A Decomposed Hybrid Approach to Business Process Modeling with LLMs Technical Report

Ali Nour Eldin[1,2], Nour Assy[2], Olan Anesini[2], Benjamin Dalmas[2], and Walid Gaaloul[1]

[1] Telecom SudParis, Institut Polytechnique de Paris, France,
email: {name}.{last_name}@telecom-sudparis.eu
[2] Bonitasoft, France, emails: {name}.{last-name}@bonitasoft.com

1 Objective

This report details the Structured Algorithm for Model Generation by incorporating the entire algorithm into the document.

2 Preliminaries

In this section, we give some definitions related to process models that will be used in the subsequent sections.

A process model is a set of connected entities that formally represent process behavior. Since we use BPMN as the modeling language, entities can include flow objects (activities, events, and gateways), swimlanes (pools and lanes), artifacts (e.g., data objects, text annotations, or groups), and connecting objects (sequence flows, message flows, and associations). In this work, we limit the scope to entities shared among different languages, specifically flow objects represented by *activities*, *start/end events*, and *gateways* (including parallel and exclusive gateways), and connecting objects represented by *sequence flows*. Two graph structures will be used throughout the paper: a *dependency graph*, which captures the partial dependencies in the execution order of activities, and a *BPMN process model*.

Definition 1 (Dependency Graph). *A dependency graph is defined as a directed graph $DG = (V, E)$, where:*

- *V represents a finite set of vertices, each corresponding to an activity or start/end event.*
- *$E \subseteq V \times V$ represents a set of directed edges, where each edge $(u, v) \in E$ signifies that v can be executed right after the completion of u.*
- *$v_s \in V$ is the starting vertex, iff $\nexists (v, v_s) \in E$ [3].*
- *$v_e \in V$ is an ending vertex, iff $\nexists (v_e, v) \in E$.*

[3] Since we do not support pools, there is always one source vertex

- *For $e = (u, v) \in E$, we use the notations $e.source = u$ and $e.target = v$.*
- *For $v \in V$, the direct successors of v is* $\mathrm{Succ}_{DG}(v) = \{u \in V \mid (v, u) \in E\}$ *and the direct predecessors of is* $\mathrm{Pred}_{DG}(v) = \{u \in V \mid (u, v) \in S\}$*. When it is clear from context, the subscript DG is omitted.*

Let $DG = (V, E)$ be a dependency graph. We define paths, cycles and branching relations are as follows:

Definition 2 (Path). *A path from $u \in V$ to $v \in V$, denoted as $P_{u,v} = \langle (v_1, v_2), \ldots, (v_{n-1}, v_n) \rangle$, is the sequence of unique edges leading from u to v where $\forall 1 \leq i, j \leq n-1, i \neq j, n \geq 2$, $(v_i, v_{i+1}) \in E$, $v_1 = u, v_n = v$, and $\nexists (v_i, v_{i+1}) = (v_j, v_{j+1})$. We denote by:*

- *$P^v_{u,v} = \{v_1, \ldots, v_n\}$: the set of the path vertices.*
- *$P^s_{u,v} = (v_1, v_2)$: the first edge of the path.*
- *$P^e_{u,v} = (v_{n-1}, v_n)$: the last edge of the path.*
- *$\mathbb{P}_{u,v}$: the possibly empty set of all paths from u to v.*

Definition 3 (Cycle). *A cycle C_u is a path $P_{u,u} = \langle (u, v_1), \ldots, (v_n, u) \rangle$ that starts and ends with u such that $\forall 1 \leq i \leq n, v_i \neq u$. We denote by $\mathbb{C}_u$ the possibly empty set of all cycles of u.*

Definition 4 (Branching relation (parallel, exclusive)). *Let $v \in V$ be a vertex such that* $\mathrm{Succ}(v) = \{v_1, \ldots, v_n\}$ *and $n \geq 2$. Let $B = \{\{a, b\} \mid a, b \in$* $\mathrm{Succ}(v)$ *and $a \neq b\}$ be the set of all unique pairs of* $\mathrm{Succ}(v)$*. We denote by:*

