Formal Semantics and Implementation of BPMN 2.0 Inclusive Gateways

David Raymond Christiansen, Marco Carbone, and Thomas Hildebrandt*

IT University of Copenhagen,
Rued Langgaards Vej 7,
2300 Copenhagen, Denmark

Abstract. We present the first direct formalization of the semantics of inclusive gateways as described in the Business Process Modeling Notation (BPMN) 2.0 Beta 1 specification. The formal semantics is given for a minimal subset of BPMN 2.0 containing just the inclusive and exclusive gateways and the start and stop events. By focusing on this subset we achieve a simple graph model that highlights the particular non-local features of the inclusive gateway semantics. We sketch two ways of implementing the semantics using algorithms based on incrementally updated data structures and also discuss distributed communication-based implementations of the two algorithms.

1 Introduction

Business Process Modeling Notation (BPMN), a standardized notation for representing processes within organizations, is soon to be released in a major new revision. According to the draft BPMN 2.0 specification,

> The primary goal of BPMN is to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes. Thus, BPMN creates a standardized bridge for the gap between the business process design and process implementation [5, p. 1].

Because BPMN seeks to serve as a kind of universal communication tool for business processes, it is vitally important that all parties agree on the meaning of a BPMN diagram. The BPMN 2.0 specification provides a rather detailed, but still only informal description of its semantics [5, p. 389].

The so-called *inclusive gateways* of BPMN seem particularly challenging to provide semantics for, since a non-trivial (and non-local) backwards search in

* This research is supported by the Danish Research Agency through the FIRST Graduate School (www.first.dk), the Jingling Genie project (http://sites.google.com/site/jinglinggenie/) and the Computer Supported Mobile Adaptive Business Processes project (grant #274-06-0415, www.cosmobiz.dk).

the flow graph is included in the specification of their semantics. This is similar to the OR-joins of YAWL and EPC which have been the subject of several papers aiming to clarify their non-local semantics [6,1,3]. In this work, we focus on a small subset of BPMN 2.0, called BPMN$_{\mathrm{inc}}$ (BPMN inclusive) which just includes the primitives needed to illustrate the complexity of inclusive gateways and formalize their semantics in a way that can be generalized to full BPMN 2.0.

BPMN$_{\mathrm{inc}}$ is defined in Section 2. A formal semantics is provided for BPMN$_{\mathrm{inc}}$ in Section3, followed by a discussion of how it can be implemented efficiently and in a distributed manner. Related work is discussed in Section 4.

2 BPMN$_{\mathrm{inc}}$ and Its Informal Semantics

BPMN process diagrams contain a large number of different graphical elements. Broadly speaking, there are four classes of elements that are of interest when designing semantics:

Sequence Flow describes the order in which various parts of the process occur.
Events represent things that can happen during a process, such as a message being sent or a timer.
Activities represent work performed by a company, and can either be atomic (in which case they are called *tasks*) or they can represent another process diagram.
Gateways provide flow control within a process diagram [5, p. 21].

With the exception of Sequence Flow, there are multiple variations of each of the above elements, providing for different kinds of flow control and allowing representation of different kinds of business activities.

For purposes of this paper, only a small subset of BPMN that is sufficient to illustrate certain difficult properties will be used. BPMN$_{\mathrm{inc}}$, the subset, contains:

- Sequence flow
- Exclusive gateways
- Inclusive gateways
- Start events
- End events

Above, a gateway is exclusive when it behaves as an exclusive conditional while it is inclusive when its activation depends on further conditions on the incoming flows as well as allowing for multiple parallel outcomes. Start and end events model initiation and termination of BPMN processes. In the following, certain aspects of BPMN$_{\mathrm{inc}}$ will not be defined in full detail. For example, the conditions that determine which outgoing sequence flow should be chosen after a gateway is activated are simply assumed to exist and be subject to evaluation, giving either a true or false result, while the mechanism of this evaluation remains unspecified.

Activities are not included in BPMN$_{\mathrm{inc}}$, as their possible effects are not modeled. For our purposes, an activity will be equivalent to an exclusive gateway with a single incoming and a single (default) outgoing sequence flow.

Finally, note that the parallel split from BPMN where the flow of execution is split into two parallel flows can just be seen as a special case of inclusive gateways, where all outgoing conditions evaluate to true and the default flow connects to the end event. Parallel join is not straightforwardly encodable using inclusive gateways, but it is however straightforward to include in our semantics.

2.1 Sequence Flow and Tokens

Sequence flow represents the order in which the execution of a BPMN process occurs. [5, p. 21] It is represented as an arrow.

