Parallelism Insertion Formalisation

Quentin Nivon, Gwen Salaün

September 16, 2025

At this stage of the BPMN process generation, two operators of the language are still to be managed: the ',' and the '&' operators. The ',' operator separates tasks that are not constrained to each others, while the '&' operator separates tasks that should be put in parallel. However, in BPMN, parallelism precisely describes the absence of constraints between elements, as two parallel elements may end up executing sequentially in any order, or at the same time. Thus, we decided to manage the ',' and the '&' operators the same way.

In this approach, adding parallelism to the graph consists in replacing the exclusive split (resp. merge) gateways by parallel split (resp. merge) gateways. However, replacing an exclusive gateway by a parallel gateway may lead to potentially severe issues if not done carefully. In BPMN, parallelism can induce two major issues: *deadlocks* and *livelocks*. A deadlock occurs whenever a parallel merge gateway does not (and will not) receive a sufficient number of tokens to be triggered, that is, one token per incoming flow. This phenomenon prevents the gateway from merging its incoming tokens, and thus from sending a token to its outgoing flow. Consequently, the process cannot complete its execution. A livelock occurs whenever a parallel split gateway can be reached infinitely often by a token that it sent and which was not merged with its siblings before reaching this parallel split again. This parallel split thus produces new tokens infinitely often, thus preventing the process from terminating. Such behaviour happens when there is no node *synchronising all the paths* starting from the children of a parallel split gateway that can reach itself.

Definition 1 (Paths Synchronisation Node)**.** *Let $G = (V,\ E,\ \Sigma)$ be a BPMN process, and $\mathcal{P}_G$ be its corresponding set of paths. A node $v \in V$ is said to* synchronise a set of paths $\mathcal{P}'_G \subseteq \mathcal{P}_G$ *if and only if $\forall p \in \mathcal{P}'_G,\ v \in p$. $\mathcal{P}'_G$ may have several synchronisation nodes, which can be computed using the $\texttt{sync}(\mathcal{P}'_G)$ operator:*

$$\texttt{sync}(\mathcal{P}'_G) = \begin{cases} v_1,\ ...,\ v_n & \text{if } \forall i \in [1...n],\ \forall p \in \mathcal{P}'_G,\ v_i \in p \\ \bot & \text{otherwise} \end{cases}$$

Definition 2 ((Syntactic) Livelock)**.** *Let $G = (V,\ E,\ \Sigma)$ be a BPMN process, and let $v \in V$ be a parallel split gateway. G contains a* (syntactic) livelock *if:*

- $\exists v_c \in \texttt{child}(v)$ *such that* $v_c \xrightarrow{R} v$ *(v_c can reach v);*
- $\nexists v_s \in V$ *such that:*
  - $v_s \in \texttt{sync}(\bigcup_{v_c \in \texttt{child}(v)} \mathcal{P}_G(v_c))$*;*
  - $\forall p = (v_1, ..., v_s, ..., v_n) \in \bigcup_{v_c \in \texttt{child}(v)} \mathcal{P}_G(v_c),\ \exists i \in [1...n] \mid v_i = v \Rightarrow \texttt{index}(v_s) < i.$

(either the paths starting from children of v are not synchronised, or at least one of them can reach v before reaching v_s)

Example. Figure **??** illustrates this potential issue on a simple BPMN process. As the reader can see, the parallel split gateway sends a token τ_1 to B, which reaches the end event, and another token τ_2 to A. These two tokens are never merged, and τ_2 eventually reaches the parallel split gateway. When it receives τ_2, it sends τ_1' to B, and τ_2' to A. As the parallel split gateway is triggered infinitely often, there is no way for this process to eventually complete its execution, as it has no possibility to prevent itself from producing new tokens. Thus, this process is infinitely recursive.

0.1 Insertion of the Parallel Gateways

As mentioned earlier, inserting parallel gateways to the graph mostly consists in replacing some exclusive gateways by parallel ones while avoiding syntactic livelocks. However, simply switching the type of a gateway is often not sufficient to handle properly all the possible forms that the process could take. Indeed, a simple exclusive split gateway with n children tasks could generate almost $2 \times (2^n - 1)$ syntactically different BPMN constructs.

Example. Let us consider the four mutual exclusion constraints $A \mid B$, $A \mid C$, $A \mid D$, and $C \mid D$. Before starting to insert parallelism in it, the BPMN subprocess corresponding to these four constraints would be the one shown in Figure **??**. However, if one wants to parallelise the tasks that can be parallelised without breaking any of the four aforementioned mutual exclusions, while adding as much parallelism as he can, he would build the BPMN sub-process displayed in Figure **??**. It is as parallelised as possible, as the only non-parallel tasks are the ones that should be mutually exclusive. For instance, according to the constraints, tasks B and C are not mutually exclusive, although they are in Figure **??**. However, after the insertion of parallel gateways, they end up in parallel, as they should be.

0.1.1 Inserting Parallel Splits

The generation of parallel structures, such as the one presented above, relies on combinatorics to build all the possible such structures given a set of tasks. This is ensured by a function $\eta : V^n \to W$, assuming that W is the set of all existing workflows. Function η is recursively defined and its return value is, for clarity, written in the form of expressions compliant with the language defined in Section **??**.

