboration configuration \( \langle C, \sigma, \delta \rangle \) is the initial one, i.e. predicate \( \text{isInit}(C, \sigma, \delta) \) holds, if \( \forall \ P \in \text{participant}(C) \) we have that \( \text{isInit}(P, \sigma) \), and \( \forall \ m \in M . \delta(m) = 0 \).
Running Example (3/9). The initial configuration of the collaboration in Fig. 8 is as follows. Given participant(C) = {PC, PT-A}, we have that \( \langle PC, \sigma \rangle, \sigma(e_0) = 1 \), \( \sigma(e_i) = 0 \) \( \forall e_i \) with \( i = 1..10 \), and \( \langle PT-A, \sigma \rangle, \sigma(e_{11}) = 1 \) and \( \sigma(e_j) = 0 \) \( \forall e_j \) with \( j = 12..22 \). We also have that \( \delta(Offer, Confirmation, Ticket, Travel, Payment) = 0 \).

The operational semantics is defined by means of a labelled transition system (LTS) on collaboration configuration and formalises the execution of message marking evolution according to the process evolution. Its definition relies on an auxiliary transition relation on the behaviour of process.

The auxiliary transition relation is a triple \( \langle P, A, \rightarrow \rangle \) where: \( P \), ranged over by \( \langle P, \sigma \rangle \), is a set of process configurations; \( A \), ranged over by \( \alpha \), is a set of labels (of transitions that process configurations can perform); and \( \rightarrow \subseteq P \times A \times P \) is a transition relation. We will write \( \langle P, \sigma \rangle \xrightarrow{\alpha} \langle P, \sigma' \rangle \) to indicate that \( \langle P, \sigma, \alpha, \langle P, \sigma' \rangle \rangle \in \rightarrow \) and say that ‘the process in the configuration \( \langle P, \sigma \rangle \) can do a transition labelled \( \alpha \) and become the process configuration \( \langle P, \sigma' \rangle \) in doing so’. Since process execution only affects the current states, and not the process structure, for the sake of readability we omit the structure from the target configuration of the transition. Thus, a transition \( \langle P, \sigma \rangle \xrightarrow{\alpha} \langle P, \sigma' \rangle \) is written as \( \langle P, \sigma \rangle \xrightarrow{\alpha} \sigma' \). The labels used by this transition relation are generated by the following production rules.

\[
\begin{align*}
\text{(Actions)} \quad \alpha & := \tau \mid !m \mid ?m \\
\text{(Internal actions)} \quad \tau & := \epsilon \mid \text{kill}
\end{align*}
\]

The meaning of labels is as follows. Label \( \tau \) denotes an action internal to the process, while \( !m \) and \( ?m \) denote sending and receiving actions, respectively. The meaning of internal actions is as follows: \( \epsilon \) the movement of a token through the process unless the termination action denoted by \( \text{kill} \).

The transition relation over process configurations formalises the execution of a process; it is defined by the rules at the top of Fig. 10.

Before commenting on the rules, we introduce the auxiliary functions they exploit. Specifically, function \( \text{inc} : S \times E \rightarrow S \) (resp. \( \text{dec} : S \times E \rightarrow S \) ), where \( S \) is the set of states, allows updating a state by incrementing (resp. decrementing) by one the number of tokens marking an edge in the state. Formally, they are defined as follows: \( \text{inc}(\sigma, e) = \sigma \cdot \{ e \mapsto \sigma(e) + 1 \} \) and \( \text{dec}(\sigma, e) = \sigma \cdot \{ e \mapsto \sigma(e) - 1 \} \). These functions extend in a natural ways to sets of edges as follows: \( \text{inc}(\sigma, \emptyset) = \sigma \) and \( \text{inc}(\sigma, \{ e \} \cup E) = \text{inc}(\text{inc}(\sigma, e), E) \); the cases for \( \text{dec} \) are similar. As usual, the update function for \( \delta \) are defined in a way similar to \( \sigma \)'s definitions. We also use the function \( \text{zero} : S \times E \rightarrow S \) that allows updating a state by setting to zero the number of tokens marking an edge in the state. Formally, it is defined as follows: \( \text{zero}(\sigma, e) = \sigma \cdot \{ e \mapsto 0 \} \). Also in this case the function extends in a natural ways to sets of edges as follows: \( \text{zero}(\sigma, \emptyset) = \sigma \) and \( \text{zero}(\sigma, \{ e \} \cup E) = \text{zero}(\text{zero}(\sigma, e), E) \).

To check the completion of a process and sub-process we exploit the boolean predicate \( \text{completed}(P, \sigma) \). It is defined according to the prescriptions of the BPMN standard, which states that “a process instance is completed if and only if [...] there is no token remaining within the process instance; no activity of the process is still active.
For a process instance to become completed, all tokens in that instance must reach an end node and “a sub-process instance completes when there are no more tokens in the Sub-Process and none of its Activities is still active” [11, pp. 426, 431]. Thus, the process/sub-process completion can be formalised as follows.

**Definition 3.** Let $P$ be a process, having the form $\text{end}(e_s, e_{cmp}) \mid P'$ or $\text{endSnd}(e_s, m, e_{cmp}) \mid P'$, the predicate $\text{completed}(P, \sigma)$ is defined as

$$\sigma(e_{cmp}) > 0 \land \sigma(e_s) = 0 \land \text{isZero}(P', \sigma)$$

where $\text{isZero}(\cdot)$ is inductively defined on the structure of its first argument as follows:

- $\text{isZero}(\text{start}(e_{enb}, e_o), \sigma)$ if $\sigma(e_{enb}) = 0$ and $\sigma(e_o) = 0$;
- $\text{isZero}(\text{end}(e_i, e_{cmp}), \sigma)$ if $\sigma(e_i) = 0$;
- $\text{isZero}(\text{startRcv}(e_{enb}, m, e_o))$ if $\sigma(e_{enb}) = 0$ and $\sigma(e_o) = 0$;
- $\text{isZero}(\text{endSnd}(e_i, m, e_{cmp}))$ if $\sigma(e_i) = 0$;
- $\text{isZero}(\text{terminate}(e_i), \sigma)$ if $\sigma(e_i) = 0$;
- $\text{isZero}(\text{eventBased}(e_i, (m_1, e_{o1}), \ldots, (m_k, e_{o_k})), \sigma)$ if $\sigma(e_i) = 0$ and $\forall j \in \{1, \ldots, k\}, \sigma(e_{o_j}) = 0$;
- $\text{isZero}(\text{andSplit}(e_i, E_o), \sigma)$ if $\sigma(e_i) = 0$ and $\forall e \in E_o \cdot \sigma(e) = 0$;
- $\text{isZero}(\text{xorSplit}(e_i, E_o), \sigma)$ if $\sigma(e_i) = 0$ and $\forall e \in E_o \cdot \sigma(e) = 0$;
- $\text{isZero}(\text{andJoin}(E_i, e_o), \sigma)$ if $\forall e \in E_i \cdot \sigma(e) = 0$ and $\sigma(e_o) = 0$;
- $\text{isZero}(\text{xorJoin}(E_i, e_o), \sigma)$ if $\forall e \in E_i \cdot \sigma(e) = 0$ and $\sigma(e_o) = 0$;
- $\text{isZero}(\text{task}(e_i, e_o), \sigma)$ if $\sigma(e_i) = 0$ and $\sigma(e_o) = 0$;
- $\text{isZero}(\text{taskSnd}(e_i, m, e_o), \sigma)$ if $\sigma(e_i) = 0$ and $\sigma(e_o) = 0$;
- $\text{isZero}(\text{empty}(e_i, e_o), \sigma)$ if $\sigma(e_i) = 0$ and $\sigma(e_o) = 0$;
- $\text{isZero}(\text{interRcv}(e_i, m, e_o), \sigma)$ if $\sigma(e_i) = 0$ and $\sigma(e_o) = 0$;
- $\text{isZero}(\text{interSnd}(e_i, m, e_o), \sigma)$ if $\sigma(e_i) = 0$ and $\sigma(e_o) = 0$;
- $\text{isZero}(\text{subProc}(e_i, P, e_o), \sigma)$ if $\sigma(e_i) = 0$ and $\forall e \in \text{edges}(P) \cdot \sigma(e) = 0$;
- $\text{isZero}(P_1 \mid P_2, \sigma)$ if $\text{isZero}(P_1, \sigma)$ and $\text{isZero}(P_2, \sigma)$.

Notably, the completion of a process does not depend on the exchanged messages, and it is defined considering the arbitrary topology of the model, which hence may have one or more end events with possibly more than one token in the completing edges.

Finally, we use the function $\text{marked}(\sigma, E)$ to refer to the set of edges in $E$ with at least one token, which is defined as follows:

$$\text{marked}(\sigma, \emptyset) = \emptyset$$

$$\text{marked}(\sigma, \{e\} \cup E) = \{e\} \cup \text{marked}(\sigma, E) \quad \text{if } \sigma(e) > 0$$

$$\text{marked}(\sigma, \emptyset) = \emptyset \quad \text{otherwise}$$

We now briefly comment on some of the operational rules in Fig. [10]. Rule $P-$Start starts the execution of a process/sub-process when it has been activated (i.e., the enabling edge $e_{enb}$ is marked). The effect of the rule is to increment the number of tokens in the edge outgoing from the start event. Rule $P-$End is enabled when there is at least one token in the incoming edge of the end event, which is then moved to the completing edge. Rule $P-$StartRcv start the execution of a process when it is in its initial state. The effect of the rule is to increment the number of tokens in the edge outgoing from the start event and remove the token from the enabling edge. A label corresponding to
the consumption of a message is observed. Rule $P$-EndSnd is enabled when there is at least a token in the incoming edge of the end event, which is then removed which is then moved to the completing edge. At the same time a label corresponding to the production of a message is observed. Rule $P$-Terminate starts when there is at least one token in the incoming edge of the terminate event, which is then removed. Rule $P$-EventG is activated when there is a token in the incoming edge and there is a message $m_j$ to be consumed, so that the application of the rule moves the token from the incoming edge to the outgoing edge corresponding to the received message. A label corresponding to the consumption of a message is observed. Rule $P$-AndSplit is applied when there is at least one token in the incoming edge of an AND split gateway; as result of its application the rule decrements the number of tokens in the incoming edge and increments that in each outgoing edge. Similarly, rule $P$-XorSplit is applied when a token is available in the incoming edge of a XOR split gateway, the rule decrements the token in the incoming edge and increment the token in one of the outgoing edges, non-deterministically chosen. Rule $P$-AndJoin decrements the tokens in each incoming edge and increments the number of tokens of the outgoing edge, when each incoming edge has at least one token. Rule $P$-XorJoin is activated every time there is a token in one of the incoming edges, which is then moved to the outgoing edge. Rule $P$-Task deals with simple tasks, acting as a pass through. It is activated only when there is a token in the incoming edge, which is then moved to the outgoing edge. Rule $P$-TaskRcv is activated when there is a token in the incoming edge and a label corresponding to the consumption of a message is observed. Similarly, rule $P$-TaskSnd, instead of consuming, send a message before moving the token to the outgoing edge. A label corresponding to the production of a message is observed. Rule $P$-InterRcv (resp. $P$-InterSnd) follows the same behaviour of rule $P$-TaskRcv (resp. $P$-TaskSnd). Rule $P$-Empty simply propagates tokens, it acts as a pass through. Rules $P$-SubProcStart, $P$-SubProcEvolution, $P$-SubProcEnd and $P$-SubProcKill deal with the behaviour of a sub-process element. The former rule is activated only when there is a token in the incoming edge of the sub-process, which is then moved to the enabling edge of the start event in the sub-process body. Then, the sub-process behaves according to the behaviour of the elements it contains according to the rules $P$-SubProcEvolution. When the sub-process completes the rule $P$-SubProcEnd is activated. It removes all the tokens from the sequence edges of the sub-process body 3 and adds a token to the outgoing edge of the sub-process. Rule $P$-SubProcKill deals with a sub-process element observing a killing action in its behaviour due to an occurrence of Terminate end event. This rule is activated only when there is a token in the incoming edge of termination event by the rule $P$-Terminate. Then, the sub-process stop its internal behaviours and passes the control to the upper layer, indeed the rule removes all the tokens in the sub-process and adds a token to the outgoing edge of the sub-process. Rule $P$-Kill deal with the propagation of killing action on the scope of $P$ and rule $P$-Int deal with interleaving in a standard way for process elements. Notice that we do not need symmetric versions of the last two rules, as we identify processes up to commutativity and associativity of process collection.