We follow the BPMN 2.0 specification in representing the execution state of a process with tokens on sequence flow[5], which is represented graphically as a small solid black circle placed next to the sequence flow. When the sequence flow before some element of a process diagram receives a token, then the element is activated in some way dependent on the precise type of element. The control-flow semantics of the gates are then given by the tokens required on their incoming sequence flow and the tokens produced on the outgoing sequence flow. Note that a sequence flow may have more than one token.

2.2 Exclusive Gateways

The semantics of exclusive gateways are quite uncomplicated. When a token arrives on any incoming sequence flow, it evaluates the conditions on the outgoing sequence flow until it finds one that returns true. It then places a token on that sequence flow and stops evaluating conditions. If no condition evaluates to true, then the sequence flow marked as default receives the token. [5, p. 401]

In BPMN$_{\text{inc}}$, every exclusive gateway must have a default outgoing sequence flow, which is indicated by placing a slash through the line immediately next to the gateway. In the exclusive gateway in Fig. 1, when a token arrives on *any one*

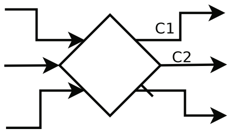

Fig. 1. An exclusive gateway

of the incoming sequence flow on the left, C_1 is evaluated. If it is true, a token is emitted on the sequence flow that C_1 is associated with. If it is not true, then C_2 is evaluated, and if it is true, then a token is emitted on the sequence flow attached to C_2. Finally, if no condition evaluates to true, then a token is emitted on the default sequence flow.

2.3 Inclusive Gateways

The BPMN inclusive gateway is problematic because it requires a search to be made for tokens upstream of it.[1]. To quote the specification:

> The Inclusive Gateway is activated if
> – At least one incoming sequence flow has at least one *Token* and

[1] Note that other gateways, such as the complex gateway, share this behavior.

– for each empty incoming sequence flow, there is no *Token* in the graph anywhere upstream of this sequence flow, i.e., there is no directed path (formed by Sequence Flow) from a Token to this sequence flow unless
  - the path visits the inclusive gateway or
  - the path visits a node that has a directed path to a non-empty incoming sequence flow of the inclusive gateway. [5, p. 401]

$\mathrm{BPMN}_{\mathrm{inc}}$ includes all this behavior. Note that the specification is independent of which other events, activities or gateways are allowed in the diagram. Consequently, our formal semantics for inclusive gateways can be straightforwardly extended to any superset.

According to the above definition, in the process shown in Fig. 2, if there is a token on edges A and B, then the rightmost inclusive gateway is allowed to activate because there is a directed path from the topmost exclusive gateway to A. However, if the topmost exclusive gateway fires first and sends a token on its top sequence flow, then the gateway labeled α must complete, depositing a token at C, before the rightmost inclusive gateway can fire. The example thus also illustrates that the BPMN 2.0 specification of the behavior of inclusive gateways may lead to race conditions.

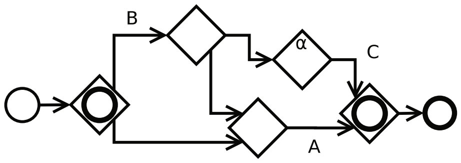

Fig. 2. Process with inclusive gateways

The other exception in the search defined above relates to gateways in cycles. Fig. 3 shows one such process. The inclusive gateway can fire if there is a token on the first sequence flow (immediately after the start event) even though the top incoming sequence flow does not have a tokens.

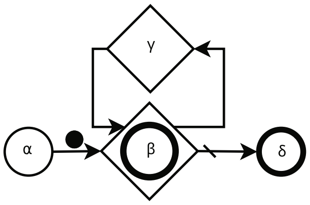

Fig. 3. A simple process with a loop

A simple approach to a token-based semantics, in which gates determine whether to fire based only upon which of the incoming sequence flow contain tokens, is clearly unable to implement the requirements of the specification without resorting to some kind of global information about the flow of control in the rest of the process.

After the inclusive gateway fires, the conditions on all outgoing sequence flow are evaluated. If any of them are true, then *all* sequence flow whose conditions are true receive tokens. If none of them are true, then the default outgoing sequence flow receives a token.

2.4 Start and End Events

A start event is responsible for emitting the first token that starts the process. BPMN has relatively complex semantics for running multiple parallel instances of subprocesses. Because BPMN$_{\text{inc}}$ has no concept of subprocesses, start events need only emit the first token.

Likewise, no specific semantics for the end of a subprocess are necessary in BPMN$_{\text{inc}}$. Therefore, end events simply consume tokens, and the process is complete when there are no tokens remaining.