- *$\| \subseteq B$ the set of pairs of vertices that are in a parallel branching relation.*
- *$\# \subseteq B$ the set of pairs of vertices that are in an exclusive branching relation.*
- *$\|(u) = \{v_0, v_1, \ldots, v_n\} \mid \{\{u, v_0\}, \ldots, \{u, v_n\}\} \subseteq \|$*

Definition 5 (BPMN Process Model). *A BPMN process model is defined as $M = (N, G, S)$, where:*

- *$N = V$ is the set of vertices in DG.*
- *G is a finite set of gateways. A gateway $g \in G$ can be parallel, represented by $+$ or exclusive represented by $\times$. We denote by $type(g) \in \{\times, +\}$ the function that returns the type of g.*
- *$S \subseteq (N \times G) \cup (N \times G)$ is a set of directed sequence flows. For each $s = (u, v) \in S$, we use the notations $s.source = u$, and $s.target = v$.*
- *For $u \in N \cup G$, the direct successors of u is* $\mathrm{Succ}_M(u) = \{v \in N \cup G \mid (u, v) \in S\}$ *and the direct predecessors is* $\mathrm{Pred}_M(u) = \{v \in N \cup G \mid (v, u) \in S\}$*. When it is clear from context, subscript M is omitted.*
- *For $u \in N \cup G$, the transitive successors of u is* $\mathrm{TSucc}(u) = \{v \in N \mid \exists g_1, \ldots, g_k \in G, (u, g_1), (g_k, v), (g_i, g_{i+1}) \in S\}$ *and the transitive predecessors of u is* $\mathrm{TPred}(u) = \{v \in N \mid \exists g_1, \ldots, g_k \in G, (v, g_1), (g_k, u), (g_i, g_{i+1}) \in S\}$*. For $u \in N$,* $\mathrm{TSucc}(u) = \mathrm{Succ}_G(u)$ *and* $\mathrm{TPred}(u) = \mathrm{Pred}_G(u)$*.*

3 Structured Algorithm for Model Generation

The dependency graph G and the branching relations, parallel $||$ and exclusive $\#$ returned by the first component are passed to this second component which is responsible for constructing the corresponding BPMN diagram M. This step is similar to a process discovery problem [2], where the goal is to accurately translate the relations between vertices (dependency, parallel, and exclusive) extracted from the traces of an event log into a process model. However, the main difference lies in the definition of these relations due to the differing nature of the inputs (unstructured text vs. structured traces). Therefore, we proposed an algorithm inspired by the internal workings of existing process discovery algorithms that constructs the BPMN in three steps. The first step involves analyzing the dependency graph to detect and filter loops, as these are known to pose challenges [1] (detailed in Section 3.1). Next, an algorithm constructs the BPMN model with a focus on discovering split and join gateways, which is a non-trivial task given our aim to create simple and understandable process models (detailed in Section 3.2). Finally, the filtered loops are added to the process model (detailed in Section 3.3).

3.1 Loop Filtering

This step involves removing the loops from the dependency graph DG by detecting and filtering the dependency relations in E that break the cycles which results in an acyclic dependency graph. For example, given the dependency graph in Figure 1, there are four cycles $C_{e1} = \langle (e, e) \rangle$, $C_{e2} = \langle (e, f), (f, e) \rangle$, $C_{f_1} = \langle (f, f) \rangle$ and $C_{f_2} = \langle (f, e), (e, f) \rangle$.

