

Refinement of BPMN 2.0 Inclusive and Complex Gateway Activation Concept towards Process Engine

Jan Kubovy* and Josef Küng

Institute for Application Oriented Knowledge Processing
Johannes Kepler University in Linz
jkubovy@faw.jku.at, jkueng@faw.jku.at

Abstract. This paper presents a possible refinement of Business Process Model and Notation (BPMN) Gateway activation concept for non-event-based gateways, introduced in. The core refinement is the concrete formal definition of upstream token concept and calculation of the enabledness of an inclusive gateways (or also Or-Join) using modified Dijkstra's algorithm. The introduced algorithm for upstream token calculation considers also situations where two or more gateways are mutually dependent.

Keywords: BPMN, inclusive gateway, complex gateway, activation, process engine, ASM method, refinement

1 Introduction

The Business Process Model and Notation (BPMN), currently in version 2.0 [6], is a well known and popular process modeling standard by the Object Management Group (OMG). The one, non-primitive, *Or-Join* activation concept of converging inclusive and complex gateways present in [6] is explored in Section 2. We use the BPMN Core Modeling Concepts [1] as a starting point for our refinements and define an algorithm for the computation of upstream tokens [1] for a particular gateway. We also demonstrate a proper function of the introduced algorithm in cases where two or more gateways mutually depend on each other (also called *vicious circles*), except some special cases, e.g., symmetric *vicious circle* shown in Figure 1.

For formalization the Abstract State Machine (ASM) method [2] was chosen, since this method is also used in the original work [1; 3] and this paper is also part of a long-term ongoing work, which aims to formalize and enhance the BPMN 2.0 standard and is also using the ASM method.

In Section 2 we approach the upstream token concept using graph coloring algorithm, describe the work of this algorithm and show the enabledness test for inclusive and complex gateways using the proposed algorithm. Next, in Section 3

* This work was supported in part by the Austrian Science Fund (FWF) under grant no. TRP 223-N23.

we demonstrate the correctness of the algorithm, even in cases when some gateways may be mutually dependent on each other and identify exceptions when the proposed algorithm will not work.

2 Upstream Token

An upstream token of a flow node is a token in any sequence flow if there is a path starting with that sequence flow and reaching that flow node and if such sequence flow is not directly connected to that flow node. Upstream tokens are needed for flow nodes such as inclusive and complex gateways. This is defined using inhibiting and anti-inhibiting paths [7] as a token which has an inhibiting path but no anti-inhibiting path to the corresponding flow node. Such upstream tokens are used as activation condition of an inclusive gateway [6, Table 13.3], or as a reset condition of a complex gateway [6, Table 13.5] and also as an $\text{UpstreamToken} \rightarrow \text{Set}$ [1].

In this section we propose an algorithm as a refinement of the $\text{UpstreamToken} \rightarrow \text{Set}$ [1; 3], which will for any given flow node in a process diagram identify all relevant sequence flows and color them based on their directly incoming sequence flows to the chosen flow node. This computation and coloring is made on an oriented cyclic graph represented by an ordered pair $\mathcal{G}(\mathcal{N}, \mathcal{E})$. The $\mathcal{G}(\mathcal{N}, \mathcal{E})$ can be obtained by converting all flow nodes of the process diagram to nodes \mathcal{N} of \mathcal{G} and all sequence flows to oriented edges \mathcal{E} with opposite orientation than the original sequence flows of the process diagram.

Definition 1 (Graph transformation). *Let $\mathcal{G}_d(\mathcal{N}, \mathcal{E}_{sf})$ be a process diagram, \mathcal{N} be a set of all flow nodes in \mathcal{G}_d and \mathcal{E}_{sf} be set of all sequence flows in \mathcal{G}_d . We construct an oriented graph $\mathcal{G}(\mathcal{N}, \mathcal{E})$ where $\forall e \in \mathcal{E} \exists ! e_{sf} \in \mathcal{E}_{sf} (e = \{(n_0, n_1) \in \mathcal{N}\} \wedge e_{sf} = \{(n_1, n_0) \in \mathcal{N}\})$.*

The next step is to color the edges of the graph \mathcal{G} for every flow node that requires such coloring for its activation, e.g., inclusive or complex gateways, using the modified Dijkstra's algorithm [4] shown in Listing 1.