3 Formal Semantics for BPMN$_{\text{inc}}$

In this section, we define a formal semantics for BPMN$_{\text{inc}}$. Unlike the other possible approaches to providing a formal semantics for BPMN (see Section 4), the semantics provided here does not attempt to translate the BPMN$_{\text{inc}}$ process to another formal system. Instead, semantics is given operationally directly as a token-based semantics for the BPMN$_{\text{inc}}$ process graph.

Throughout this section, the process in Fig. 3 will be used to demonstrate the formalization. The letters α, β, γ and δ are not a part of BPMN$_{\text{inc}}$, but they provide a means of referring to individual gateways.

3.1 BPMN$_{\text{inc}}$ Process Graphs

Below, we give the formal definition of BPMN$_{\text{inc}}$ process graphs:

Definition 1 (BPMN$_{\text{inc}}$ **Process).** [2] *A process $\mathcal{P}$ is a tuple $(N, \mathcal{L}, S, \preceq, C, M)$ such that:*

- *N is a set of nodes;*
- *the labeling $\mathcal{L} : N \to \{Excl, Incl, Start, End\}$ maps nodes in N to either Excl (BPMN exclusive gateways), Incl (BPMN inclusive gateways), Start (start events), and End (end events);*
- *$S \subseteq N \times N$ is a set of sequence flows such that $(n, n') \in S$ implies $\mathcal{L}(n) \neq End$ and $\mathcal{L}(n') \neq Start$;*

[2] This definition is somewhat analogous to that of Petri nets[7], where N corresponds to transitions, and S corresponds to places, except that the sequence flow connects exactly one node to another node.

- $\preceq \subseteq S \times S$ *is an ordering relation over* S *such that*
  - *for all* n *if* $(n, n') \in S$ *and* $(n, n'') \in S$ *then either* $(n, n') \preceq (n, n'')$ *or* $(n, n'') \preceq (n, n')$ *(*$\preceq$ *is total over the outgoing sequence flows for any given gateway);*
  - *and* $(n_1, n_2) \preceq (n_3, n_4)$ *implies* $n_1 = n_3$ *(*$\preceq$ *does not relate sequence flows with different source gateways);*
- $C : S \rightarrow Cond$ *is a partial map (defined on the outgoing sequence flows of inclusive and exclusive gateways) to (an assumed set of) conditions such that the maximal (w.r.t. the ordering* $\preceq$*) outgoing flow for each gateway always has the condition true; and*
- $M : S \rightarrow \mathbb{N}$ *is a marking of* S *with tokens.*

Above, $\preceq$ is an ordering relation defined such that the outgoing sequence flow of any node is totally ordered. The maximal outgoing flow of exclusive and inclusive gateways w.r.t. this ordering plays the special role as the *default* flow. Graphically, this is represented by the layout order of the outgoing sequence flow, with a dash through the line of the default flow (if there are more than one outgoing flow). In the rest of the paper we will let $D \triangleq \{s \mid \mathcal{L}(\mathit{fst}(s)) \in \{\mathit{Excl}, \mathit{Incl}\} \wedge \forall s' : s \preceq s' \text{ implies } s' = s\}$ be the set of all default flows. We can then use this ordering when implementing the exclusive gateway in order to determine the order in which the conditions on the outgoing sequence flow should be evaluated. C represents a mapping from sequence flow to logical conditions that are evaluated in order to determine which outgoing sequence flow of some gateway receives tokens when that gateway fires. For our purposes, it suffices simply to define some condition $C(s)$ for some sequence flow $s \in S$ to be true just in case some evaluation function $\mathit{eval}(C(s))$ returns true. Note that it is not strictly necessary to introduce C and eval, as the rules could simply be rewritten to be nondeterministic, yielding a simpler semantics with equivalent behavior. However, the current formulation is closer to the semantics of full BPMN, and it makes clear exactly how this particular nondeterminism is to be removed when expanding the subset.

Example 1. *The process* $\mathcal{P}_1$ *in Fig. 3 is represented as the tuple* $(N, \mathcal{L}, S, \preceq, C, M)$*, where*

$$\begin{aligned}
N &= \{\alpha, \beta, \gamma, \delta, \} \\
\mathcal{L} &= \{(\alpha, \mathit{Start}), (\beta, \mathit{Incl}), (\gamma, \mathit{Excl}), (\delta, \mathit{End})\} \\
S &= \{(\alpha, \beta), (\beta, \gamma), (\beta, \delta), (\gamma, \beta)\} \\
C &= \{((\beta, \gamma), C_{\beta,\gamma}), \} \\
M &= \{((\alpha, \beta), 1), ((\beta, \gamma), 0), ((\beta, \delta), 0), ((\gamma, \beta), 0)\} \\
&\quad \textit{and } (\beta, \gamma) \preceq (\beta, \delta) \textit{ is the only non-trivial pair in } \preceq
\end{aligned}$$

where $C_{\beta,\gamma}$ *denotes some condition with associated evaluation function eval that returns either true or false. The set* D *of default flows given as the maximal outgoing flows w.r.t. the order* $\preceq$ *is* $\{(\beta, \delta), (\gamma, \beta)\}$*.*

Before giving the formal semantics of processes, we formalize the notions of incoming and outgoing sequence flows.