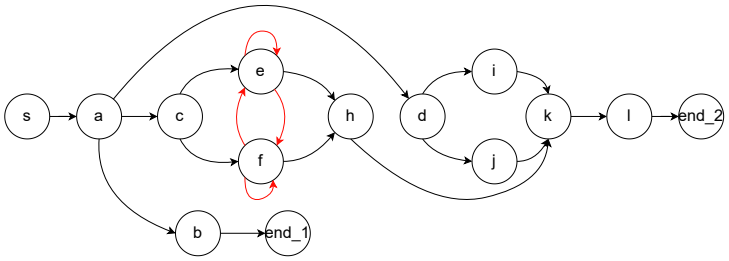


Fig. 1: Dependency graph of the running example with cycles

Breaking cycles involves removing specific edges, known as *looping edges*, which are the last edges in the paths that create the looping behavior. For example, given the cycle $C_{e2} = \langle (e, f), (f, e) \rangle$, the edge (f, e) creates the loop. It is characterized by the property that, following the path from the graph's start to the edge's source vertex, no other vertices in the cycle are revisited before

reaching this edge. In this example, the looping edges are (e, e), (f, e), (f, f), and (e, f). Thus, all the looping edges are removed to break the loops.

The DG in Figure 2 provides another example. The cycles in this example are $C_a^1 = \langle (a, b), (b, c), (c, a) \rangle$, $C_c^1 = \langle (c, a), (a, b), (b, c) \rangle$, and $C_b^1 = \langle (b, c), (c, a), (a, b) \rangle$. To break the loop, following the path from the graph's start to the edge's source vertex of the cycles, the looping edge is characterized by the property that no other vertices in the cycle are revisited before reaching this edge. Applying the path condition in Definition 6, only C_a^1 is valid. Therefore, $L = \{(c, a)\}$.

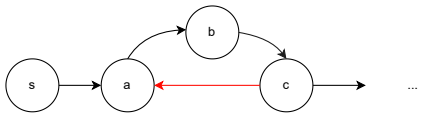


Fig. 2: An example of a long loop.

These are formalized in Definition 6 as follows:

Definition 6 (Looping edge). *Let C_u be a cycle of $u \in V$. The looping edge e^{loop} is an edge $e \in C_u^E$ such that $P^V_{v_s, C_u^S.source} \cap \left(C_u^V \setminus \{C_u^S.source\}\right) = \emptyset$.*

After filtering all the looping edges, an acyclic dependency graph, denoted as DG_A, is obtained and used in the next step to discover the split and join gateways. The result after extract looping edge is the removal of the red edges from the Dependency Graph in the Figures 1 and 2.

3.2 Split & Join Discovery

Accurately discovering split and join gateways while keeping the model simple is challenging. To simplify model generation, we construct blocks of split/join gateways without enforcing the block-structuredness property. Algorithm 1 outlines the high-level steps. It takes as input the acyclic dependency graph DG_A and the branching parallel relations $\parallel$ and returns as output a BPMN model M.

Initially, we traverse all paths in DG_A from start to all possible ends, ordering them with a depth-first algorithm based on the longest path. This ensures joins are introduced alongside splits, focusing on identifying the maximum number of gateways by analyzing the longest paths first. The result is a list of ordered edges derived from these paths (Line 5). In our example of Fig. 1 (excluding the red looping edges that have been filtered), the ordered depth-first traversal returns the paths ordered $[P_1, P_2, P_3, P_4, P_5]$ where $P_1 = \langle (s, a), (a, c), (c, e), (e, h), (h, k), (k, l), (l, end_2) \rangle$, $P_2 = \langle (s, a), (a, c), (c, f), (f, h), (h, k), (k, l), (l, end_2) \rangle$, $P_3 = \langle (s, a), (a, d), (d, i), (i, k), (k, l), (l, end_2) \rangle$, $P_4 = \langle (s, a), (a, d), (d, j), (j, k), (k, l), (l, end_2) \rangle$, and $P_5 = \langle (s, a), (a, b), (b, end_1) \rangle$. The ordered list of unique edges E_O extracted from these paths, respecting their order, is $E_O = [(s, a), (a, c), \ldots, (c, f), (f, h) \ldots]$.