3 Actually, due to the completion definition, only the completing edges of the end events within the sub-process body need to be set to zero.
Fig. 10: BPMN Semantics.
Now, the labelled transition relation on collaboration configurations formalises the execution of message marking evolution according to process evolution. In the case of collaborations, this is a triple $\langle C, A, \rightarrow \rangle$ where: $C$, ranged over by $\langle C, \sigma, \delta \rangle$, is a set of collaboration configurations; $A$, ranged over by $\alpha$, is a set of labels (of transitions that collaboration configurations can perform as well as the process configuration); and $\rightarrow \subseteq C \times A \times C$ is a transition relation. We will write $\langle C, \sigma, \delta \rangle \xrightarrow{\alpha} \langle C', \sigma', \delta' \rangle$ to indicate that $\langle \langle C, \sigma, \delta \rangle, \alpha, \langle C', \sigma', \delta' \rangle \rangle \in \rightarrow$ and say that ‘the collaboration in the configuration $\langle C, \sigma, \delta \rangle$ can do a transition labelled $\alpha$ and become the collaboration configuration $\langle C', \sigma', \delta' \rangle$ in doing so’. Since collaboration execution only affects the current states, and not the collaboration structure, for the sake of readability we omit the structure from the target configuration of the transition. Thus, a transition $\langle C, \sigma, \delta \rangle \xrightarrow{\alpha} \langle \sigma', \delta' \rangle$ is written as $\langle C, \sigma ' \rangle \xrightarrow{\alpha} \langle \sigma', \delta' \rangle$. We recall $\alpha$ are the following: label $\tau$ denotes an action internal to the process, while $!m$ and $?m$ denote sending and receiving actions, respectively. The rules related to the collaboration are defined at the bottom of Fig. [10]

The first three rules allow a single pool, representing organisation $p$, to evolve according to the evolution of its enclosed process $P$. In particular, if $P$ performs an internal action, rule C-Internal, or a receiving/delivery action, rule C-Receive/C-Deliver, the pool performs the corresponding action at collaboration layer. Notably, rule C-Receive can be applied only if there is at least one message available (premise $\delta(m) > 0$); of course, one token is consumed by this transition. Recall indeed that at process level label $?m$ just indicates the willingness of a process to consume a received message, regardless the actual presence of messages. Moreover, when a process performs a sending action, represented by a transition labelled by $!m$, such message is delivered to the receiving organization by applying rule C-Deliver. The resulting transition has the effect of increasing the number of tokens in the message edge $m$. The C-Int rule permits to interleave the execution of actions performed by pools of the same collaboration, so that if a part of a larger collaboration evolves, the whole collaboration evolves accordingly. Notice that we do not need symmetric versions of rule C-Int, as we identify collaborations up to commutativity and associativity of pools collection.

5 Properties of BPMN Collaborations

We provide here a rigorous characterisation, with respect to the BPMN formalisation presented so far, of the key concepts studied in this work: well-structuredness, safeness and soundness. We characterise these properties both at the level of processes and collaborations.

5.1 Well-Structured BPMN Collaborations

Common process modelling notations, such as BPMN, allow process models to have almost any topology. However, it is often desirable that models abide by some structural rules. In this respect, a well-known property of a process model is that of well-structuredness [13]. Informally, a process is well-structured if, for every element with multiple outgoing edges (a split), there is a corresponding node with multiple incoming
Moreover, to simplify the definition of well-structuredness for processes, we provide functions in(P) and out(P), which determine the incoming and outgoing sequence edges of a process element P as follows:

\[
\begin{align*}
\text{in}(\text{start}(\text{e}_{\text{emb}}, \text{e}_o)) &= \emptyset & \text{out}(\text{start}(\text{e}_{\text{emb}}, \text{e}_o)) &= \{\text{e}_o\} \\
\text{in}(\text{end}(\text{e}_i, \text{e}_{\text{cmp}})) &= \{\text{e}_i\} & \text{out}(\text{end}(\text{e}_i, \text{e}_{\text{cmp}})) &= \emptyset \\
\text{in}(\text{startRcv}(\text{e}_{\text{emb}}, \text{m}, \text{e}_o)) &= \emptyset & \text{out}(\text{startRcv}(\text{e}_{\text{emb}}, \text{m}, \text{e}_o)) &= \{\text{e}_o\} \\
\text{in}(\text{endSnd}(\text{e}_i, \text{m}, \text{e}_{\text{cmp}})) &= \{\text{e}_i\} & \text{out}(\text{endSnd}(\text{e}_i, \text{m}, \text{e}_{\text{cmp}})) &= \emptyset \\
\text{in}(\text{terminate}(\text{e}_i)) &= \{\text{e}_i\} & \text{out}(\text{terminate}(\text{e}_i)) &= \emptyset \\
\text{in}(\text{andSplit}(\text{e}_i, \text{E}_o)) &= \{\text{e}_i\} & \text{out}(\text{andSplit}(\text{e}_i, \text{E}_o)) &= \{\text{E}_o\} \\
\text{in}(\text{xorSplit}(\text{e}_i, \text{E}_o)) &= \{\text{e}_i\} & \text{out}(\text{xorSplit}(\text{e}_i, \text{E}_o)) &= \{\text{E}_o\} \\
\text{in}(\text{andJoin}(\text{E}_i, \text{e}_o)) &= \{\text{E}_i\} & \text{out}(\text{andJoin}(\text{E}_i, \text{e}_o)) &= \{\text{e}_o\} \\
\text{in}(\text{xorJoin}(\text{E}_i, \text{e}_o)) &= \{\text{E}_i\} & \text{out}(\text{xorJoin}(\text{E}_i, \text{e}_o)) &= \{\text{e}_o\} \\
\text{in}(\text{eventBased}(\text{e}_i, (\text{m}_1, \text{e}_{\text{val}}), \ldots, (\text{m}_h, \text{e}_{\text{val}}))) &= \{\text{e}_i\} & \text{out}(\text{eventBased}(\text{e}_i, (\text{m}_1, \text{e}_{\text{val}}), \ldots, (\text{m}_h, \text{e}_{\text{val}}))) &= \{\text{e}_o\} \\
\text{in}(\text{task}(\text{e}_i, \text{e}_o)) &= \{\text{e}_i\} & \text{out}(\text{task}(\text{e}_i, \text{e}_o)) &= \{\text{e}_o\} \\
\text{in}(\text{taskRcv}(\text{e}_i, \text{m}, \text{e}_o)) &= \{\text{e}_i\} & \text{out}(\text{taskRcv}(\text{e}_i, \text{m}, \text{e}_o)) &= \{\text{e}_o\} \\
\text{in}(\text{taskSnd}(\text{e}_i, \text{m}, \text{e}_o)) &= \{\text{e}_i\} & \text{out}(\text{taskSnd}(\text{e}_i, \text{m}, \text{e}_o)) &= \{\text{e}_o\} \\
\text{in}(\text{empty}(\text{e}_i, \text{e}_o)) &= \{\text{e}_i\} & \text{out}(\text{empty}(\text{e}_i, \text{e}_o)) &= \{\text{e}_o\} \\
\text{in}(\text{interRcv}(\text{e}_i, \text{m}, \text{e}_o)) &= \{\text{e}_i\} & \text{out}(\text{interRcv}(\text{e}_i, \text{m}, \text{e}_o)) &= \{\text{e}_o\} \\
\text{in}(\text{interSnd}(\text{e}_i, \text{m}, \text{e}_o)) &= \{\text{e}_i\} & \text{out}(\text{interSnd}(\text{e}_i, \text{m}, \text{e}_o)) &= \{\text{e}_o\} \\
\text{in}(\text{subProc}(\text{e}_i, \text{P}_1, \text{e}_o)) &= \{\text{e}_i\} & \text{out}(\text{subProc}(\text{e}_i, \text{P}_1, \text{e}_o)) &= \{\text{e}_o\} \\
\text{in}(\text{P}_1 | \text{P}_2) &= (\text{in}(\text{P}_1) \cup \text{in}(\text{P}_2)) \setminus (\text{out}(\text{P}_1) \cup \text{out}(\text{P}_2)) & \text{out}(\text{P}_1 | \text{P}_2) &= (\text{out}(\text{P}_1) \cup \text{out}(\text{P}_2)) \setminus (\text{in}(\text{P}_1) \cup \text{in}(\text{P}_2))
\end{align*}
\]

Moreover, to simplify the definition of well-structuredness for processes, we provide the definition of well-structured core by means of the boolean predicate isWSCore(·).