Definition 2 (Incoming/Outgoing Sequence Flow). *The incoming sequence flow of some node n, written $S_{in}(n)$, is defined as $S_{in}(n) = \{(n_0, n_1) \in S \mid n_1 = n\}$. The outgoing sequence flow S_{out} is defined as $S_{out}(n) = \{(n_0, n_1) \in S \mid n_0 = n\}$.*

Additionally:

Definition 3 (Source/Target Nodes on Sequence Flow). *The source and target nodes of some sequence flow $s = (n_0, n_1)$ are defined as $fst(s) = n_0$ and $snd(s) = n_1$.*

Example 2. *For $\mathcal{P}$ from Example 1 $S_{in}(\beta) = \{(\alpha, \beta), (\gamma, \beta)\}$ and $S_{out}(\beta) = \{(\beta, \gamma), (\beta, \delta)\}$.*

3.2 BPMN$_{\mathbf{inc}}$ Formal Semantics

The approach to semantics discussed in this section successfully implements the rules governing inclusive gateways informally discussed above. Evaluation is divided into two phases. The first phase of the evaluation consists of annotating each sequence flow with the set of paths from that sequence flow to each upstream token. In the second phase we use that information to determine if inclusive gateways can fire. Once a gateway fires, a token (and at most one) is obviously consumed.

The annotation map G has type $S \to 2^{N^*}$, or in other words, it maps sequence flow to sets of sequences of nodes. To avoid confusion, sequences of nodes are written in square brackets, where [] represents the empty sequence. G is computed with algorithm 3.1. One way to picture the operation of the algorithm (which we will elaborate a bit more in Section 3.3 below) is by imagining that each token sends a message down its sequence flow. That message, which starts with an empty payload, accumulates the unique identifiers of each of the gateways that it crosses. The message travels out of *all* of the outgoing sequence flow of each gateway it enters. If the gateway that the message is about to cross is already listed in the message payload, then it stops.

The algorithm consists of two working procedures AddToPath and StepPathAnnotation along with the main code contained in **main**. Procedure AddToPath simply takes a path of nodes $[a_0, a_1, \ldots, a_n]$ and appends a new node b at its end only if b is not already present. The other procedure, namely StepPathAnnotation takes a flow annotation G as an argument and returns a new one by updating G according to the marking M. More precisely, each sequence flow s containing a token (that is, where $M(s) > 0$) is annotated with the empty string. On the other hand, each edge with no tokens is updated according to the annotations of upstream sequence flows: if (n_0, n) and (n, n_1) are in S then $G((n, n_1))$ is also updated with AddToPath$(G((n_0, n)), n)$. Finally, the code in **main** computes the least fixed point of StepPathAnnotation.

Algorithm 3.1. ComputePathAnnotations(S, N, M)

```
procedure AddToPath($[a_0, a_1, \ldots, a_n], b$)
  /*Add $b$ to the end of the path only if $b$ is not in the path already*/
  if $b \in \{a_0, a_1, \ldots, a_n\}$
    then $p \leftarrow [a_0, a_1, \ldots, a_n]$
    else $p \leftarrow [a_0, a_1, \ldots, a_n, b]$
  return ($p$)

procedure StepPathAnnotation($G$)
  $G' \leftarrow \emptyset$   /*$G'$ represents the new annotation to be generated from $G$. */
  for each $s = (n_0, n_1) \in S$
    do
      if $M(s) > 0$
        then
          $G'(s) \leftarrow \{[]\}$
          /* Sequence flow containing a token always has
          an empty path to a token */
        else
          $p_s \leftarrow \emptyset$
          for each $s' \in S_{in}(n_0)$
            do $p_s \leftarrow p_s \cup \{\text{AddToPath}(p, n_0) \mid p \in G(s')\}$
          $G'(s) \leftarrow p_s$
          /* Each sequence flow gets incoming sequence
          flows' + the node on the left */
  return ($G'$)

main
  $G_0 \leftarrow$ StepPathAnnotation($\emptyset$)
  repeat
    do
      /* Iterate until the fixed point is found */
      $G_1 \leftarrow G_0$
      $G_0 \leftarrow$ StepPathAnnotation($G_0$)
  until $G_0 = G_1$
  return ($G_1$)
```