Algorithm 1 BPMN Construction

1: **Input:** Acyclic Dependency Graph $DG_A = (V, E)$, parallel vertices $||$
2: **Output:** Generated BPMN Model $M = (N, G, S)$
3: Initialize M
4: Create a gateway reference $GR = null$
5: $E_O = \text{OrderedDF}(DG_A)$
6: **for all** $e \in E_O$ **do**
7: $source \leftarrow e.source$
8: $target \leftarrow e.target$
9: **if** $\nexists s', s'' \in S \mid s'.source = source \ \wedge \ s''.target = target$ **then**
10: $N \leftarrow N \cup \{source, target\}$
11: $S \leftarrow S \cup \{(source, target)\}$
12: **else if** $\exists s \in S \mid s.source = source \ \wedge \ target \notin N$ **then**
13: call $AddSplitGateways(DG_A, M, ||, source, target, GR)$
14: **else**
15: call $AddJoinGateways(M, source, target, GR)$

All the edges of P_1 are added in temporal order in the process model. Lines 9-11 add the source and target elements that are not already added to M. From P_2, the dependency (c, f) should be added, where c already exists and has a relation. Therefore, the split algorithm is called according to line 13 (detailed in Section 3.2, Figure 3a). This is followed by (f, h), where the join algorithm is invoked as specified in line 15 (detailed in Section 3.2, Figure 4a). From path P_3, for (a, d), the split algorithm is called (Figure 3b). This is followed by (d, i), which is added in line 13. Then, the join algorithm is called to add the relation between (i, k). For P_4, (d, j) is handled by calling the split algorithm, followed by the join algorithm for (j, k) (Figure 4b). Finally, from P_5, (a, b) is added by calling the split algorithm, and (b, end_1) is added directly (lines 9-11).

Split Operation The split operation in Algorithm 2 has two parts. The first adds a simple gateway (Figure 3a). The second, a recursive operation, nests gateways (Figure 3b).

(a) a simple split (b) a nested split

Fig. 3: Add split gateways to BPMN.

Figure 3a illustrates the addition of a new simple gateway with the dependency relation (c, f). As c is an existing source with a relation in the process

model (e), the split algorithm is applied. The first step involves checking the successor of c, which is e (lines 3-7). Since the successor is an activity, a split gateway is added where c is the incoming connection, and e and f are the outgoing connections (lines 9-10).

Algorithm 2 AddSplitGateways

1: **Input:** Acyclic Dependency Graph DG_A, Process Model M, parallel vertices $\|$, Entity *source*, Entity *target*, gateway reference GR
2: $target_M \leftarrow s.target \mid \exists s = (source, target_M) \in S$
3: **if** $type(target_M) \notin G$ **then**
4: $g_{new} \leftarrow Gateway(\text{“}+\text{”})$ if $\exists\{target_M, target\} \in \|$ else $Gateway(\text{“}\times\text{”})$
5: $G \leftarrow G \cup \{g_{new}\}$
6: $S \leftarrow S \setminus \{(source, g_{current}\}$
7: $S \leftarrow S \cup \{(source, g_{new}), (g_{new}, target), (g_{new}, target_M)\}$
8: **else**
9: $s \leftarrow (source, target_M)$
10: $AddNestingSplitGateway(DG_A, M, \|, GR, s, source, target)$

The advanced algorithm in Algorithm3 is utilized to add a split gateway after the source, which can result in nested gateways. The algorithm involves three main steps: (i) adding the target as a successor to the existing gateway (lines 4-6), (ii) inserting a new split gateway before the existing gateway (line 7-18), and (iii) adding a new gateway after the existing one using the nesting split algorithm (lines 20-24).

For example, in Figure 3b, the dependency relation is (a, b). The recursive algorithm begins by identifying the successor of a, which is the gateway (1). Next, all successors with the transition (3) are searched, resulting in the extracted list $\{c, d\}$. The relationship between the target b and the extracted list $\{c, d\}$ is then analyzed. Since there is no parallelism in $\|$ that includes (b, c) or (b, d), it is concluded that b is in an exclusive decision with these activities: $(b, c) \Rightarrow$ exclusive; $(b, d) \Rightarrow$ exclusive. Because this relation is exclusive and not of the same type as (3), a new gateway (7) is added before the existing gateway (lines 13-18).