**Definition 4 (Well-structured processes).** A process P is well-structured (WS) if P has one of the following forms:

\[
\begin{align*}
\text{start}(\text{e}_{\text{emb}}, \text{e}_o) & \mid P' & \text{end}(\text{e}_i, \text{e}_{\text{cmp}}) & \quad (1) \\
\text{start}(\text{e}_{\text{emb}}, \text{e}_o) & \mid P' & \text{terminate}(\text{e}_i) & \quad (2) \\
\text{start}(\text{e}_{\text{emb}}, \text{e}_o) & \mid P' & \text{endSnd}(\text{e}_i, \text{m}, \text{e}_{\text{cmp}}) & \quad (3) \\
\text{startRcv}(\text{e}_{\text{emb}}, \text{m}, \text{e}_o) & \mid P' & \text{end}(\text{e}_i, \text{e}_{\text{cmp}}) & \quad (4) \\
\text{startRcv}(\text{e}_{\text{emb}}, \text{m}, \text{e}_o) & \mid P' & \text{terminate}(\text{e}_i) & \quad (5) \\
\text{startRcv}(\text{e}_{\text{emb}}, \text{m}, \text{e}_o) & \mid P' & \text{endSnd}(\text{e}_i, \text{m}, \text{e}_{\text{cmp}}) & \quad (6)
\end{align*}
\]

where \(\text{in}(P') = \{\text{e}_o\}, \text{out}(P') = \{\text{e}_i\}\), and isWSCore(P').

isWSCore(·) is inductively defined on the structure of its first argument as follows:

1. isWSCore(task(\text{e}_i, \text{e}_o));
2. isWSCore(taskRcv(\text{e}_i, \text{m}, \text{e}_o));
3. isWSCore(taskSnd(\text{e}_i, \text{m}, \text{e}_o));
4. isWSCore(\text{empty}(\text{e}_i, \text{e}_o));
5. isWSCore(interRcv(\text{e}_i, \text{m}, \text{e}_o));
6. isWSCore(interSnd(\text{e}_i, \text{m}, \text{e}_o));
7. let $\text{isWSCore}(P_1), \ldots, \text{isWSCore}(P_n)$, then $\text{isWSCore}\left(\text{andSplit}(e_i, E_o) \mid P_1 \mid \ldots \mid P_n \mid \text{andJoin}(E_i, e_o)\right)$ if $\text{in}(P_j) = \{e_j\}, e_j \in E_o, \text{out}(P_j) = \{e'_j\}, e'_j \in E_i$ with $j \in \{1, \ldots, n\}$
8. let $\text{isWSCore}(P_1), \ldots, \text{isWSCore}(P_n)$, then $\text{isWSCore}\left(\text{xorSplit}(e_i, E_o) \mid P_1 \mid \ldots \mid P_n \mid \text{xorJoin}(E_i, e_o)\right)$ if $\text{in}(P_j) = \{e_j\}, e_j \in E_o, \text{out}(P_j) = \{e'_j\}, e'_j \in E_i$ with $j \in \{1, \ldots, n\}$
9. let $\text{isWSCore}(P_1), \ldots, \text{isWSCore}(P_n)$, then $\text{isWSCore}\left(\text{eventBased}(e_i, (m_1, e_{o1}), \ldots, (m_h, e_{oh})) \mid P_1 \mid \ldots \mid P_n \mid \text{xorJoin}(E_i, e_o)\right)$ if $\text{in}(P_j) = \{e_j\}, e_j \in \{e_{o1}, \ldots, e_{oh}\}, \text{out}(P_j) = \{e'_j\}, e'_j \in E_i$ with $j \in \{1, \ldots, n\}$
10. let $\text{isWSCore}(P_1)$ and $\text{isWSCore}(P_2)$, then $\text{isWSCore}\left(\text{xorJoin}(\{e_2, e_3\}, e_1) \mid P_1 \mid P_2 \mid \text{xorSplit}(e_4, \{e_5, e_6\})\right)$ if $\text{in}(P_1) = \{e_1\}, \text{out}(P_1) = \{e_2\}, \text{out}(P_2) = \{e_3\}$
11. let $\text{isWSCore}(P'_1)$, then $\text{isWSCore}\left(\text{subProc}(e_i, \text{start}(e_{emb}, e_o) \mid P'_1 \mid \text{terminate}(e_i), e_o)\right)$ or $\text{isWSCore}\left(\text{subProc}(e_i, \text{start}(e_{emb}, e_o) \mid P'_1 \mid \text{end}(e_i, e_{cmp}), e_o)\right)$ or $\text{isWSCore}\left(\text{subProc}(e_i, \text{startRcv}(e_{emb}, m, e_o) \mid P'_1 \mid \text{endSnd}(e_i, m, e_{cmp}), e_o)\right)$ or $\text{isWSCore}\left(\text{subProc}(e_i, \text{startRcv}(e_{emb}, m, e_o) \mid P'_1 \mid \text{terminate}(e_i), e_o)\right)$
12. let $\text{isWSCore}(P_1)$ and $\text{isWSCore}(P_2)$, then $\text{isWSCore}(P_1 \mid P_2)$ if $\text{out}(P_1) = \text{in}(P_2)$

Well-structuredness can be also simply extended to collaborations, by requiring each process involved in a collaboration to be well-structured.

**Definition 5 (Well-structured collaborations).** Let $C$ be a collaboration, $\text{isWS}(C)$ is inductively defined as follows:

- $\text{isWS}(\text{pool}(P, P))$ if $P$ is well-structured;
- $\text{isWS}(C_1 \mid C_2)$ if $\text{isWS}(C_1)$ and $\text{isWS}(C_2)$.

**Running Example (4/9).** Considering the proposed running example and according to the above definitions, process $P_C$ is well-structured, while process $P_{T\cdot A}$ is not well-structured, due to the presence of the unstructured loop formed by the XOR join and an AND split. Thus, the overall collaboration is not well-structured. □

### 5.2 Safe BPMN Collaborations

Another important condition usually required is *safeness*, i.e. the occurrence of no more than one token along the same sequence edge of a process instance. Safeness is an important criterion of correctness, since an unsafe model could lead to errors in the execution, as shown in the following examples.

**Example 1.** Fig. 11(a) shows an example of unsafe BPMN collaboration, which combines the activities of $\text{ORG A}$ and $\text{ORG B}$. After the process in $\text{ORG A}$ starts, thanks to the behaviour of the AND split gateway, tasks $A$ and $B$ are executed. After task $A$
is completed, a token arrives at the XOR join gateway, while the execution of task $B$ triggers the execution of the ORG $B$'s process, and task $G$ is executed. The completion of this task produces a token on the incoming edge of the XOR split; the token can now follow different paths. Let us consider the case in which task $H$ is executed. Thus, task $C$ is executed and another token arrives at the XOR join gateway of the ORG $A$’s process. Hence, task $D$ will be executed twice. In this way, there will be two tokens in the upper path of the AND join gateway and no token in the lower path. Since the AND join cannot synchronise, the process is interrupted without reaching its end. □

Example 2. Fig. 11(b) shows an example of unsafe process. As soon as the process starts, thanks to the behaviour of the AND split gateway, tasks Task 1 and Task 2 are executed. After the completion of Task 2, the produced token can follow different paths. Thus, it can be the case that Task 3 is executed twice. Thus, there will be two tokens in the upper path of the AND join and no token in its lower path. This results in an unsafe condition blocking the finalisation of the process behaviour. □

In the examples above the unsafeness of the models may cause a deadlock, since the AND join can not allow the execution to proceed while the process has not yet reached its end.

We now provide a formal definition of safeness, which is based on the auxiliary function $maxMarking(\cdot)$ that, given a configuration $\langle P, \sigma \rangle$, determines the maximum number of tokens marking the sequence edges of elements in $P$ according to the state $\sigma$ (this function relies on the standard function $\max(\cdot)$ returning the maximum in a list of natural numbers).
maxMarking(start(e_{enb}, e_o), σ) = σ(e_o)
maxMarking(end(e_i, e_{enm}), σ) = σ(e_i)
maxMarking(startRcv(e_{enb}, m, e_o), σ) = σ(e_o)
maxMarking(endSnd(e_i, m, e_{enm}), σ) = σ(e_i)
maxMarking(terminate(e_i), σ) = σ(e_i)
maxMarking(andSplit(e_i, E_o), σ) = max(σ(e_i), σ(E_o))
maxMarking(xorSplit(e_i, E_o), σ) = max(σ(e_i), σ(E_o))
maxMarking(andJoin(E_i, e_o), σ) = max(σ(E_i), σ(e_o))
maxMarking(xorJoin(E_i, e_o), σ) = max(σ(E_i), σ(e_o))
maxMarking(task(e_i, e_o), σ) = max(σ(e_i), σ(e_o))
maxMarking(taskRcv(e_i, m, e_o), σ) = max(σ(e_i), σ(e_o))
maxMarking(taskSnd(e_i, m, e_o), σ) = max(σ(e_i), σ(e_o))
maxMarking(empty(e_i, e_o), σ) = max(σ(e_i), σ(e_o))
maxMarking(interRcv(e_i, m, e_o), σ) = max(σ(e_i), σ(e_o))
maxMarking(interSnd(e_i, m, e_o), σ) = max(σ(e_i), σ(e_o))
maxMarking(subProc(P_1, P_2, e), σ) = max(maxMarking(P_1, σ), maxMarking(P_2, σ))

We also need the following definitions determining the safeness of a process in a given state.

**Definition 6 (Current state safe process).** A process configuration \( \langle P, σ \rangle \) is current state safe (cs-safe) if and only if \( \text{maxMarking}(P, σ) \leq 1 \).

We can finally conclude with the definition of safe processes and collaborations which requires that cs-safeness is preserved along the computations. Now, a process is defined to be safe if it is preserved that the maximum marking does not exceed one along the process execution. We use \( \to^* \) to denote the reflexive and transitive closure of \( \to \).

**Definition 7 (Safe processes).** A process \( P \) is safe if and only if, given \( σ \) such that \( \text{isInit}(P, σ) \), for all \( σ' \) such that \( \langle P, σ \rangle \to^* σ' \) we have that \( \langle P, σ' \rangle \) is cs-safe.

**Definition 8 (Safe collaborations).** A collaboration \( C \) is unsafe if and only if, given \( σ \) and \( δ \) such that \( \text{isInit}(C, σ, δ) \), there exists a collaboration configuration \( \langle C, σ', δ' \rangle \) such that \( \langle C, σ, δ \rangle \to^* σ', δ' \rangle \) with \( \text{pool}(p, P) \) in \( C \) and \( P \) not safe. A collaboration \( C \) is safe if and only if \( C \) is not unsafe.

**Running Example (5/9).** Let us consider again our running example depicted in Fig. 8. Process \( P_C \) is safe since there is not any process fragment capable of producing more than one token. Process \( P_{TA} \) instead is not safe. In fact, if task Make Travel Offer is executed more than once, we would have that the AND split gateway will produce more than one token in the sequence flow connected to the Booking Received event. Thus, also the resulting collaboration is not safe. □

### 5.3 Sound BPMN Collaborations

We provide here the notion of soundness for BPMN collaborations, which is usually proposed as a correctness criterion for different business process formalizations.
Generally, “a business process model is sound if it can successfully terminate without leaving over active elements and all the model elements can be activated in one of the execution traces” [24]. Here we refer to the classical soundness definition for processes [25] that informally requires the satisfaction of three sub-properties: (i) Option to complete: a process, once started, can always complete; (ii) Proper completion: when a process completes, there exists no related activity of this process which is still running or enabled; (iii) No dead activities: a process model does not contain any dead activity, i.e., for each activity there exists at least one completed trace producible on that model and containing this activity.