Example 3. *Applying Algorithm 3.1 to $\mathcal{P}$ from Example 1 yields the map*

$$\begin{aligned}(\alpha, \beta) &\mapsto \{[]\}\\ (\beta, \gamma) &\mapsto \{[\beta], [\beta\gamma]\}\\ (\gamma, \beta) &\mapsto \{[\beta\gamma]\}\\ (\beta, \delta) &\mapsto \{[\beta], [\beta\gamma]\}\end{aligned}$$

This annotation is illustrated visually in Fig. 4

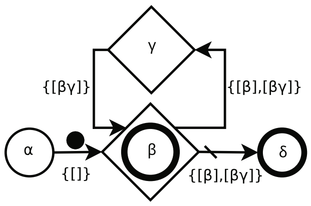

Fig. 4. Example path annotation

We can now state the following:

Theorem 1. *Algorithm 3.1 always terminates.*

Proof (Sketch). The fix point always exists simply because the topology of the process is finite, paths can at most contain each node once, and the function AddToPath is monotone. □

Given a BPMN$_{\text{inc}}$ process $(N, \mathcal{L}, S, \preceq, C, M)$ with path annotation G, a new marking M' can be obtained by applying any of the following rules to any node $n = (a, t)$.

First, to help keep the definition of the rule covering inclusive gateways readable, the function *NotBlocking*, which determines whether a particular empty incoming sequence flow prevents an inclusive gateway from activating, is defined as follows:

Definition 4. *The predicate NotBlocking*(s) *is true if and only if*

$$\begin{pmatrix} G(s) = \emptyset \quad \vee \\ \forall p \in G(s) : \begin{pmatrix} snd(s) \in p \quad \vee \\ \exists s' \in S_{in}(snd(s)) : \begin{pmatrix} M(s') > 0 \quad \wedge \\ \exists n \in p : Reachable(n, s') \end{pmatrix} \end{pmatrix} \end{pmatrix}$$

where

$$Reachable(n, s) \quad \triangleq \quad \Big(n = fst(s) \quad \vee \quad \exists s' \in S_{in}(fst(s)) : Reachable(n, s')\Big)$$

NotBlocking corresponds to the conditions defined in the specification under which an inclusive gateway is allowed to fire even though it has empty incoming sequence flow. When listing the requirements for empty incoming sequence flow to inclusive gateways, the specification states that the gateway may only fire if there is no token upstream of the empty sequence flow, unless the path to the token visits the inclusive gateway or any node on the path is upstream of one of the incoming sequence flows with a token [5, p. 401]. In the definition of *NotBlocking*:

– The clause $G(s) = \emptyset$ corresponds to there being no token upstream of the sequence flow, as it means there are no paths from a token to s.

– The first disjunct within the universally quantified clause represents the situation in which the path in question visits the inclusive gateway.
– The second disjunct represents the path in question being upstream of an incoming sequence flow that has a token. This is because if a node that is upstream of the empty incoming sequence flow includes another node that is part of a path to a sequence flow with a token, then that node is upstream of that sequence flow.

We now give the formal semantics of BPMN$_{\text{inc}}$ through three rules. In the following, we define $\delta_\phi = 1$ whenever the predicate ϕ is true and $\delta_\phi = 0$ otherwise. First, we give the rules for the activation of inclusive gateways:

$$(\textsc{Incl})\quad \frac{\begin{array}{ll}\forall s \in S_{in}(n) : (M(s) > 0 \vee \mathit{NotBlocking}(s)) & \exists s_t \in S_{in}(n) : M(s_t) > 0 \\ \exists s_{ok} \in S_{out}(n) : (s_{ok} \notin D \wedge \mathit{eval}(C(s_{ok}))) & \mathcal{L}(n) = \mathit{Incl}\end{array}}{\begin{array}{c}\forall s \in S_{in}(n) :\ M'(s) = M(s) - \delta_{M(s)>0} \\ \forall s' \in S_{out}(n) :\ M'(s') = M(s') + \delta_{(s' \notin D \wedge eval(C(s')))}\end{array}}$$

$$(\textsc{Incl}_{\text{DF}})\quad \frac{\begin{array}{ll}\forall s \in S_{in}(n) : (M(s) > 0 \vee \mathit{NotBlocking}(s)) & \exists s_t \in S_{in}(n) : M(s_t) > 0 \\ \neg\,(\exists s_{ok} \in S_{out}(n) : (s_{ok} \notin D \wedge \mathit{eval}(C(s_{ok})))) & \mathcal{L}(n) = \mathit{Incl}\end{array}}{\begin{array}{c}\forall s \in S_{in}(n) : M'(s) = M(s) - \delta_{M(s)>0} \\ \forall d \in S_{out}(n) : M'(d) = M(d) + \delta_{d \in D}\end{array}}$$

These rules construct a new marking M', which gives the marking of the ingoing and outgoing flows after the gateway has fired. (Incl$_{\text{DF}}$) activates the default outgoing sequence flow if no outgoing sequence flow with conditions receive tokens.