Algorithm 3 AddRecursiveSplitGateway

Input: Acyclic Dependency Graph DG_A, Process Model M, parallel vertices $\|$, gateway reference GR, gateway type $gatewayType$, current relation s, Entity source $source$, Entity target $target$
$g_{current} \leftarrow s.target$
$succT_N \leftarrow SuccT_M^N(g_{current})$
if $succT_N = ||(target) \ \wedge \ type(g_{current}) = \text{“}+\text{”}$ **then**
  $S \leftarrow S \cup \{(g, target)\}$
  $GR \leftarrow g_{current}$
else if $succT_N = ||(target) \ \wedge \ type(g_{current}) = \text{“}\times\text{”}$ **then**
  $g_{new} \leftarrow Gateway(\text{“}+\text{”})$
  $G \leftarrow G \cup \{g_{new}\}$
  $S \leftarrow S \setminus \{(source, g_{current}\}$
  $S \leftarrow S \cup \{(source, g_{new}), (g_{new}, g_{current}), (g_{new}, target)\}$
  $GR \leftarrow g_{new}$
else if $succ_N \cap ||(target) \leftarrow \emptyset$ **then**
  $g_{new} \leftarrow Gateway(\text{“}\times\text{”})$
  $G \leftarrow G \cup g_{new}$
  $S \leftarrow S \setminus \{(source, g_{current}\}$
  $S \leftarrow S \cup \{(source, g_{new}), (g_{new}, g_{current}), (g_{new}, target)\}$
  $GR \leftarrow g_{new}$
else
  $succ_N \leftarrow Succ_M^N(g_{current})$
  for all $v \in succ_N$ **do**
    if $v \in G$ **then**
      $s \leftarrow (g_{current}, v)$
      $AddNestingSplitGateway(DG_A, M, \|, GR, s, source, target)$

Join Operation Similar to the split operation, the join operation presented in Algorithm 4 consists of two main parts: adding a simple join gateway and adding a nested join gateway.

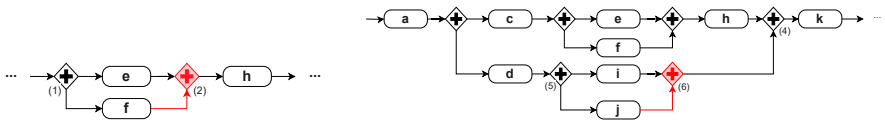
Algorithm 4 AddJoinGateways

Input: Process Model M, gateway reference GR, Entity $source$, Entity $target$
$source_M \leftarrow s.source \mid \exists s = (source_M, target) \in S$
if $type(source_M) \notin G$ **then**
  $g_{new} \leftarrow Gateway(type(GR))$
  $G \leftarrow G \cup \{g_{new}\}$
  $S \leftarrow S \setminus \{(source, g_{current}\}$
  $S \leftarrow S \cup \{(source, g_{new}), (source_M, g_{new}), (g_{new}, target)\}$
else
  $s \leftarrow (source_M, target)$
  $AddNestingJoinGateway(M, GR, s, source, target)$

Figure 4a illustrates the addition of a simple join. In this example, the dependency relation is (f, h). Since f is a source already added and h is a target already incorporated into the process model, the join algorithm is applied. The first step involves checking the predecessor of h, which is e (lines 3-7). As the predecessor is an activity, a join gateway is directly added after h. The incoming connections to this gateway are e and f, and the outgoing connection is h (lines 9-10). The type of the gateway matches the last gateway added into the BPMN.

The advanced algorithm in Algorithm 5 adds a join gateway when a gateway already exists before the target, potentially resulting in nested gateways. It involves three main steps: (i) adding the target as the predecessor of the existing gateway (lines 3-5), (ii) adding a new join gateway after the existing gateway (lines 6-31), and (iii) recalling the complex join algorithm to add a new gateway before the existing one (line 32).