Considering the BPMN notion of completeness (see definitions [5]), we have that soundness is reduced to the satisfaction of one property, i.e. option to complete. In fact, the proper completion property is included in the definition of successful completion, since it requires that “there exists no related activity of this process which is still running or enabled”. Moreover, the no dead activities property is equivalent to requiring the complete execution of a process. In fact, the only way to have dead activities is that the incoming sequence flow of that activity is never reached by a token. This can happen either when there is a deadlock upstream the considered activity or when there are some conditions on gateways. The first case is subsumed in the notion of completeness, while the second case is not caught by our semantics. In fact, the presence of gateway conditions does not cause incorrect behaviours, but at most may produce dead code representing situations that never occur.

Here we refer to the soundness as the need that from any reachable configuration it is possible to arrive in a (completed) configuration where all marked end events are marked exactly by a single token and all sequence edges are unmarked. This refers to the current state sound process we following define.

**Definition 9 (Current state sound process).** A process configuration \( \langle P, \sigma \rangle \) is current state sound (cs-sound) if and only if one of the following hold:

- \( \forall e_{cmp} \in \text{marked}(\sigma, \text{end}(P)) \cdot \sigma(e_{cmp}) = 1 \text{ and } \text{isZero}(P, \sigma); \)
- \( \forall e \in \text{edges}(P) \cdot \sigma(e) = 0. \)

**Definition 10 (Sound process).** A process \( P \) is sound if and only if, given \( \sigma \) such that \( \text{isInit}(P, \sigma) \), for all \( \sigma' \) such that \( \langle P, \sigma' \rangle \rightarrow^* \sigma' \) we have that there exists \( \sigma'' \) such that \( \langle P, \sigma'' \rangle \rightarrow^* \sigma'' \), and \( \langle P, \sigma'' \rangle \) is cs-sound.

**Definition 11 (Sound collaboration).** A collaboration \( C \) is sound if and only if, given \( \sigma \) and \( \delta \) such that \( \text{isInit}(C, \sigma, \delta) \), for all \( \sigma' \) and \( \delta' \) such that \( \langle C, \sigma, \delta \rangle \rightarrow^* \langle \sigma', \delta' \rangle \) we have that there exist \( \sigma'' \) and \( \delta'' \) such that \( \langle C, \sigma', \delta' \rangle \rightarrow^* \langle \sigma'', \delta'' \rangle \), and \( \forall P \in C \) we have that \( \langle P, \sigma'' \rangle \) is cs-sound and \( \forall m \in M. \delta''(m) = 0. \)

Thanks to the expressibility of our formalisation to distinguish sequence tokens from message tokens we provide a novel property, named message-disregarding soundness, that relaxes the usual soundness notion by considering sound also those collaborations in which asynchronously sent messages are not handled by the receiver.
Definition 12 (Message-Disregarding sound collaboration). A collaboration \( C \) is sound if and only if, given \( \sigma \) and \( \delta \) such that isInit\((C, \sigma, \delta)\), for all \( \sigma' \) and \( \delta' \) such that \( \langle C, \sigma, \delta \rangle \rightarrow^* \langle \sigma', \delta' \rangle \) we have that there exist \( \sigma'' \) and \( \delta'' \) such that \( \langle C, \sigma', \delta' \rangle \rightarrow^* \langle \sigma'', \delta'' \rangle \), and \( \forall P \in C \) we have that \( \langle P, \sigma'' \rangle \) is cs-sound.

Running Example (6/9). Let us consider again our running example. It is easily to see that process \( P_C \) is sound, since it is always possible to reach the end event and when reached there is no token marking the sequence flows. Also process \( P_{TA} \) is sound, since when a token reaches the terminate event, all the other tokens are removed from the edges by means of the killing effect. However, the resulting collaboration is not sound. In fact, when a travel offer is accepted by the customer, we would have that the AND-Split gateway will produce two tokens, one of which re-activates the task Make Travel Offer. Thus, even if the process completes, the message lists are not empty. However, the collaboration satisfied the Message-Disregarding sound property we define.

6 Relationships among Properties

In this section we study the relationships among the considered properties. In particular we investigate the relationship between (i) well-structuredness and safeness, (ii) well-structuredness and soundness, and (iii) safeness and soundness.

6.1 Well-structuredness vs. Safeness in BPMN

In this section we present some of the main results of this work concerning the correlation between well-structuredness and safeness, both at process and collaboration level. Specifically, we demonstrate that all well-structured models are safe, and that the vice versa does not hold (proofs are reported in Appendix 10).

The proof that well-structured processes are indeed safe (Theorem 1) is based on some auxiliary notions. To this aim, first we show that a process in the initial state is cs-safe (Lemma 1). Then, we show that cs-safeness is preserved by the evolution of well-structured core process elements (Lemma 2) and processes (Lemma 3). These latter two lemmas rely on the notion of reachable processes. In fact, the syntax in Fig. 9 is too liberal, as it allows terms that cannot be obtained (by means of transitions) from a process in its initial state.

Definition 13 (Reachable processes). A process configurations \( \langle P, \sigma \rangle \) is reachable if there exists \( \langle P, \sigma' \rangle \) configurations such that isInit\((P, \sigma')\) and \( \langle P, \sigma' \rangle \rightarrow^* \langle P, \sigma'' \rangle \).

Lemma 1. Let \( P \) be a process, if isInit\((P, \sigma)\) then \( \langle P, \sigma \rangle \) is cs-safe.

Proof (sketch). Trivially, from definition of isInit\((P, \sigma)\). \( \square \)

Lemma 2. Let \( isWSCore(P) \), and let \( \langle P, \sigma \rangle \) be reachable and cs-safe process configuration, if \( \langle P, \sigma' \rangle \rightarrow^* \langle P, \sigma'' \rangle \) then \( \langle P, \sigma'' \rangle \) is cs-safe.
Proof (sketch). We proceed by induction on the structure of well-structured core process elements.

Lemma 3. Let $P$ be WS, and let $\langle P, \sigma \rangle$ be a process configuration reachable and cs-safe, if $\langle P, \sigma \rangle \xrightarrow{\alpha} \sigma'$ then $\langle P, \sigma' \rangle$ is cs-safe.

Proof (sketch). We proceed by case analysis on the structure of $P$, which is a WS process (see Definition 4).

Theorem 1. Let $P$ be a process, if $P$ is well-structured then $P$ is safe.

Proof (sketch). We show that if $\langle P, \sigma \rangle \rightarrow^n \sigma'$ then $\langle P, \sigma' \rangle$ is cs-safe, by induction on the length $n$ of the sequence of transitions from $\langle P, \sigma \rangle$ to $\langle P, \sigma' \rangle$.

The reverse implication of Theorem 1 is not true. In fact there are safe processes that are not well-structured. The collaboration diagram represented in Fig. 12 is an example. The involved processes are trivially safe, since there are not fragments capable of generating multiple tokens; however they are not well-structured.

We now extend the previous results to collaborations.

Theorem 2. Let $C$ be a collaboration, if isWS($C$) then $C$ is safe.

Proof (sketch). We proceed by contradiction.

6.2 Well-structuredness vs. Soundness in BPMN

In this section we present the relationship between well-structuredness and soundness, both at process and collaboration level. Specifically, we prove that a well-structured process is always sound (Theorem 3), but there are sound processes that are not well-structured. To this aim, first we show that a reachable well-structured core process element can always complete its execution (Lemma 4). This latter Lemma is based on the auxiliary definition of the final state of core elements in a process, given for all elements with the exception of start and end events.

Definition 14 (Final state of core elements in $P$). Let $P$ be a process, then isCompleteEl($P, \sigma$) is inductively defined on the structure of process $P$ as follows:
isCompleteEl(task(e₁, e₀), σ) if σ(e₁) = 0 and σ(e₀) = 1
isCompleteEl(taskRcv(e₁, m, e₀), σ) if σ(e₁) = 0 and σ(e₀) = 1
isCompleteEl(taskSnd(e₁, m, e₀), σ) if σ(e₁) = 0 and σ(e₀) = 1
isCompleteEl(empty(e₁, e₀), σ) if σ(e₁) = 0 and σ(e₀) = 1
isCompleteEl(interRcv(e₁, m, e₀), σ) if σ(e₁) = 0 and σ(e₀) = 1
isCompleteEl(interSnd(e₁, m, e₀), σ) if σ(e₁) = 0 and σ(e₀) = 1
isCompleteEl(andSplit(e₁, E₀, σ) if σ(e₁) = 0 and ∀e ∈ E₀ . σ(e) = 1
isCompleteEl(xorSplit(e₁, E₀, σ) if σ(e₁) = 0 and ∃e ∈ E₀ . σ(e) = 1
   and ∀eₖ ∈ E₀ \ e . σ(e) = 0
isCompleteEl(andJoin(E₁, e₀), σ) if ∀e ∈ E₁ . σ(e) = 0 and σ(e₀) = 1
isCompleteEl(xorJoin(E₁, e₀), σ) if ∀e ∈ E₁ . σ(e) = 0 and σ(e₀) = 1
isCompleteEl(eventBased(e₁, (m₁, e₀₁), . . ., (mₖ, e₀ₖ)), σ) if σ(e₁) = 0
   and ∃e ∈ {e₀₁, . . ., e₀ₖ} . σ(e) = 1 and ∀eₖ ∈ {e₀₁, . . ., e₀ₖ \ e}. σ(e) = 0
isCompleteEl(subProc(e₁, P, e₀)) if σ(e₁) = 0, σ(e₀) = 1 and ∀e ∈ edges(P), σ(e) = 0
isCompleteEl(P₁ | P₂, σ) if ∀e ∈ out(P₁ | P₂) : isCompleteEl(getEl(e, P₁ | P₂))
   and ∀e ∈ (edges(P₁ | P₂), out(P₁ | P₂)) : σ(e) = 0

where getEl(e, P) returns the element in P with incoming edge e:

- getEl(e, interRcv(e₁, m, e₀)) = \begin{cases} interRcv(e₁, m, e₀) & \text{if } e = e₀ \\ \epsilon & \text{otherwise} \end{cases}

- getEl(e, interSnd(e₁, m, e₀)) = \begin{cases} interSnd(e₁, m, e₀) & \text{if } e = e₀ \\ \epsilon & \text{otherwise} \end{cases}

- getEl(e, task(e₁, e₀)) = \begin{cases} task(e₁, e₀) & \text{if } e = e₀ \\ \epsilon & \text{otherwise} \end{cases}

- getEl(e, taskRcv(e₁, m, e₀)) = \begin{cases} taskRcv(e₁, m, e₀) & \text{if } e = e₀ \\ \epsilon & \text{otherwise} \end{cases}

- getEl(e, taskSnd(e₁, m, e₀)) = \begin{cases} taskSnd(e₁, m, e₀) & \text{if } e = e₀ \\ \epsilon & \text{otherwise} \end{cases}

- getEl(e, empty(e₁, e₀)) = \begin{cases} empty(e₁, e₀) & \text{if } e = e₀ \\ \epsilon & \text{otherwise} \end{cases}

- getEl(e, andSplit(e₁, E₀)) = \begin{cases} andSplit(e₁, E₀) & \text{if } e ∈ E₀ \\ \epsilon & \text{otherwise} \end{cases}

- getEl(e, andJoin(E₁, e₀)) = \begin{cases} andJoin(E₁, e₀) & \text{if } e = e₀ \\ \epsilon & \text{otherwise} \end{cases}

- getEl(e, xorSplit(e₁, E₀)) = \begin{cases} xorSplit(e₁, E₀) & \text{if } e ∈ E₀ \\ \epsilon & \text{otherwise} \end{cases}

- getEl(e, xorJoin(E₁, e₀)) = \begin{cases} xorJoin(E₁, e₀) & \text{if } e = e₀ \\ \epsilon & \text{otherwise} \end{cases}
Lemma 4. Let \( \langle P, \sigma \rangle \) be a reachable process configuration and isWSCore\( (P) \), then there exists \( \sigma' \) such that
\[
\langle P, \sigma \rangle \xrightarrow{\sigma'} \langle P, \sigma' \rangle
\]
and isCompleteEl\( (P, \sigma') \).