Definition 3 (Generation of Parallel Split Structures)**.** *Let $G = (V, E, \Sigma)$ be a BPMN process, and let $g \in V$ be an exclusive split gateway having n children tasks $(t_1, ..., t_n) \in V$ whose parallel structures have to be generated. Function η is defined as:*

$$\eta(\{t_1, ..., t_n\}) = \begin{cases} \bot & \text{if } |\{t_1, ..., t_n\}| = 0 \\ t_1 & \text{if } |\{t_1, ..., t_n\}| = 1 \\ \{\eta(\widehat{\{t_1, ..., t_n\}}) \mid \eta(\overline{\{t_1, ..., t_n\}})\} \cup \{\eta(\widehat{\{t_1, ..., t_n\}}) \ \& \ \eta(\overline{\{t_1, ..., t_n\}})\} & \text{o/w} \end{cases}$$

where $\widehat{\{t_1, ..., t_n\}} = \texttt{any}(2^{\{t_1,...,t_n\}}_{1^+}, 1)$ *and* $\overline{\{t_1, ..., t_n\}} = \{t_1, ..., t_n\} \setminus \widehat{\{t_1, ..., t_n\}}$.

Example. Given 3 tasks A, B, and C, the η function generates 12 syntactically different BPMN subprocesses, that are presented in Figure **??**.

Applying this operation generates several syntactically different graphs. These graphs are built in a BPMN-like fashion, which allows the appliance of the minimisation rules described in Definition **??**. The applications of these rules returns a set of graphs that are now semantically different. For each of them, a copy of G is made, in which the exclusive split from which these graphs were produced is replaced by the current generated graph. Among all these copies, only the ones not creating any syntactic livelock, nor violating any desired mutual exclusion, are kept, while the others are discarded. They are called *syntactically compliant processes.*

Definition 4 (Syntactically Compliant BPMN Process)**.** *Let* $G = (V, E, \Sigma)$ *be a BPMN process.* G *is said to be* syntactically compliant *if* $\forall v \in V$ *such that* $\theta(v) = \diamondsuit_S$, v *is* syntactically compliant.

Remark 1. *A BPMN process containing no parallel split gateway is trivially considered as syntactically compliant.*

Definition 5 (Syntactically Compliant Parallel Split Gateway)**.** *Let* $G = (V, E, \Sigma)$ *be a graph.* $\forall v \in V$ *such that* $\theta(v) = \diamondsuit_S$, v *is said to be* syntactically compliant *if and only if:*

- v *does not create any* syntactic livelock *(i.e., it complies with Definition 2);*
- v *does not break any* desired mutual exclusion*, i.e.,* $\forall v_1, v_2 \in \texttt{child}(v),\ v_1 \neq v_2$:

  $(\forall t_1 \in T_1,\ \nexists t_2 \in T_2 \mid t_2 \in \texttt{mutex}(t_1)) \ \wedge\ (\forall t_2 \in T_2,\ \nexists t_1 \in T_1 \mid t_1 \in \texttt{mutex}(t_2))$

  where $\forall i \in \{1, 2\},\ T_i = \bigcup_{p \in \mathcal{P}_G(v_i)} \texttt{tasks}(p[\texttt{:sync}(\mathcal{P}_G(v_1) \cup \mathcal{P}_G(v_2))])$

0.1.2 Inserting Parallel Merges

Each copy of G now contains its parallel split gateways, but no parallel merge gateways yet. The insertion of these parallel merge gateways, essential to prevent deadlocks and livelocks in the process, is done in two sequential steps. First, the parallel split gateways previously added are analysed to check whether they require a synchronisation node to avoid creating syntactic livelocks in the process. If this is the case, a parallel merge gateway is inserted before this synchronisation node, and becomes the new synchronisation node of the parallel split. This ensures that the parallel split gateway will not create syntactic livelocks in the final process. Next, the remaining exclusive merge gateways of the BPMN process are checked to see whether their closest common ancestor is a parallel split. If this is the case, a parallel version of this gateway may not suffer from deadlocks, so it is switched to a parallel gateway.

0.2 Detection of Deadlocks/Livelocks and Parallelism Removal

Our previous modifications of the copies of G introduced parallelism in them. Although performing this parallelisation phase carefully, it is rather complex to ensure the absence of deadlocks or livelocks at design time by performing only a syntactic analysis of the process. However, such behaviours can be easily detected when *executing* the BPMN process.

Definition 6 ((Execution) Deadlock)**.** *Let* $G = (V, E, \Sigma)$ *be a BPMN process.* G *contains* a(n execution) deadlock *whenever* $\exists C \in \mathcal{H}(G)$ *such that:*

- $C = \texttt{push}(C)$*;*
- C *is not a* final configuration.