In Figure 4b, the dependency relation is (j, k), with the last added split being (5). The algorithm checks for a backward path from the selected gateway (4) that intersects the split gateway (5). If all predecessors intersect the split gateway, the temporal relation between the gateways is validated. If they are of the same type, a new relation is added from the target to the selected gateway; otherwise, a new gateway is added after the selected gateway. If at least two predecessors have a relation, a new join gateway matching the split gateway's type is added after these predecessors and before the selected gateway. Otherwise, the nesting join algorithm is applied to all predecessor gateways. In this example, (4) has h and i as predecessors, with a backward path from i to (5). Therefore, a new join gateway of the same type as (5) is added after i (lines 23-30).



(a) a simple join (b) a nested join

Fig. 4: Add join gateways to BPMN.

Algorithm 5 AddNestingJoinGateway

Input: Process Model M, gateway reference GR, Entity $source$, Entity target $pred$, Entity $target$, current relation s
$pred_M \leftarrow Pred_M(pred)$
if $\forall v \in pred_M, \exists P_{GR,v}$ **then**
  if $type(pred) == typeOf(GR)$ **then**
    $S \leftarrow S \cup \{(source, pred)\}$
  else
    $g_{new} \leftarrow Gateway(type(GR))$
    $G \leftarrow G \cup \{g_{new}\}$
    $S \leftarrow S \setminus \{s\}$
    $S \leftarrow S \cup \{(source, g_{new}), (pred, g_{new}), (g_{new}, s.target)\}$
else if $\forall v \in pred_M, \nexists P_{GR,v}$ **then**
  $g_{new} \leftarrow Gateway(type(GR))$
  $G \leftarrow G \cup \{g_{new}\}$
  $S \leftarrow S \setminus \{s\}$
  $S \leftarrow S \cup \{(source, g_{new}), (pred, g_{new}), (g_{new}, s.target)\}$
else if $\|\forall v \in pred_M, \exists P_{GR,v}\| \geq 2$ **then**
  $g_{new} \leftarrow Gateway(type(GR))$
  $G \leftarrow G \cup \{g_{new}\}$
  for all $v \in pred_M$ **do**
    $S \leftarrow S \setminus \{(v, pred)\}$
    $S \leftarrow S \cup \{(v, g_{new})\}$
  $S \leftarrow \{(g_{new}, pred)\}$
else
  $pred_{new} \leftarrow Pred_M(v) | v \in pred_M$
  $s = (pred_{new}, pred) \in S$
  if $type(pred_{new}) \in G$ **then**
    $g_{new} \leftarrow Gateway(type(GR))$
    $G \leftarrow G \cup \{g_{new}\}$
    $S \leftarrow S \setminus \{s\}$
    $S \leftarrow S \cup \{(source, g_{new}), (pred_{new}, g_{new}), (g_{new}, pred)\}$
  else
    call $AddNestingJoinGateway(M, GR, source, pred_{new}, target, s)$
$GR \leftarrow null$

3.3 Loop Construction

After adding all BPMN entities, loops extracted (as described in Definition 6) should be included. Two merge algorithms are then applied to combine the loops based on their sources and targets, helping to detect the start and end of the block where the loop needs to be constructed.

The "Merge Loop by sources" algorithm 6 is designed to merge loops based on an acyclic dependency graph based on their structure and relationships, and based on the looping edge defined in 6. This merge groups loops that share the same source and whose targets have common successors (line 7-9). In the example, the looping edge $L = \{(f, f), (e, e), (f, e), (e, f)\}$, and the result from

this algorithm is $ML_s = \{(f, \{f, e\}), (e, \{e, f\})\}$, which is the input of the second algorithm 7.