Proof (sketch). We proceed by induction on the structure of well-structured core process. \( \square \)

Theorem 3. Let \( P \) be a WS process, then \( P \) is sound.

Proof (sketch). We proceed by case analysis. \( \square \)

The reverse implication of Theorem 3 is not true. In fact there are sound processes that are not well-structured; see for example the process represented in Fig. [13]. This process is surely unstructured, and it is also trivially sound, since it is always possible to reach an end event without leaving tokens marking the sequence flows.

![Fig. 13: An example of sound process not Well-Structured.](image)

However, Theorem 3 does not extend to the collaboration level. In fact, when we put well-structured processes together in a collaboration, this could be either sound or unsound. This is also valid for message-disregarding soundness.

Theorem 4. Let \( C \) be a collaboration, \( C \) is WS does not imply \( C \) is sound.

Proof (sketch). We proceed by contradiction. \( \square \)

Theorem 5. Let \( C \) be a collaboration, \( C \) is WS does not imply \( C \) is message-disregarding sound.

Proof (sketch). We proceed by contradiction. \( \square \)
6.3 Safeness vs. Soundness in BPMN

In this section we present the relationship between safeness and (message-disregarding) soundness, both at process and collaboration level. Specifically we demonstrate that there are unsafe models that are sound. This is a peculiarity of BPMN due to its capability to support the terminate end event and (unsafe) sub-processes.

Let us first reason at process level and consider some examples.

Example 3. Fig. 14 shows an example of unsafe process, since the AND split gateway produces two tokens that are then merged by the XOR join gateway producing two tokens on the outgoing edge of the XOR join. However, after Task C is executed and one token enables the terminate end event, the kill label is produced and the second token in the sequence flow is removed (rule $P$-Terminate), rendering the process sound.

![Fig. 14: An example of unsafe but sound process.](image)

Let us consider now the collaboration level. We have that unsafe collaborations could either sound or unsound, as proved by the following Theorem.

Theorem 6. Let $C$ be a collaboration, $C$ is unsafe does not imply $C$ is unsound.

Proof (sketch). We proceed by contradiction.

Running Example (9/9). As we will see in Section 7, the collaboration in our running example is composed by two sound processes. In particular, the process of the Travel Agency is unsafe but sound, since the terminate event permits a successfully termination of the process. However, the collaboration is unsound but message-disregarding sound, since there could be messages in the message lists.

Example 4. Let us consider the example in Fig. 15. The process in ORG A is unsafe but it is sound, since the terminate event permits a correct completion of the process. However, if the XOR split gateway of ORG B produces a token on the bottom sequence flow and Task E is executed, Task B will never received the message from Task D. Thus, even if each process has a token that reaches the terminate event and all the other tokens in the process are removed by the killing action, the message lists are not empty. Indeed, the collaboration is message-disregarding sound.

27
7 Properties Compositionality

In this section we study safeness and soundness compositionality, i.e. how the behaviour of processes affects that of the entire resulting collaboration. In particular, we show the interrelationship between the studied properties at collaboration and at process level. At process level we also consider the compositionality of sub-processes, investigating how sub-processes behaviour impacts on the safeness and soundness of process including them.

7.1 On Safeness Compositionality

We show here that safeness is compositional, that is the composition of safe processes always results in a safe collaboration.

**Theorem 7.** Let $C$ be a collaboration, if all processes in $C$ are safe then $C$ is safe.

**Proof (sketch).** We proceed by contradiction (see Appendix 10).

We also show that the unsafeness of a collaboration cannot be in general determined by information about the unsafeness of the processes that compose it. Indeed, putting together an unsafe process with a safe or unsafe one, the obtained collaboration could be either safe or unsafe. Let us consider now some cases.

**Running Example (7/9).** In our example, the collaboration is composed by a safe process and an unsafe one. In fact, focussing on the process of the Travel Agency, we can immediately see that it is not safe: the loop given by a XOR join and an AND split produces multiple tokens on one of the outgoing edges of the AND split. Now, if we consider this process together with the safe process of Customer, the resulting collaboration is not safe. Indeed, the XOR split gateway, which checks if the offer is interesting, forwards only one token on one of the two paths. As soon as a received offer is considered interesting, the Customer process proceeds and completes. Thus, due to the lack of safeness, the travel agency will continue to make offers to the customer, but no more offer messages arriving from the travel agency will be considered by the customer.

**Example 5.** Another example refers to the case in which a collaboration composed by a safe process and an unsafe one results in a safe collaboration, as shown in Fig. 16. If we
focus only on the process in *ORG B* we can immediately notice that it is not safe: again the loop given by a XOR join and an AND split produces multiple tokens on the same edge. However, if we consider this process together with the safe process of *ORG A*, the resulting collaboration is safe. In fact, task D receives only one message, producing a token that is successively split by the AND gateway. No more message arrives from the send task, so, although there is a deadlock, we have no problem of safeness.

**Example 6.** In Fig. 17 we have two unsafe processes, since each of them contains a loop capable of generating an unbounded number of tokens. However, if we consider the collaboration obtained by the combination of these processes, it turns out to be safe. Indeed, as in the previous example, tasks C and B are executed only once, as they receive only one message. Thus, the two loops are blocked and cannot effectively generate multiple tokens.
Example 7. Also the collaboration in Fig. 18 is composed by two unsafe processes: process in ORG A contains an AND split followed by a XOR join that produces two tokens on the outgoing edge of the XOR gateway; process in ORG B contains the same loop as in the previous examples. In this case the collaboration composed by these two processes is unsafe. Indeed, the XOR join in ORG A will effectively produce two tokens since the sending of task B is not blocking.

Let us now consider processes including sub-processes. We show that the composition of unsafe sub-processes always results in unsafe processes, but the vice versa does not hold. There are also unsafe processes including safe sub-process when the unsafeness does not depend from the behaviour of the sub-process.

Theorem 8. Let \( P \) be a process including a sub-process \( \text{subProc}(e_i, P_1, e_o) \), if \( P_1 \) is unsafe then \( P \) is unsafe.

Proof (sketch). We proceed by contradiction (see Appendix 10).

7.2 On Soundness Compositionality

As well as for the safeness property, we show now that it is not feasible to detect the soundness of a collaboration by relying only on information about soundness of processes that compose it. However, the unsoundness of processes implies the unsoundness of the resulting collaboration.

Theorem 9. Let \( C \) be a collaboration, if all processes in \( C \) are unsound then \( C \) is unsound.

Proof (sketch). We proceed by contradiction (see Appendix 10).

On the other hand, when we put together sound processes, the obtained collaboration could be either sound or unsound, since we have also to consider messages. It can happen that either a process waits for a message that will never be received or it receive more than the number of messages it is able to process. Let us consider some examples.

Running Example (8/9). In our running example, the collaboration is composed by two sound processes. In fact, the Customer process is well-structured, thus sound. Focusing on the process of the Travel Agency, it is also sound since when it completes the terminate end event aborts all the running activities and removes all the tokens still present (more details will follow in Section 2). However, the resulting collaboration is not sound, since the message lists could not be empty.

Example 8. In Fig. 19 we have a collaboration resulting from the composition of two sound processes. If we focus only on the processes in ORG A and ORG B we can immediately note that they are sound. However, the resulting collaboration is not sound. In fact, for instance, if Task A is executed, Task C in ORG B will never receive the message and the AND-Join gateway cannot be activated, thus the process of ORG B cannot complete its execution.
Example 9. Also the collaboration in Fig. 20 is trivially composed by two sound processes. However, in this case also the resulting collaboration is sound. In fact, Task E will always receive the message by Task B and the processes of ORG A and ORG B can correctly complete.

Let’s now consider soundness in a multi-layer structure. We show that the composition of unsound sub-processes does not result in un-sound processes. There are also sound processes including unsound sub-process. In fact, when we put unsound sub-process together in a process, this could be either sound or unsound.

Theorem 10. Let $P$ be a process including a sub-process $\text{subProc}(e_i, P_1, e_o)$, if $P_1$ is unsound does not imply $P$ is unsound.

Proof (sketch). We proceed by contradiction (see Appendix 10). □

8 Related Works

Regarding safeness, the closest approach to ours is the contribution given by Dijkman et al. [5] discussing about safeness in Petri Nets resulting from the translation of BPMN models. In such work, safeness of BPMN terms means that no activity will ever be
enabled or running more than once concurrently. This definition is given using the natural language, while in our work we give a precise characterisation of safeness for both BPMN processes and collaborations. Other approaches introducing mapping from BPMN to formal languages, such as YAWL [26] and COWS [27], do not consider safeness, even if it is recognised as an important characteristic to support verification [28].

We also provide a rigorous correlation between well-structuredness and safeness. To do that we have been inspired by the definition of well-structuredness given in [13]. Differently from [13], we also formalise the definition of well-structuredness in presence of sub-processes and we extend it at collaboration layer. Other contributions are available in the literature with reference to well-structuredness and its correlation with safeness. However, also those that are formally defined do not specifically focus on BPMN. Gruhn and Laue [29] discuss patterns that are related to structured and unstructured business process models, providing examples of sound and safe models that are not well-structured. However, they work on EPC and proceed only by examples, without giving rigorous proofs. Van der Aalst et al. [30] state that a workflow net is well-structured if the split/join constructions are properly nested. Also El-Saber and Boronat [32] propose a formal definition of well-structured processes, in terms of a rewriting logic, but they do not extend this definition at collaboration level.