Definition 7 ((Execution) Livelock)**.** *Let* $G = (V, E, \Sigma)$ *be a BPMN process.* G *contains* a(n execution) livelock *whenever* $\exists C, C' \in \mathcal{H}(G)$ *such that:*

- $\forall n \in C,\ C[n] = 0 \Leftrightarrow C'[n] = 0$*;*
- $\forall n \in C,\ C'[n] \geq C[n]$*;*
- $\exists n \in C$ *such that* $C'[n] > C[n]$.

0.2.1 Detection of Deadlocks/Livelocks

The detection of such configurations is based on simulation of the given BPMN process. However, the simulation that we use to detect such erroneous configurations slightly differs from the one presented in Section **??**. Indeed, in Section **??**, we perform one simulation of the process, representing one of its possible executions. Here, we want to ensure that there is no possible execution of the process that reaches a deadlock or a livelock. In our context, there is a single type of node possibly

making the execution of a process differ from another: the exclusive split gateway. Indeed, when a token reaches this node, it is sent to any of its children nodes, non-deterministically. To ensure that the detection performs on each possible configuration of the process, the solution that we opted for consists in duplicating the current configuration every time a token must be sent away from an exclusive split gateway. The simulation no longer returns a single history $\mathcal{H}$, but a set of histories $S_{\mathcal{H}} = \{\mathcal{H}_1, ..., \mathcal{H}_n\}$, each of which corresponds to a possible execution of the process. However, this modification introduces a major issue: a simulation may no longer terminate. Originally, simulating a correct process (i.e., without deadlock/livelock) necessarily terminates, due to the semantics of the different BPMN operators, and to the fact that the children of an exclusive split gateway are probabilistically chosen. From now on, as all the children of an exclusive split gateway receive a token from their parent, certain configurations may remain in a state where, for instance, a token is continuously circulating through a strongly connected component of the process. To avoid such situations, the simulation now makes use of a *fixed point* analysis.

Definition 8 (Fixed Point)**.** *Let* $G = (V, E, \Sigma)$ *be a BPMN process.* $\mathcal{H}(G)$ *contains a* fixed point *whenever* $\exists C, C' \in \mathcal{H}(G)$ *such that:*

— $\forall n \in C,\ C[n] = C'[n]$*;*
— $\forall n' \in C',\ C[n'] = C'[n']$*.*

When a fixed point is reached, the simulation stops generating all the possible configurations, and is asked to terminate. This is ensured by a mechanism that forces the simulator to transmit the tokens of an exclusive split gateway only to its child that is the closest to an end event. When the simulation has terminated, the set of histories $S_{\mathcal{H}}$ is analysed to verify whether there exists an history $\mathcal{H} \in S_{\mathcal{H}}$ containing a deadlock or a livelock. If not, the BPMN process remains as is. Otherwise, some of its parallel elements have to be removed.

0.2.2 Parallelism Removal

If the deadlock/livelock detector found a deadlock or a livelock in the current BPMN process, this process has to be modified. Although being rather simple to detect deadlocks/livelocks in processes, it is more difficult to identify their sources. For instance, a node holding a token in a deadlock configuration may not be the source of that deadlock. The simplest solution that we found to identify the source of such errors so far consists in removing step by step the parallel gateways of the BPMN process, until reaching a graph containing no deadlock/livelock. The removal is made in a simple way: each parallel gateway of the process is replaced by an exclusive one, which leads to the generation of several new BPMN processes, each of them containing one less parallel gateway. If the removed gateway is a parallel merge gateway, and if this gateway is the mandatory synchronisation node of a parallel split gateway, this parallel split gateway is also replaced by an exclusive split gateway to avoid syntactic livelocks. Among all the deadlock/livelock-free generated BPMN processes, the one with the largest number of parallel tasks is elected as best candidate. This procedure is summarised in Algorithm 1. This final BPMN process now satisfies a new set of constraints $Cons_4$.

Algorithm 1 Algorithm for Generating the Most Parallel Process

Inputs: $G = (V, E, \Sigma)$ (BPMN Process), M_S (Split with Mandatory Merges)
Output: G_P (Most Parallel BPMN Process)

$S_{next} \leftarrow []$
$G_P \leftarrow \perp$

if $\neg$hasDeadlockOrLivelock(G) **then**
    $G_P \leftarrow$ getMostParallelProcessBetween(G, G_P)
end if

for $v \in V$ **do**
    if $\theta(v) =$  **then**
        $G' \leftarrow$ copy(G) where $\theta(v) =$  ▷ v is a $\Rightarrow$ v becomes a
        $S_{next} \leftarrow S_{next} \cup [G']$
    else if $\theta(v) =$  **then**
        if $M_S[v] \neq \perp$ **then** ▷ v is a mandatory merge $\Rightarrow$ v and $M_S[v]$ become 
            $G' \leftarrow$ copy(G) where $\theta(v) =$ and $\theta(M_S[v]) =$
        else ▷ v is not a mandatory merge $\Rightarrow$ only v becomes a 
            $G' \leftarrow$ copy(G) where $\theta(v) =$
        end if
        $S_{next} \leftarrow S_{next} \cup [G']$
    end if
end for

for $G' \in S_{next}$ **do**
    $G_P \leftarrow$ getMostParallelProcessBetween(this(G'), G_P)
end for

return G_P