The clause $\mathcal{L}(n) = \mathit{Incl}$ in the premise of (Incl) simply restricts the rule to only apply to inclusive gateways. The condition $\exists s_t \in S_{in}(n) : M(s_t) > 0$ corresponds to the requirement in the specification that "At least one incoming sequence flow has at least one Token"[5, p. 401]. The condition $\forall s \in S_{in}(n) : (M(s) > 0 \vee \mathit{NotBlocking}(s))$ guarantees that each empty incoming sequence flow meets the requirements, as discussed above. Finally, the condition $(\exists s_{ok} \in S_{out}(n) : (S_{ok} \notin D \wedge \mathit{eval}(C(s_{ok}))))$ guarantees that the outgoing sequence flow should not be activated. In that case, (Incl$_{\text{DF}}$) should be activated instead. The two components of the conclusion of (Incl) are responsible for removing tokens from the incoming sequence flow and distributing them over the outgoing sequence flow according to the conditions, respectively.

The structure of (Incl$_{\text{DF}}$) is similar to that of (Incl). It simply states that if an inclusive gateway fires and none of its non-default outgoing sequence flow will receive a token, then the default sequence flow receives the token. Otherwise, it works as (Incl).

Exclusive gateways are much simpler:

$$\text{(Excl)}\ \frac{\begin{array}{ll} s \in S_{in}(n) & M(s) > 0 \\ s' \in S_{out}(n) & \mathit{eval}\left(C\left(s'\right)\right) \\ \forall s'' \in S_{out}(n) : \mathit{eval}(C(s'')) \implies s' \preceq s'' & \mathcal{L}(n) = \mathit{Excl} \end{array}}{\begin{array}{c} \forall s_{in} \in S_{in}(n) : M'(s) = M(s) - \delta_{s_{in}=s} \\ \forall s_{out} \in S_{out}(n) : M'(s') = M(s') + \delta_{s_{out}=s'} \end{array}}$$

Example 4. *Given G from Example 3, we can compute our new marking using the evaluation rules. Incl can be applied at β, as both (α, β) and (γ, β) satisfy InclSearch. Assuming that $eval(C_{\beta\to\alpha})$, we can then generate a new marking*

$$M' = \{((\alpha, \beta), 0), ((\beta, \gamma), 1), ((\beta, \delta), 0), ((\gamma, \beta), 0)\}$$

M' represents the new state of the process on which the token has been passed onward from the inclusive gateway.

3.3 Incremental and Distributed Implementations

The semantics given above formalizes the specification rather directly. If implemented naively, the computation of StepPathAnnotation and NonBlocking are a quite costly way to determine if an inclusive gateway can fire. Below we discuss two possible ways of implementing the semantics more efficiently by updating the data structures incrementally.

Incrementally computed path annotations. Alternatively to computing the path annotations from scratch for every step we can compute them incrementally. We then initially (and only once) compute the path annotations using the ComputePathAnnotation algorithm. Every time a gateway n is fired we update the path annotations for each flow s reachable from the outgoing flows of n as follows: If the marking of an outgoing s flow changes from 0 to 1 we remove n from the head of all the paths in annotations of the flows reachable from s (i.e. they receive a token and there was no token before). If all the incoming flows of the gateway n get marking 0 we also remove all paths from annotations of flows reachable from the outgoing flows that have n at their head. Figure 5 shows an example of this update on the example given in Figure 2 (default flows are left unspecified in the figure). After γ fires, it is removed from the heads of the paths of the downstream sequence flow annotations.

Precomputed path annotations and incrementally computed normalized markings. Given a marking, define the *normalized* marking $M_{0,1}$ by

$$M_{0,1}(s) = \delta_{M(s)>0}$$

For any BPMN$_{\text{inc}}$ process there are finitely many $(2^{|S|})$ normalized markings.