The lack of structuredness can be correlated to other structural properties that may lead to problems in the execution. One of these is the lack of synchronisation, whose absence, together with that of deadlock, contributes to make a workflow process sound [14]. Although recognised as one of the most important correctness criteria, there are many different notions of soundness in the literature, referring to different process languages and even for the same process language, e.g. for EPC soundness definition is given by Mendling in [33], and for Workflow Nets van der Aalst [15] provides two equivalent soundness definitions. However, these definitions cannot be used directly for BPMN because of its peculiarities. In fact, although the BPMN process flow resembles to some extent the behaviour of Petri Nets, it is not the same. BPMN 2.0 provides a comprehensive set of elements that go far beyond the definition of mere place/transition flows and enable modelling at an higher level of abstraction. For example, using Petri Nets it is difficult to describe certain operations typical of the business process domain, such as the termination event, and often it is required to rely on some limiting assumptions (e.g., safeness). The relationships between the studied properties differ from one process language to another. For instance, well-structured workflow nets are guaranteed to be sound if they are live [16], while van Hee et al. prove that a well-structured workflow process is always sound [18]. In our work, we prove that a well-structured process is also sound, but a well-structured collaboration is not always sound. Moreover, while soundness implies safeness in the area of Petri Nets [15], this is not true in BPMN.

The only definition of soundness regarding collaborative processes is given in [35] in the field of the Global Interaction Nets, in order to detect errors in technology-independent collaborative business processes models. However, differently from our work, this approach does not apply to BPMN, which is the modelling notation aimed by our study. Therefore, our investigation of properties at collaboration level provides novel insights with respect to the state-of-the-art of BPMN formal studies.
9 Concluding Remarks

Our study formally defines some important correctness properties, namely well-structuredness, safeness, soundness, and message-disregarding soundness, both at process and collaboration level. By extending their definitions to collaborations, we have been able to investigate safeness and soundness compositionality. Moreover, we demonstrate the relationships between the studied properties, with the aim of classifying BPMN collaboration diagrams according to the properties they satisfy. Specifically, we show that well-structured collaborations represent a subclass of safe ones. In fact, there is a class of collaborations that are safe, even if with an unstructured topology. These models are typically discarded by the modelling approaches in the literature, as they are over suspected of carrying bugs. However, we have shown that some of these models are even sound, hence they can play a significant role in practice.

References

1. OMG: Business Process Model and Notation (BPMN V 2.0) (2011)
## 10 Appendix: Correspondence

<table>
<thead>
<tr>
<th>Graphical Representation</th>
<th>Textual Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="start" /></td>
<td>start($e_{enb}, e_o$)</td>
</tr>
<tr>
<td><img src="image" alt="end" /></td>
<td>end($e_i, e_{cmp}$)</td>
</tr>
<tr>
<td><img src="image" alt="startRcv" /></td>
<td>startRcv($e_{enb}, m, e_o$)</td>
</tr>
<tr>
<td><img src="image" alt="endSnd" /></td>
<td>endSnd($e_i, m, e_{cmp}$)</td>
</tr>
<tr>
<td><img src="image" alt="terminate" /></td>
<td>terminate($e_i$)</td>
</tr>
<tr>
<td><img src="image" alt="eventBased" /></td>
<td>eventBased($e_i, (m_1, e_{a1}), (m_2, e_{a2}), (m_3, e_{a3})$)</td>
</tr>
<tr>
<td><img src="image" alt="andSplit" /></td>
<td>andSplit($e_1, {e_2, e_3, e_4}$)</td>
</tr>
<tr>
<td><img src="image" alt="xorSplit" /></td>
<td>xorSplit($e_1, {e_2, e_3, e_4}$)</td>
</tr>
<tr>
<td><img src="image" alt="andJoin" /></td>
<td>andJoin(${e_1, e_2, e_3}, e_4$)</td>
</tr>
<tr>
<td><img src="image" alt="xorJoin" /></td>
<td>xorJoin(${e_1, e_2, e_3}, e_4$)</td>
</tr>
<tr>
<td><img src="image" alt="task" /></td>
<td>task($e_i, e_o$)</td>
</tr>
<tr>
<td><img src="image" alt="taskRcv" /></td>
<td>taskRcv($e_i, m, e_o$)</td>
</tr>
<tr>
<td><img src="image" alt="taskSnd" /></td>
<td>taskSnd($e_i, m, e_o$)</td>
</tr>
<tr>
<td><img src="image" alt="empty" /></td>
<td>empty($e_i, e_o$)</td>
</tr>
<tr>
<td><img src="image" alt="interRcv" /></td>
<td>interRcv($e_i, m, e_o$)</td>
</tr>
<tr>
<td><img src="image" alt="interSnd" /></td>
<td>interSnd($e_i, m, e_o$)</td>
</tr>
<tr>
<td><img src="image" alt="subProc" /></td>
<td>subProc($e_i, P, e_o$)</td>
</tr>
<tr>
<td><img src="image" alt="pool" /></td>
<td>pool($p, P$)</td>
</tr>
</tbody>
</table>
Here we reported the complete correspondence between the BPMN graphical notation and our syntax. For the sake of presentation, join and split gateways include only three incoming/outgoing branching respectively.
Appendix: Proofs

In this appendix we report the proofs of the results presented in the paper.

**Lemma 1** Let \( P \) be a process, if isInit(\( P, \sigma \)) then \( \langle P, \sigma \rangle \) is cs-safe.

**Proof.** Trivially, from definition of isInit(\( P, \sigma \)). By definition of isInit(\( P, \sigma \)), we have that 
\( \sigma(e_{\text{start}}) = 1 \) \( \forall e \in \text{start}(P) \). Hence, we have that \( \text{maxMarking}(P, \sigma) \leq 1 \), which allows us to conclude. \( \square \)

**Lemma 2** Let \( \text{WSCore}(P) \), and let \( \langle P, \sigma \rangle \) be reachable and cs-safe process configuration, if \( \langle P, \sigma \rangle \xrightarrow{\alpha} \sigma' \) then \( \langle P, \sigma' \rangle \) is cs-safe.

**Proof.** We proceed by induction on the structure of WSCore process elements.

Base cases: we show here only few interesting cases among the multiple base cases; since by hypothesis \( N \) is WS, it can only be either a task or an intermediate event. Let us consider the simple task, since all the other cases are similar. \( \square \)

1. \( P = \text{task}(e_i, e_o) \). By hypothesis \( \langle P, \sigma \rangle \) is cs-safe, then \( \text{maxMarking}(P, \sigma) = \max(\sigma(e_i), \sigma(e_o)) \leq 1 \). The only rule that can be applied to infer the transition \( \langle P, \sigma \rangle \xrightarrow{\alpha} \sigma' \) is P-Task. In order to apply the rule there must be \( \alpha(e) = 0 \). By hypothesis \( \sigma(e_i) = 0 \) and \( \sigma(e_o) = 1 \). Thus, \( \text{maxMarking}(P, \sigma') = \sigma(e_o) \). Since \( \sigma(e_o) = 1 \) we have that \( \text{maxMarking}(P, \sigma') \leq 1 \), which allows us to conclude.

Inductive cases: we consider the following cases, the other are deal with similarly.

1. Let us consider \( \langle \text{andSplit}(e_i, E_o), P_1 | \ldots | P_n | \text{andJoin}(E_i, e_o), \sigma \rangle \). There are the following possibilities:
   - \( \langle \text{andSplit}(e_i, E_o), \sigma \rangle \) evolves by means of rule P-AndSplit. We can exploit the fact that this is a reachable well-structured process configuration to prove that \( \sigma(e_i) = 1 \) \( \forall e \in E_o, \sigma(e) = 0 \). Thus, \( \langle \text{andSplit}(e_i, E_o), \sigma \rangle \xrightarrow{\text{inc}} \langle \text{dec}(\sigma(e_i), E_o) \rangle \). Hence, \( \text{maxMarking}(\text{andSplit}(e_i, E_o), \sigma') = 1 \). By hypothesis \( \langle \text{andSplit}(e_i, E_o) | P_1 | \ldots | P_n | \text{andJoin}(E_i, e_o), \sigma \rangle \) is cs-safe, i.e. if there is a token on the state \( \langle \text{andSplit}(e_i, E_o), \sigma' \rangle \) all the other edges do not have token. This means that cs-safeness is not affected. Therefore, the overall term is cs-safe.
   - Node \( P_1 | \ldots | P_n \) evolves without affecting the split and join gateways. In this case we can easily conclude by inductive hypothesis.
   - Node \( P_1 | \ldots | P_n \) evolves and affects the split and/or join gateways. In this case we can reason like in the first case, by relying on inductive hypothesis.

2. \( \langle \text{andJoin}(E_i, e_o), \sigma \rangle \) evolves by means of rule P-AndJoin. We can exploit the fact that this is a reachable well-structured node to prove that \( \forall e \in E_i, \sigma(e) = 1 \) \( \alpha(e) = 0 \). Thus, \( \langle \text{andJoin}(E_i, e_o), \sigma \rangle \xrightarrow{\text{dec}} \langle \text{dec}(\sigma(e_i), E_o) \rangle \). Hence, \( \text{maxMarking}(\text{andJoin}(E_i, e_o), \sigma') = 1 \). By hypothesis \( \langle \text{andJoin}(E_i, e_o) | P_1 | \ldots | P_n | \text{andJoin}(E_i, e_o), \sigma \rangle \) is cs-safe, i.e. if there is a token on the state \( \langle \text{andJoin}(E_i, e_o), \sigma' \rangle \) all the other edges do not have token. This means that cs-safeness is not affected. Therefore, the overall term is cs-safe.
   - Let us consider \( \text{xorJoin}(\{e_2, e_3\}, e_1) | P_1 | P_2 | \text{xorSplit}(e_4, \{e_2, e_3\}) \) with \( \text{in}(P_1) = \{e_3\}, \text{out}(P_1) = \{e_4\}, \text{in}(P_2) = \{e_5\}, \text{out}(P_2) = \{e_2\} \)
\[ \langle \text{xorJoin}(\{e_2, e_3\}, e_1), \sigma \rangle \] evolves by means of rule \( P-\text{XorJoin} \). We can exploit the fact that this is a reachable well-structured process configuration to prove that the term is marked \( \sigma(e_1) = 0 \) and either \( \sigma(e_2) = 1 \) or \( \sigma(e_3) = 1 \); let us assume the marking is \( \sigma(e_1) = 1 \) (since the other case is similar). Thus \( \langle \text{xorJoin}(\{e_2, e_3\}, e_1), \sigma \rangle \overset{\alpha}{\longrightarrow} \langle \text{inc}\sigma(e_2), e_1 \rangle \). Hence, \( \text{maxMarking}(\text{xorJoin}(\{e_2, e_3\}, e_1), \sigma) \rangle = 1 \). By hypothesis \( \langle \text{xorJoin}(\{e_2, e_3\}, e_1), \sigma \rangle \) is cs-safe, i.e. if there is a token on the state \( \langle \text{xorJoin}(\{e \cup E_1, e_0 \}, \sigma) \rangle \) all the other edges do not have token. This means that cs-safeness is not affected. Therefore, the overall term is cs-safe.