Definition 2 (Colored graph). *A colored graph is a tuple $\mathcal{G}_\kappa = (\mathcal{N}, \mathcal{E}, \kappa, \delta, \mathcal{C}_\mathcal{N}, \mathcal{C}_\mathcal{E})$, where:*

- \mathcal{N} is a set of nodes same as in \mathcal{G}_d or \mathcal{G} ,
- $\mathcal{E} : \mathcal{N} \times \mathcal{N}$ is a set of edges after transforming \mathcal{G}_d to \mathcal{G} , where the 1st parameter represents a node the edge is outgoing from and the 2nd parameter represents the a node the edge is incoming into¹,
- $\kappa : \mathcal{N} \times \mathcal{N} \rightarrow \text{Boolean}$ is a function capturing closeness of the node related to a node a color calculation is done for,
- $\delta : \mathcal{N} \times \mathcal{N} \rightarrow \text{Integer}$ is a function capturing the distance between two nodes,

¹ Note that the direction of edges $e \in \mathcal{E}$ is opposite to the direction of sequence flows $s \in \mathcal{E}_{sf}$

- $\mathcal{C}_N : \mathcal{N} \times \mathcal{N} \rightarrow \text{Color}$ is a function capturing a node color related to a node a color calculation is done for,
- $\mathcal{C}_E : \mathcal{N} \times \mathcal{E} \rightarrow \text{Color}$ is a function capturing a edge color related to a node a color calculation is done for.

The 1st node parameter of functions κ , δ , \mathcal{C}_N and \mathcal{C}_E represents the node a color calculation is done for. The 2nd parameter of those functions represents the relevant node or edge on the incoming inhibiting or anti-inhibiting paths of the node passed to those functions in the 1st parameter.

2.1 Work of the algorithm

The rule **ColorProcessGraph** : N shown in Listing 1 finds a shortest path from the **root** node of \mathcal{G}_κ (representing an inclusive or complex gateway in \mathcal{G}_d) given as the only parameter to any reachable node in the directed graph \mathcal{G}_κ in the way the original Dijkstra's algorithm [4] was designed. Additionally the algorithm colors visited nodes and tested edges, even those tested edges which are not further used for the search (i.e. incoming edges of closed nodes $\mathbf{n} \in \mathcal{N}$ indicated by $\kappa(\mathbf{root}, \mathbf{n}) = \top$). The algorithm starts with coloring each of the directly outgoing edges from the **root** node with a distinct color. Those colors are then populated through \mathcal{G}_κ till its leafs (usually representing start events in \mathcal{G}_d). If the algorithm visits a node, which has more than one incoming edge in \mathcal{G}_κ (representing a splitting gateway in \mathcal{G}_d) for the first time (i.e. such node $\mathbf{n} \in \mathcal{N}$ is not closed yet: $\kappa(\mathbf{root}, \mathbf{n}) = \perp$) it will additionally color this node with a distinct **limpid** color. Such **limpid** color will be further populated with the other colors such node is colored with. A color calculation for a **root** node (represented by a run of rule **ColorProcessGraph** : N) will never color two nodes in \mathcal{G}_κ having more than one incoming edge with the same **limpid** color. A **limpid** color is an indicator for other alternative paths of the concrete node to populate their own non-**limpid** colors. Every time such node will be tested by the algorithm again (i.e. such node is already closed) all non-**limpid** colors of the tested alternative edge will be populated to all nodes and edges in \mathcal{G}_κ which are colored with the **limpid** color of the tested node.

All **limpid** colors are only visible locally inside the run of the **ColorProcessGraph**. The resulting set of colors obtained from both color functions (\mathcal{C}_N , \mathcal{C}_E) will not contain any **limpid** colors.