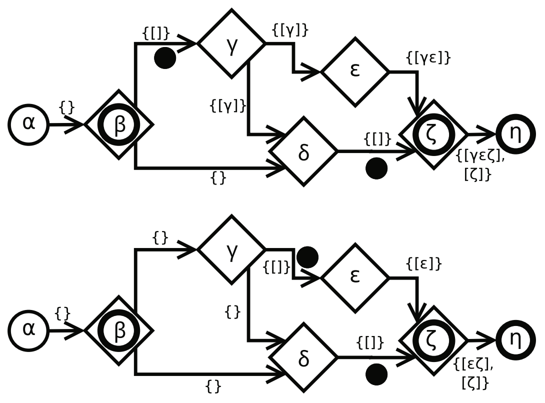

Fig. 5. Incremental path annotation update

Now, the function G computed by ComputePathAnnotations and the function *NonBlocking* for each flow only depend on the normalized marking of the flows that are reachable by following sequence flow backwards from the sequence flow in question. Consequently, as an alternative to incrementally maintaining the path annotations, the functions can be pre-computed for all of the possible normalized markings.

In addition to maintaining the markings of sequence flows, an implementation needs then only, for each sequence flow $(n, n') \in S$, to incrementally keep updated a table of normalized markings of the flows backwards reachable from the gateway n'. This can be done each time a gateway fires by updating the normalized marking (if it changes) for the flows reachable on paths starting on the outgoing flows of the gateway.

Distributed communication-based implementations. The two implementations above can be made in a distributed way by representing each gateway as a process and sequence flows as communication channels between processes. Each gateway process then maintains for each incoming flow the data structures (of respectively the path annotations or the table of normalized markings). It can then receive messages from gateways connected to incoming flows with updates, which can be forwarded (if necessary) to the gateways on outgoing flows.

4 Related Work

Previous work regarding the semantics of BPMN inclusive gateways can, broadly speaking, be divided into three main categories: semantics of BPMN that do not provide a non-local upstream search behavior for inclusive gateways, BPMN semantics that do, and semantics of other languages with similar constructs.

In [10], Völzer proposes three semantics for BPMN 2.0 inclusive gateways. Two of them are only restricted to the acyclic cases while the third semantics covers the general case (any kind of workflow) like ours. The latter is equipped with a linear algorithm for detecting the presence of upstream tokens.

All other previous work has been based on earlier versions of the BPMN standard, which were significantly less specific with regards to the semantics of the inclusive gateway. As far as the authors are aware, this is the first formal treatment of inclusive gateways making use of BPMN 2.0's semantics aside from [10], which introduced the approach used in the standard.

The definition of inclusive gateways in BPMN 1.0 is as follows:

Process flow SHALL continue when the signals (Tokens) arrive from all of the incoming Sequence Flow that are expecting a signal based on the upstream structure of the Process (e.g., an upstream Inclusive Decision).

The particular deficiency of the specification of inclusive gateways in BPMN 1.0[4] is highlighted in a paper by Dijkman, *et al*, that provides a mapping from a large subset of BPMN to Petri nets, enabling the use of standard tools for analysis of processes.

Dijkman *et al* point out that the definition of inclusive gateways fails to specify which sequence flow should expect a signal, and especially does not take into account situations in which the inclusive gateway is upstream from itself. These concerns are largely solved in the draft version of BPMN 2.0, which is quite specific on these matters.

Wong and Gibbons[11] present a translation of a subset of BPMN process diagrams to CSP. They characterize the inclusive gateway as simply accepting tokens on some subset of the incoming sequence flows and then generating tokens on some subset of the outgoing sequence flows.

Prandi *et al* [8] define a mapping from BPMN to the process calculus COWS. Similarly to other mappings based on BPMN 1, an inclusive join is simply translated into a process that can receive a signal on any subset of its inputs and then sends a signal onwards.[8, p.256]

Ye, *et al*'s translation to YAWL [12] is perhaps the most faithful to BPMN in its implementation of inclusive gateways. The inclusive gateway is mapped to a YAWL OR-join. In YAWL, the OR-join has non-local semantics similar to those defined for BPMN's inclusive gateway [9]. Indeed, Dijkman suggested that the designers of BPMN borrow these semantics directly in an aside in [2].

The inclusive gateway semantics proposed in [1] is compatible with the vague phrasing of BPMN 1.0. However, the model adopted to deal with cyclical process graphs is not compatible with the better-specified semantics in [5], as it divides the tokens in the cycle into groups based on the iteration in which they are produced, and then considers each iteration separately. BPMN 2.0 [5] did not adopt this model, and it allows tokens from many different iterations to interact.

Dumas *et al.* [3] provide a semantics for BMPN's inclusive gateways based on the imprecise BPMN 1.0 specification, and their solution ends up very similar to ours. However, they provide a means for resolving the resulting deadlock in the vicious circle example, which is a situation in which two inclusive gateways depend on each other cyclically. Since the informal specification that was eventually adopted in BPMN 2.0 does not include this resolution strategy, and as our work is a faithful translation, we do not include it.