Algorithm 6 Merge Loop by sources

1: **Input:** Acyclic Dependency Graph DG_A, looping edge L
2: **Output:** Merged loops ML_s
3: create a list ML_s
4: **for all** $l \in L$ **do**
5: create a set $targets \leftarrow \{l.target\}$
6: **for all** $l' \in L \setminus \{l\}$ **do**
7: **if** $l.source == l'.source \ \wedge \ \text{Succ}_{DG_A}(l.target) == \text{Succ}_{DG_A}(l'.target)$ **then**
8: $targets \leftarrow targets \cup \{l'.target\}$
9: $L \leftarrow L \setminus \{l'\}$
10: $ML_s \leftarrow ML_s \cup (l.source, targets)$
11: $L \leftarrow L \setminus \{l\}$

The "Merge Loop" by targets algorithm 7 is designed to consolidate loops that have already been grouped based on their sources. The goal is to further merge these loops by examining their targets and associated predecessors. This merge ML, is derived from ML_s by combining sources if they have common predecessors (lines 7-9). In the example, the extracted $ML_s = \{(f, \{f, e\}), (e, \{e, f\})\}$, and the result from this algorithm is $ML = \{(\{f, e\}, \{f, e\})\}$. This indicates the existence of a loop for the entire block of $\{e, f\}$.

Algorithm 7 Merge Loop

1: **Input:** Acyclic Dependency Graph DG_A, Loop merged by sources ML_s
2: **Output:** Merged loops ML
3: create a list ML
4: **for all** $l \in ML$ **do**
5: create a set $sources \leftarrow \{l.source\}$
6: **for all** $l' \in ML_s \setminus \{l\}$ **do**
7: **if** $l.target == l'.target \ \wedge \ Pred_{DG_A}(l.source) == Pred_{DG_A}(l'.source)$ **then**
8: $sources \leftarrow sources \cup \{l'.source\}$
9: $ML_s \leftarrow ML_s \setminus \{l'\}$
10: $ML \leftarrow ML \cup (sources, l.target)$
11: $ML_s \leftarrow ML_s \setminus \{l\}$

The next step in the loop construction is the block construction. During the generation of the BPMN, it is not possible to establish all relationships because the loop relation is removed, resulting in some incomplete blocks. For example, after removing the entities in red in Figure 5, the result is contracted before adding the loop. In the loop block construction algorithm (Algorithm 8), the

goal is to complete the loop block (gateway (3)) before adding the loops as described in algorithm 9.

Algorithm 8 Loop Block Construction

1: **Input:** Acyclic dependency graph DG_A, Process Model $M = (N, G, S)$, Merged loops ML, parallel vertices $\|$
2: **for all** $l = (sources, targets) \in ML$ **do**
3: **if** $\|sources\| \neq 1 \ \wedge \ \nexists (u, v) \in S \mid (v \in N \ \wedge \ u \in sources)$ **then**
4: $g_{split} \leftarrow LCA_M(sources)$
5: $G \leftarrow G \cup \{g_{join}\} \mid type(g_{join}) = type(g_{split})$
6: $S \leftarrow S \cup \{(s, g_{join}\} \mid s \in sources$
7: **if** $\|targets\| \neq 1 \ \wedge \ \nexists (v, u) \in S \mid (v \in N \ \wedge \ u \in targets)$ **then**
8: $N \leftarrow N \cup \{temp\}$
9: **for all** $t \in targets$ **do**
10: call $AddSplitGateways(DG_A, M, \|, temp, t, \emptyset)$
11: $N \leftarrow N \setminus \{temp\}$

Since in the example used in the paper, the blocks of e, f are completed where the source block is gateway (1) and the target block is gateway (2) (cf. Figure 6), we add other example (cf. Figure 5) to well explain the algorithm, where only one loop exists, which is $(\{e, f\}, \{b\})$. Since there are two sources, $\{e, f\}$, it is necessary to validate the completion of their block (line 3). If the block is completed, no action is required; otherwise, the block should be constructed based on the split. To accomplish this, by applying the lowest common ancestor algorithm [3] on the process model, the split gateway can be identified as the last ancestor e and f (line 4). Subsequently, a new join gateway should be added with the same type of split(lines 5-6).