2.2 Enableness test

The enableness of an inclusive gateway is defined as:

Theorem 1 (Enableness of inclusive gateway). *The Inclusive Gateway is enabled if [1; 6; 7]:*

- At least one incoming sequence flow has at least one token and
- there is no upstream token, meaning:

Listing 1: rule ColorProcessGraph : N

```

rule ColorProcessGraph(root)
   $\forall n \in \mathcal{N}$  do
     $\kappa(\text{root}, n) \leftarrow \perp$ 
     $\delta(\text{root}, n) \leftarrow \infty$ 
     $\mathcal{C}_{\mathcal{N}}(\text{root}, n) \leftarrow \emptyset$ 

   $\forall e \in \mathcal{E}$  do  $\mathcal{C}_{\mathcal{E}}(\text{root}, e) \leftarrow \emptyset$ 
   $\delta(\text{root}, \text{root}) \leftarrow 0$ ;
  set  $\leftarrow \{\text{root}\}$ 

  while |set| > 0 do
    min  $\leftarrow \infty$ 
    node  $\leftarrow$  undef

     $\forall n \in \text{set}$  do
      if  $\delta(\text{root}, n) < \text{min}$  then
        min  $\leftarrow \delta(\text{root}, n)$ 
        node  $\leftarrow n$ 

    remove node from set
     $\kappa(\text{root}, \text{node}) \leftarrow \top$ 

     $\forall \text{next} \in \mathcal{N}$  ( $\text{edge}(\text{node}, \text{next}) \in \mathcal{E} \wedge \text{next} \neq \text{root}$ ) do

      if  $\delta(\text{root}, \text{node}) + 1 < \delta(\text{root}, \text{next})$ 
         $\wedge \neg \kappa(\text{root}, \text{next})$  then

         $\delta(\text{root}, \text{next}) \leftarrow \delta(\text{root}, \text{node}) + 1$ 
        add next to set

      if  $\exists n \in \mathcal{N}$  ( $\text{edge}(n, \text{next}) \in \mathcal{E} \wedge n \neq \text{node}$ ) then
        add nextLimpid to  $\mathcal{C}_{\mathcal{N}}(\text{desc})$ 

    if node = root then
      color  $\leftarrow$  nextColor
      add color to  $\mathcal{C}_{\mathcal{N}}(\text{next})$ 
      add color to  $\mathcal{C}_{\mathcal{E}}(\text{root}, \text{edge}(\text{node}, \text{next}))$ 
    else
       $\forall \text{color} \in \mathcal{C}_{\mathcal{N}}(\text{node})$  do
        if ( $\nexists c \in \mathcal{C}_{\mathcal{N}}(\text{next})$  ( $\text{isLimpid}(c)$ )
           $\vee \neg \text{isLimpid}(\text{color})$ ) then
          add color to  $\mathcal{C}_{\mathcal{N}}(\text{next})$ 
          add color to  $\mathcal{C}_{\mathcal{E}}(\text{root}, \text{edge}(\text{node}, \text{next}))$ 

       $\forall \text{limpid} \in \mathcal{C}_{\mathcal{N}}(\text{next})$  ( $\text{isLimpid}(\text{limpid})$ ) do
         $\forall n \in \mathcal{N}$  ( $\text{limpid} \in \mathcal{C}_{\mathcal{N}}(n)$ ) add color to  $\mathcal{C}_{\mathcal{N}}(n)$ 
         $\forall e \in \mathcal{E}$  ( $\text{limpid} \in \mathcal{C}_{\mathcal{E}}(\text{root}, e)$ ) do
          add color to  $\mathcal{C}_{\mathcal{E}}(\text{root}, e)$ 

```

- a token that has an inhibiting path,
- but no anti-inhibiting path

Based on the colored graph \mathcal{G}_κ an upstream token for a concrete flow node is defined in Definition 3 and shown in Listing 2.