- Node \( P_1 \mid P_2 \) evolves without affecting the split and join gateways. In this case we can easily conclude by inductive hypothesis.

- Node \( P_1 \mid P_2 \) evolves and affects the split xor join and xor split gateways. In this case we can reason like in the first case, by relying on inductive hypothesis.

- \( \langle \text{xorSplit}(e_2, \{e_2, e_3\}), \sigma \rangle \) evolves by means of rule \( P-\text{XorSplit} \). We can exploit the fact that this is a reachable well-structured node to prove that the term is marked either as \( \sigma(e_1) = 1 \). Hence, it evolves in a cs-safe term; in fact let us assume that it evolves in this way \( \langle \text{xorSplit}(e_2, \{e \cup E_1\}), \sigma \rangle \overset{\alpha}{\longrightarrow} \langle \text{inc}\sigma(e_2), e \rangle \). Hence, \( \text{maxMarking}(\text{xorSplit}(e_2, \{e \cup E_1\}), \sigma) \rangle = 1 \). By hypothesis \( \langle \text{xorJoin}(\{e_2, e_3\}, e_1) \mid P_1 \mid P_2 \mid \text{xorSplit}(e_2, \{e_2, e_3\}), \sigma \rangle \) is cs-safe, i.e. if there is a token on the state \( \langle \text{xorSplit}(e_2, \{e_2, e_3\}), \sigma \rangle \) all the other edges do not have token. This means that cs-safeness is not affected. Therefore, the overall term is cs-safe.

- Let us consider \( \langle P, \sigma \rangle = \langle P_1 \mid P_2, \sigma \rangle \). The relevant case for cs-safeness is when \( P \) evolves by applying \( P-\text{Int}_1 \). We have that \( \langle P_1 \mid P_2, \sigma \rangle \overset{\alpha}{\longrightarrow} \sigma' \) with \( \langle P_1, \sigma \rangle \overset{\alpha}{\longrightarrow} \sigma' \). By definition of maxMarking function we have that \( \text{maxMarking}(P, \sigma) = \max(\text{maxMarking}(P_1, \sigma), \text{maxMarking}(P_2, \sigma)) \). By inductive hypothesis we have that \( \text{maxMarking}(P_1, \sigma) = \maxMarking(P_1, \sigma') \leq 1 \) which is cs-safe. Since \( P_2 \) is well structured and cs-safe, then also \( \langle P_2, \sigma' \rangle \) is cs-safe, which permits us to conclude.

\[ \square \]

**Lemma 3** Let \( P \) be WS, and let \( \langle P, \sigma \rangle \) be a process configuration reachable and cs-safe, if \( \langle P, \sigma \rangle \overset{\alpha}{\longrightarrow} \sigma' \) then \( \langle P, \sigma' \rangle \) is cs-safe.

**Proof.** According to Definition \[ P \] can have 6 different forms. We proceed by case analysis on the parallel component of \( \langle P, \sigma \rangle \) that causes the transition \( \langle P, \sigma \rangle \overset{\alpha}{\longrightarrow} \sigma' \).

We show now the case \( P = \text{start}(e_{\text{enb}}, e_0) \mid P' \mid \text{end}(e_1, e_{\text{cmp}}) \).

- \( \text{start}(e_{\text{enb}}, e_0) \) evolves by means of the rule \( P-\text{Start} \). In order to apply the rule there must be \( \sigma(e_{\text{enb}}) > 0 \), hence, by cs-safeness, \( \sigma(e_{\text{enb}}) = 1 \). We can exploit the fact that this is a reachable well-structured configuration to prove that \( \sigma(e_0) = 0 \). The rule produces the following transition \( \langle \text{start}(e_{\text{enb}}, e_0), \sigma \rangle \overset{\alpha}{\longrightarrow} \langle \text{inc}\sigma(e_{\text{enb}}, e_0) \rangle \) where \( \sigma(e_{\text{enb}}) = 0 \) and \( \sigma(e_0) = 1 \). Now, \( \langle P, \sigma \rangle = \langle \text{start}(e_{\text{enb}}, e_0) \mid P' \mid \text{end}(e_1, e_{\text{cmp}}), \sigma \rangle \) can evolve only through the application of \( P-\text{Int}_1 \) producing \( \langle P, \sigma' \rangle \) with \( \sigma(\text{in}(P')) = 1 \).

By hypothesis \( \langle P, \sigma \rangle \) is cs-safe, thus \( \sigma(e_1) \leq 1 \), \( \sigma(e_{\text{cmp}}) \leq 1 \) and \( \max(\sigma(\text{edges}(P'))) \leq 1 \). Now \( \text{maxMarking}(P', \sigma) \leq 1 \) and \( \text{maxMarking}(P', \sigma') \leq 1 \). Therefore \( \text{maxMarking}(P, \sigma') = \max(0, 1, \sigma(\text{in}(P'))), \sigma(\text{out}(P')), \sigma(e_1), \sigma(e_{\text{cmp}}) \leq 1 \), then \( \langle P, \sigma' \rangle \) is cs-safe.

- \( \text{end}(e_1, e_{\text{cmp}}) \) evolves by means of the rule \( P-\text{End} \). We can exploit the fact that this is a reachable well-structured configuration to prove that the term is marked as \( \sigma(e_1) = 1 \) and \( \sigma(e_{\text{cmp}}) = 0 \). The rule produces the following transition \( \langle \text{end}(e_1, e_{\text{cmp}}), \sigma \rangle \overset{\alpha}{\longrightarrow} \langle \text{inc}\sigma(e_1), e_{\text{cmp}} \rangle \). Now, \( \langle P, \sigma \rangle \) can only evolve by applying \( P-\text{Int}_1 \) producing \( \langle P, \sigma' \rangle \).
By hypothesis \( \langle P, \sigma \rangle \) is cs-safe, then \( \sigma(e_i) \leq 1, \sigma(e_{cmp}) \leq 1 \) and \( P' \) is cs-safe. Reasoning as previously we can conclude that \( \langle P, \sigma' \rangle \) is cs-safe.

- \( P' \) moves, that is \( \langle P', \sigma' \rangle \xrightarrow{\alpha} \sigma' \). By Lemma 2 \( \langle P', \sigma' \rangle \) is safe, thus \( \text{maxMarking}(P', \sigma') \leq 1 \). By hypothesis, \( P \) is cs-safe therefore \( \text{maxMarking}(\text{start}(e_{enb}, e_o), \sigma') \leq 1 \). We can conclude that \( \langle P, \sigma' \rangle \) is safe.

Now we consider the case \( P = \text{start}(e_{enb}, e_o) \mid P' \mid \text{terminate}(e_i) \).

- The start event evolves: like the previous case.
- The end terminate event evolves: the only transition we can apply is \( \text{Terminate} \). By applying the rule we have \( \langle \text{terminate}(e_i), \sigma \rangle \xrightarrow{\text{kill}} \text{dec}(\sigma(e_i)) \) with \( \sigma(e_i) > 0 \). Now, \( \langle P, \sigma \rangle \) can only evolve by applying \( \text{P-Kill} \) producing \( \langle P, \sigma' \rangle \) where \( \sigma' \) is completed unmarked; therefore it is cs-safe.
- \( P' \) moves: similar to the previous case.

\[ \square \]

**Theorem 1** Let \( P \) be a process, if \( P \) is well-structured then \( P \) is safe.

**Proof.** We have to show that if \( \langle P, \sigma \rangle \xrightarrow{\ast} \sigma' \) then \( \langle P, \sigma' \rangle \) is cs-safe. We proceed by induction on the length \( n \) of the sequence of transitions from \( \langle P, \sigma \rangle \) to \( \langle P, \sigma' \rangle \).

**Base Case** \( (n = 0) \): In this case \( \sigma = \sigma' \), then \( \text{isInit}(\langle P, \sigma \rangle) \) is satisfied. By Lemma 1, we conclude \( \langle P, \sigma' \rangle \) is cs-safe.

**Inductive Case:** In this case \( \langle P, \sigma \rangle \xrightarrow{\alpha} \langle P, \sigma'' \rangle \xrightarrow{\alpha} \langle P, \sigma' \rangle \) for some process \( \langle P, \sigma'' \rangle \). By induction, \( \langle P, \sigma'' \rangle \) is cs-safe. By applying Lemma 3 to \( \langle P, \sigma'' \rangle \xrightarrow{\alpha} \langle P, \sigma' \rangle \), we conclude \( \langle P, \sigma' \rangle \) is cs-safe.

\[ \square \]

**Theorem 2** Let \( C \) be a collaboration, if \( \text{isWS}(C) \) then \( C \) is safe.

**Proof.** By contradiction, let us assume \( \text{isWS}(C) \) and \( C \) is unsafe. By Definition 8 there exists a collaboration configuration \( \langle C, \sigma', \delta' \rangle \) such that \( \langle C, \sigma, \delta \rangle \xrightarrow{\ast} \langle C, \sigma', \delta' \rangle \) with \( \text{pool}(p, P) \) in \( C \) and \( \langle P, \sigma' \rangle \) not cs-safe. Thus, there exists \( \text{pool}(p, P) \) in \( C \) such that \( \langle P, \sigma \rangle \xrightarrow{\ast} \langle P, \sigma' \rangle \). From hypothesis \( \text{isInit}(\langle C, \delta \rangle) \), we have \( \text{isInit}(\langle P, \sigma \rangle) \). From hypothesis \( \text{isWS}(C) \), we have that \( P \) is WS. Therefore, by Theorem 1 \( P \) is safe. By Definition 7 \( \langle P, \sigma' \rangle \) is cs-safe, which is a contradiction.

\[ \square \]

**Lemma 4** Let \( \langle P, \sigma \rangle \) be a reachable process configuration and \( \text{isWSCore}(P) \), then there exists \( \sigma' \) such that \( \langle P, \sigma \rangle \xrightarrow{\ast} \sigma' \) and \( \text{isCompleteEl}(P, \sigma') \).

**Proof.** We proceed by induction on the structure of \( \text{isWSCore}(P) \). Base cases: by definition of \( \text{isWSCore}(P) \), \( P \) can only be either a task or an intermediate event; we show here only the case in which it is a non communicating task, the other are dealt with similarly.

- \( P = \text{task}(e_i, e_o) \). The only rule we can apply is \( \text{P-Task} \). In order to apply the rule there must be \( \sigma(e_i) > 0 \). Since \( \text{isWSCore}(P) \), \( \langle P, \sigma \rangle \) is safe, hence \( \sigma(e_i) = 1 \). Since the process configuration is reachable we have \( \sigma(e_o) = 0 \). The application of the rule produces \( \langle \text{task}(e_i, e_o), \sigma \rangle \xrightarrow{\text{inc}(\text{dec}(e_i), e_o)} \langle \sigma(e_i) = 0 \text{ and } \sigma(e_o) = 1, \rangle \text{ which permits us to conclude.