BPMN's inclusive gateway is quite similar to constructs called "OR-joins" in various other process formalisms, in particular YAWL[9] and EPCs. Indeed, Dijkman suggested that the designers of BPMN borrow YAWL's semantics directly in an aside in [2].

In [6], Kindler points out that the informal semantics of OR-joins in Event driven Process Chains (EPCs), which contain a similar non-local backwards condition, can not be formalized consistently, due to the same sort of vicious circle treated in [3]. The BPMN 2.0. specification eliminates this problem, although the vicious circle example will result in a deadlock. However, as illustrated by the example in Fig. 2 BPMN 2.0 may exhibit race-conditions.

The primary difference between BPMN inclusive gateways and OR-joins in other notations is that a token that is upstream of both an empty and a full incoming sequence flow does not block the activation of the gateway in BPMN, while other languages do not include this stipulation. Therefore, straightforward translations of inclusive gateways to OR-joins such as Ye, *et al*'s translation to YAWL[12] are insufficient to capture BPMN 2.0's semantics.

5 Conclusion and Future Work

The particular behavior defined for inclusive gateways in BPMN 2.0 makes it difficult to provide a local semantics. However, given access to global information about the current state of execution, a precise semantics for inclusive gateways can be provided.

In this paper, one such precise semantics is provided. It differs from other approaches in that it does not attempt to translate BPMN process diagrams into other, better-understood calculi or process modeling languages. Instead, the semantics is provided directly in terms of a subset of BPMN. While this approach precludes the use of tools and methods developed for these other models, it can allow use of the semantics more easily either in implementations of BPMN or in providing analyses that are more easily understood by less-technical users who may not be familiar with other process models. Due to the non-local nature of inclusive gateways, this semantics requires the use of a backwards search algorithm for determining the parts of the global execution state that are relevant to each inclusive gateway at each step of the execution. We have sketched in Section 3.3 two approaches to how this search can be replaced by incremental updates to respectively token-paths annotations and to local copies of (normalized) markings after a gateway is fired.

Possibilities for future work include extending the semantics to cover more of BPMN, formalizing the incremental implementations of the semantics and prove their correctnes, and investigating applications of the semantics to real projects that make use of BPMN 2.0. Additionally, other approaches than token-based semantics, such as graph rewriting, can possibly be developed to give a simpler semantics for inclusive gateways. We also plan to implement the semantics following the different approaches sketched in Section 3.3, analyze their complexity and compare the implementations using a set of example $\mathrm{BPMN}_{\mathrm{inc}}$ processes.

References

1. Börger, E., Sörensen, O., Thalheim, B.: On defining the behavior of OR-joins in business process models. J. UCS 15(1), 3–32 (2009)
2. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Information and Software Technology 50(12), 1281–1294 (2008)
3. Dumas, M., Großkopf, A., Hettel, T., Wynn, M.T.: Semantics of standard process models with OR-joins. In: Chung, S. (ed.) OTM 2007, Part I. LNCS, vol. 4803, pp. 41–58. Springer, Heidelberg (2007)
4. Object Management Group. BPMN 1.0: OMG final adopted specification (February 2006), `http://www.bpmn.org/Documents/OMG_Final_Adopted_BPM_1-0_Spec_06-02-01.pdf` (accessed May 10, 2010)
5. Object Management Group. Business process modeling notation (BPMN) 2.0 beta 1(August 2009), `http://www.omg.org/cgi-bin/doc?dtc/09-08-14.pdf` (accessed May 10, 2010)
6. Kindler, E.: On the semantics of EPCs: Resolving the vicious circle. Data Knowl. Eng. 56, 23–40 (2006)
7. Peterson, J.L.: Petri nets. ACM Computing Surveys 9(3), 223–252 (1977)
8. Prandi, D., Quaglia, P., Zannone, N.: Formal analysis of BPMN via a translation into COWS. In: Wang, A.H., Tennenholtz, M. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 249–263. Springer, Heidelberg (2008)
9. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: yet another workflow language. Information Systems 30(4), 245–275 (2005)
10. Völzer, H.: A new semantics for the inclusive converging gateway in safe processes. In: Hull, R., Mendling, J., Tai, S. (eds.) BPM 2010. LNCS, vol. 6336, pp. 294–309. Springer, Heidelberg (2010)
11. Wong, P.Y.H., Gibbons, J.: A process semantics for BPMN. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 355–374. Springer, Heidelberg (2008)
12. Ye, J., Sun, S., Song, W., Wen, L.: Formal semantics of BPMN process models using YAWL. In: Second International Symposium on Intelligent Information Technology Application, IITA 2008, vol. 2, pp. 70–74, 20-22 (2008)