Definition 3 (Upstream token). A flow node represented by $n \in \mathcal{N}$ in \mathcal{G}_κ has a upstream token if:

- there is a token in a relevant (for that node n colored with one or more colors) edge $e \in \mathcal{E}$, which has a token
- and there is no directly outgoing edge from node n which has a token and is colored with one of the colors e is also colored with

Listing 2: UpstreamToken : $\mathcal{N} \rightarrow \text{Set}$

```

UpstreamToken(node) =
  ignore  $\leftarrow \{ c \in \mathcal{C}_\mathcal{E}(\text{node}, e) \mid \text{source}(e) = \text{node} \wedge \text{token}(e) \}$ 
  relevant  $\leftarrow \{ e \in \mathcal{E} \mid \mathcal{C}_\mathcal{E}(\text{node}, e) \neq \emptyset \}$ 

  return  $\{ t \mid t = \text{token}(e) \text{ with } e \in \text{relevant} \wedge (\mathcal{C}_\mathcal{E}(\text{node}, e) \setminus \text{ignore}) \neq \emptyset \}$ 

```

3 Cyclic workflow graphs

In this section we show that the algorithm proposed in Section 2 works for use-cases including cyclic workflow graphs except some special cases, for which a reasonable semantics is not clear and can be sorted out by static analysis (e.g. Figure 1) [7].

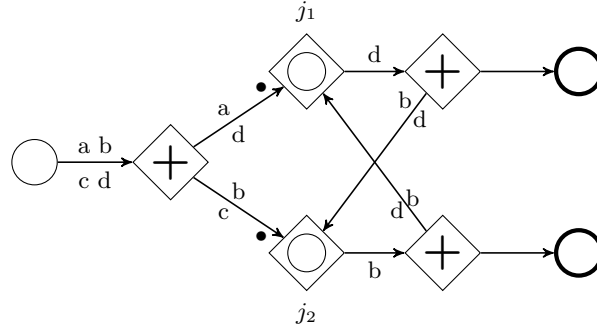


Fig. 1: A symmetric *vicious circle* [7]

The process depicted in Figure 1 was colored by `ColorProcessGraph` during the deployment of the process and the resulting coloring is indicated by small latin letters beside the sequence flows they color. Colors **a** and **b** are relevant for the inclusive gateway j_1 and colors **c** and **d** are relevant for the inclusive gateway j_2 . Tokens are represented by a dot “•” next to the sequence flow they are contained in.

In the current state depicted in Figure 1 we can see that for j_1 a directly incoming sequence flow colored with the color **a** contains a token and therefore all other sequence flows colored with the color **a** can be ignored. On the other hand, the second directly incoming sequence flow to j_1 colored with the color **b** does not contain a token but there is a sequence flow in the process colored with **b** and not with **a** containing a token which makes such token be an upstream token of the inclusive gateway j_1 . The gateway j_1 has to wait for that token, hence for the activation of the gateway j_2 .

Similarly for the inclusive gateway j_2 and its colors **c** and **d** where sequence flows colored with the color **c** can be ignored for the activation of j_2 and sequence flows containing a token and colored with the color **d** and not with the color **c** block the activation of j_2 hence j_2 has to wait for the activation of j_1 . This symmetric dependency is considered as design error with unclear underlying semantics and may be detected using static analysis.

3.1 Well-structured processes

The coloring of the process for each inclusive or complex gateway simulates the concept of inhibiting and anti-inhibiting paths and so coincide with the *Q-semantics* [7].