Inductive cases: we consider one case, the other are dealt with similarly.

- Let us consider \( P = (\text{andSplit}(e_i, E_o) \mid P_1 \mid \ldots \mid P_n \mid \text{andJoin}(E_i, e_o), \sigma) \). There are the following possibilities:
• $\langle \text{andSplit}(e_1, E_o), \sigma \rangle$ evolves by means of rule $P$-AndSplit. We can exploit the fact that this is a reachable well-structured process configuration to prove that $\sigma(e_1) = 1$ and $\forall e \in E_o, \sigma(e) = 0$. Thus, $\langle \text{andSplit}(e_1, E_o), \sigma \rangle \overset{i}{\rightarrow} \langle \text{inc}(\text{dec}(\sigma, e_1), E_o) \rangle$. Now, $P$ can evolve only through the application of $P$-Int$_1$ producing $\langle P, \sigma'_1 \rangle$ with $\sigma'_1(\text{in}(P_i)) = \ldots = \sigma''(\text{in}(P_n)) = 1$. By inductive hypothesis there exists a state $\sigma'_1$ such that $\text{isCompleteEl}(P_1 | \ldots | P_n, \sigma'_1)$. Now, $P$ can only evolve by applying rule $P$-Int$_1$, producing $\langle P, \sigma'_2 \rangle$ with $\sigma'_2(\text{edges}(E_i)) = 1$. Now, $\langle \text{andJoin}(E_i, e_o), \sigma'_2 \rangle$ can evolve by means of rule $P$-AndJoin. The application of the rule produces $\langle \text{andJoin}(E_i, e_o), \sigma'_2 \rangle \overset{i}{\rightarrow} \langle \text{inc}(\text{dec}(\sigma, E_i), e_o) \rangle$, i.e. $\sigma(e_o) = 1$ and $\forall e \in E_i, \sigma(e) = 0$. This permits us to conclude.

• $P_1 | \ldots | P_n$ evolves without affecting the split and join gateways. In this case we can easily conclude by inductive hypothesis.

• $P_1 | \ldots | P_n$ evolves and affects the split and/or join gateways. In this case we can reason like in the first case.

$\square$

**Theorem 3** Let $\langle P, \sigma \rangle$ be a WS process configuration, then $\langle P, \sigma \rangle$ is sound.

**Proof.** According to Definition $\text{isInit}(P, \sigma)$. Thus we have that $\sigma(\text{start}(P)) = 1$, and $\forall e \in \text{edges}(P) \setminus \text{start}(P), \sigma(e) = 0$. Therefore the only parallel component of $P$ that can infer a transition is the start event. In this case we can apply only the rule $P$-Start. The rule produces the following transition, $\langle \text{start}(e_{\text{enb}}, e_o), \sigma \rangle \overset{i}{\rightarrow} \langle \text{inc}(\text{dec}(\sigma, e_{\text{enb}}), e_o) \rangle$ where $\sigma(e_{\text{enb}}) = 0$ and $\sigma(e_o) = 1$. Now $\langle P, \sigma \rangle$ can evolve through the application of rule $P$-Int$_1$ producing $\langle P, \sigma'_1 \rangle$, with $\sigma'_1(\text{in}(P')) = 1$. Now $P'$ moves. By hypothesis $\text{isWSCore}(P')$, thus by Lemma 2 there exists a process configuration $\langle P', \sigma'_2 \rangle$ such that $\langle P, \sigma \rangle \overset{\ast}{\rightarrow} \langle P', \sigma'_2 \rangle$ and $\text{isCompleteEl}(P', \sigma'_2)$. The process can now evolve thorough rule $P$-Int$_1$. By hypothesis the process is WS, thus, after the application of the rule we obtain $\langle \text{start}(e_{\text{enb}}, e_o), P' | \text{end}(e_i, e_{\text{cmp}}), \sigma'_3 \rangle$, where $\sigma'_3(e_i) = 1$ and $\forall e \in \text{edges}(P'), \sigma'_3(e) = 0$. We can now apply rule $P$-End that decrements the token in $e_i$ and produces a token in $e_{\text{cmp}}$, which permits us to conclude. $\square$

**Theorem 4** Let $C$ be a collaboration, $C$ is WS does not imply $C$ is sound.

**Proof.** Let $C$ be a WS collaboration, and let us suppose that $C$ is sound. Then, it is sufficient to show a counter example, i.e. a WS collaboration that is not sound. Let us consider, for instance, the collaboration in Fig. 22. By Definition, the collaboration is WS. The soundness of the collaboration instead depends on the evaluation of the condition of the XOR-Split gateway in ORG A. If a token is produced on the upper flow and Task A is executed then Task C in ORG B will never receive the message and the AND-Join gateway can not be activated, thus the process of ORG B can not complete its execution. $\square$

**Theorem 5** Let $C$ be a collaboration, $C$ is WS does not imply $C$ is message-disregarding sound.

**Proof.** Let $C$ be a WS collaboration, and let us suppose that $C$ is sound. Then, it is sufficient to show a counter example, i.e. a WS collaboration that is not sound. We can consider again the collaboration in Fig. 22. By reasoning as previously, the message-disregarding soundness of the collaboration instead depends on the evaluation of the condition of the XOR-Split gateway in ORG A. This permits us to conclude. $\square$

**Theorem 6** Let $C$ be a collaboration, $C$ is unsafe does not imply $C$ is unsound.

41
**Proof.** Let $C$ be a unsafe collaboration, and let us suppose that $C$ is unsound. Then, it is sufficient to show a counter example, i.e. a unsafe collaboration that is sound. We can consider the collaboration in Fig. [22]. Process in ORG A and ORG B are trivially unsafe, since the AND split gateway produces two tokens that are then merged by the XOR join gateway producing two tokens on the outgoing edge of the XOR join. By definition of safeness collaboration the considered collaboration is unsafe. Concerning soundness, processes of ORG B and ORG A are sound. In fact, in each process, after one token enables the terminate end event, the kill label is produced and the second token in the sequence flow is removed (rule P-Terminate), permits the successfully termination of the collaboration. Thus, the resulting collaboration is sound. \[\square\]
Theorem 7 Let C be a collaboration, if all processes in C are safe then C is safe.

Proof. By contradiction let C be unsafe, i.e. there exists a collaboration \( \langle C, \sigma', \delta' \rangle \) such that \( \langle C, \sigma, \delta \rangle \rightarrow^* \langle \sigma', \delta' \rangle \) with \( \text{pool}(p, P) \) in C and \( \langle P, \sigma' \rangle \) not cs-safe. By hypothesis all processes of C are safe, hence it is safe the process, say \( P \), of organisation p. As \( \langle C, \sigma', \delta' \rangle \) results from the evolution of \( \langle C, \sigma, \delta \rangle \), the process \( \langle P, \sigma' \rangle \) must derive from \( \langle P, \sigma \rangle \) as well, that is \( \langle P, \sigma \rangle \rightarrow^* \sigma' \).

By safeness of \( P \), we have that \( \langle P, \sigma \rangle \) is cs-safe, which is a contradiction. □

Theorem 8 Let \( P \) be a process including a sub-process \( \text{subProc}(e_i, P_1, e_o) \), if \( P_1 \) is unsafe then \( P \) is unsafe.

Proof. Let us suppose \( P = \text{subProc}(e_i, P_1, e_o) \mid P_2 \) By contradiction let \( P \) be safe, i.e. given \( \sigma \) such that \( \text{isInit}(P, \sigma) \), for all \( \sigma' \) such that \( \langle P, \sigma \rangle \rightarrow^* \sigma' \) we have that \( \langle P, \sigma \rangle \) is cs-safe. By hypothesis \( P_1 \) is unsafe, i.e. given \( \sigma'_1 \) such that \( \text{isInit}(P_1, \sigma'_1) \), there exists \( \sigma'_2 \) such that \( \langle P_1, \sigma'_1 \rangle \rightarrow^* \sigma'_2 \) and \( \langle P_1, \sigma'_2 \rangle \) not cs-safe. Thus, we have \( \text{maxMarking}(P_1, \sigma'_2) \geq 1 \). By definition of function \( \text{maxMarking}() \), we have that \( \text{maxMarking}(P, \sigma'_2) = \text{max}(\text{maxMarking} \text{subProc}(e_i, P_1, e_o)), \text{maxMarking}(P_2)) = \text{maxMarking}(P_1, \sigma'_2) \geq 1 \).

Thus, \( P \) is not cs-safe, which is a contradiction. □

Theorem 9 Let C be a collaboration, if all processes in C are unsound then C is unsound.

Proof. Let \( P_1 \) and \( P_2 \) be two unsound processes and let \( C \) be the collaboration obtained putting together \( P_1 \) and \( P_2 \). By contradiction let \( C \) be sound, i.e., given \( \sigma \) and \( \delta \) such that \( \text{isInit}(C, \sigma, \delta) \), for all \( \sigma' \) and \( \delta' \) such that \( \langle C, \sigma, \delta \rangle \rightarrow^* \langle \sigma', \delta' \rangle \) we have that there exist \( \sigma'' \) and \( \delta'' \) such that \( \langle C, \sigma', \delta' \rangle \rightarrow^* \langle \sigma'', \delta'' \rangle \), and \( \forall P \in C \) we have that \( \langle P, \sigma'' \rangle \) is cs-safe and \( \forall m \in M. \delta''(m) = 0 \). Since \( P_1 \) and \( P_2 \) are unsound, we have, for instance, that, given \( \sigma'_1 \), such that \( \text{isInit}(P_1, \sigma'_1) \), for all \( \sigma'_2 \) such that \( \langle P, \sigma \rangle \rightarrow^* \sigma'_2 \) we have that there not exists \( \sigma'_3 \) such that \( \langle P, \sigma'_2 \rangle \rightarrow^* \sigma'_3 \), and \( \langle P, \sigma'_3 \rangle \) is cs-safe. Choosing \( \langle C, \sigma', \delta' \rangle \) such that \( \text{pool}(p, P_1) \) in \( C' \), by unsoundness of \( P_1 \) we have that there exists a process in \( C' \) that is not cs-safe, which is a contradiction. □

Theorem 10 Let \( P \) be a process including a sub-process \( \text{subProc}(e_i, P_1, e_o) \), if \( P_1 \) is unsound does not imply \( P \) is unsound.

Proof. Let \( P_1 \) be a unsound, and let us suppose that \( P \) is unsound. Then, it is sufficient to show a counter example, i.e. an sound process including an unsound sub-process. We can consider process in Fig. 23. The process is unsound since when there is a token in the end event of ORG A there is still a pending sequence token to be consumed. If we include the part of the model generating multiple tokens in the scope of a sub-process, as is shown in Fig. 24 that is when the process includes a sub-process, the process is sound. In fact, when there is a token in the end event of ORG A no other pending sequence token need to be processed. □
Fig. 23: An example of unsound process.

Fig. 24: An example of sound process with unsound sub-processes.