Next, the targets are checked to determine if they construct all the blocks. If more than one target exists, the existence of gateways before the targets is validated (line 7). If gateways do not exist, a temporary activity is added to construct this block by adding connections and applying the split operation from the temporary activity to the targets (line 8-10). Finally, this activity is removed (line 11), and the process continues to the final algorithms to add loops to the process model 9.

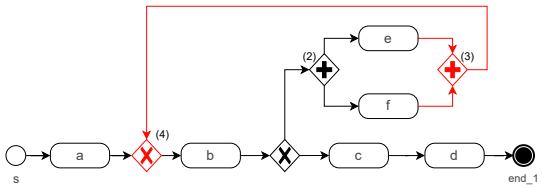


Fig. 5: Complete a loop block and add it to the BPMN model.

The final step involves adding the loop between the source and target blocks presented in Algorithm 9. A block is considered an activity if there is one element for the source or target, and it is considered a gateway if there is more than one source or target. The process iterates through all the Merged Loop results, leading to four scenarios: (i) If both source and target blocks have a sequence flow to other entities, a new gateway is added after the source block and another before the target block. These two gateways are then connected (line 7). (ii) If the source block has a sequence flow but the target block does not, a new gateway is added after the source block, followed by a connection from the new gateway to the target block (line 9). (iii) If the target block has a sequence flow but the source block does not, a new gateway is added before the target block, followed by a connection from the source block to the gateway (line 12). (iv) If neither the source nor the target block has a sequence flow, a simple connection is created between these blocks (line 14).

Algorithm 9 Loop Operation

1: **Input:** Process Model M, merge loops ML
2: **for all** $l \in ML$ **do**
3: $sourceBlock \leftarrow \bigcap\{Succ_M(s) | s \in l_s\}$ if $|l_s| > 1$ else l_s
4: $targetBlock \leftarrow LCA_M(l_t)$ if $|l_t| > 1$ else l_t
5: **if** $\exists(sourceBlock, s) \in S$ **then**
6: **if** $\exists(s, targetBlock) \in S$ **then**
7: $AddTwoLoopGateways(sourceBlock, targetBlock)$
8: **else**
9: $AddLoopAfterSourceBlock(sourceBlock, targetBlock)$
10: **else**
11: **if** $\exists(s, targetBlock) \in S$ **then**
12: $AddLoopBeforeTargetBlock(sourceBlock, targetBlock)$
13: **else**
14: $S \leftarrow S \cup \{(sourceBlock, targetBlock)\}$

In the paper example, Figure 6 represents the loop operation where the $\{e, f\}$ block is in a loop, meaning this block is both the source and the target, with a gateway as both the source and target blocks. The source block is gateway (2), and the target block is gateway (1). These blocks are connected; therefore, new gateways are added, one after the source (8) and another before the target (9), and a connection from (8) to (9) is added (line 14 in the algorithm).

In other case (Figure 5), after constructing the loop block (adding the gateway (3)), the source block is disconnected, but b which is the target block is connected to other entities. Therefore, a new # gateway (4) is added before b, with a connection from (3) to (4) (lines 11-12 in the algorithm).

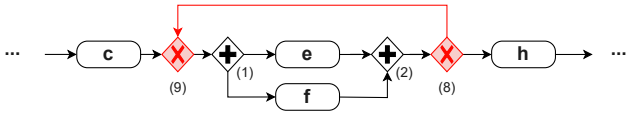


Fig. 6: Add a loop to the process model.

References

1. Augusto, A., Conforti, R., Dumas, M., Rosa, M.L., Bruno, G.: Automated discovery of structured process models from event logs: The discover-and-structure approach. Data Knowl. Eng. **117**, 373–392 (2018)
2. Augusto, A., Conforti, R., Dumas, M., Rosa, M.L., Maggi, F.M., Marrella, A., Mecella, M., Soo, A.: Automated discovery of process models from event logs: Review and benchmark. IEEE Trans. Knowl. Data Eng. **31**(4), 686–705 (2019)
3. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. J. Algorithms **57**(2), 75–94 (2005)