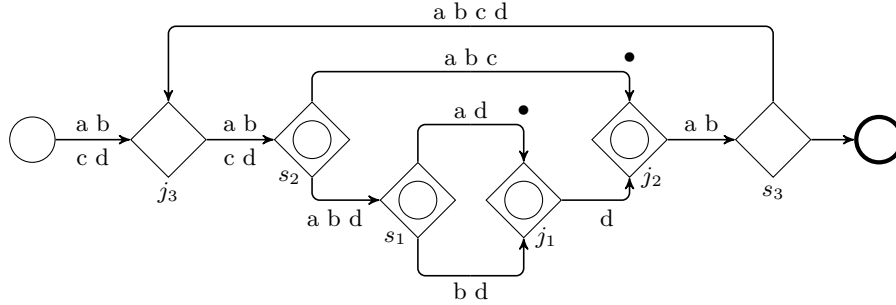


Fig. 2: A *vicious circle* in well-structured process [7]

In the example of well-structured process depicted in Figure 2 the coloring for both merging inclusive gateways (j_1 , j_2) is shown. Similarly as in Figure 1 the colors **a** and **b** are relevant for the gateway j_1 and the colors **c** and **d** are relevant

for the gateway j_2 . It can be observed that the gateway j_1 blocks activation of gateway j_2 but not vice-versa.

3.2 Non-separable processes

To complete our examples we also show that the proposed algorithm works correctly on non-separable process workflows. The example shown in Figure 3 [7] is showing process colored with colors **a** and **b** relevant to the only inclusive join j_2 . For the sake of brevity we do not color incoming paths of inclusive gateways with only one incoming sequence flow (splits s_1 and s_3).

It can be observed that in all of those cases no two inclusive gateways are mutually blocking each other or themselves.

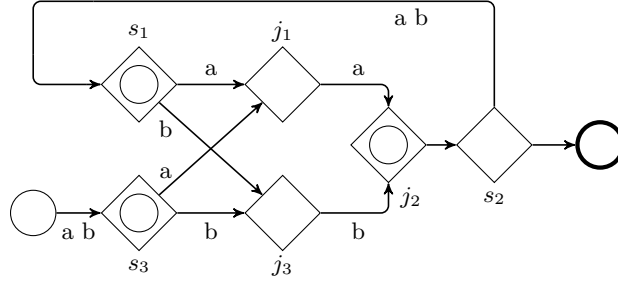


Fig. 3: A non-separable processes [7]

4 Summary

We have presented a possible refinement of the non-primitive activation concept of inclusive and complex gateways of a BPMN ground model [1; 3]. Our solution is based on a modified Dijkstra's algorithm [4] which colors the sequence flows of a process. Such coloring may be computed during deployment of a process into a process engine for every flow node, which may need it and so speedup the execution of instances of such process. Emphasis was placed on brevity, simplicity and reusability of the algorithm shown in Listing 1.

We demonstrated correct work of the algorithm and also identified cases, where the proposed algorithm will not work, such as *Symmetric Vicious Circles* [7]. We agree that such cases may be considered as design errors where the underlying semantics is unclear and may be ruled out during development or deployment of an process by static analysis [5].

References

- [1] Börger E, Sörensen O (2011) BPMN core modeling concepts: Inheritance-based execution semantics. In: Embley DW, Thalheim B (eds) Handbook of Conceptual Modeling: Theory, Practice and Research Challenges, Springer-Verlag, chap 9
- [2] Börger E, Stärk RF (2003) Abstract State Machines - A Method for High-Level System Design and Analysis. Springer-Verlag
- [3] Börger E, Thalheim B (2008) A method for verifiable and validatable business process modeling. Advances in Software Engineering, LNCS 5316:59–115
- [4] Dijkstra EW (1959) A note on two problems in connexion with graphs. In: Numerische Mathematik, vol 1, Springer, p 269–271
- [5] Dumas M, Grosskopf A, Hettel T, Wynn M (2007) Semantics of standard process models with or-joins. In: Proceedings of the 2007 OTM Confederated international conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part I, Springer-Verlag, Berlin, Heidelberg, OTM’07, pp 41–58
- [6] Object Management Group (OMG) (2011) Business process model and notation (BPMN) 2.0. www.omg.org/spec/BPMN/2.0
- [7] Völzer H (2010) A new semantics for the inclusive converging gateway in safe processes. In: Proceedings of the 8th international conference on Business process management, Springer-Verlag, Berlin, Heidelberg, BPM’10, pp 294–309