Behavior-based Decomposition of BPMN 2.0 Control Flow

Jan Kubovy, Dagmar Auer and Josef Küng
Institute for Application Oriented Knowledge Processing, Johannes Kepler University in Linz, Linz, Austria

Keywords: BPMN, Behavior Patterns, Decomposition, Modeling, Formal Method, ASM.

Abstract: The Business Process Model and Notation (BPMN) is a well-established industry standard in the area of Business Process Management (BPM). However, still with the current version 2.0 of BPMN, problems and contradictions with the underlying semantics of the meta-model can be identified. This paper shows an alternative approach for modeling the BPMN meta-model, using behavior-based decomposition. The focus in this paper is on control flow. We use Abstract State Machines (ASM) to describe the decomposition of the merging and splitting behavior of the different BPMN flow node types, such as parallel, exclusive, inclusive and complex, as defined in the BPMN 2.0 standard, resulting in behavior patterns. Furthermore an example for the composition of different gateway types is given using these behavior patterns.

1 INTRODUCTION

The Business Process Model and Notation (BPMN) is a well-established industry standard in the area of Business Process Management (BPM). Although BPMN is claiming a formal and clear underlying semantics some problems and contradictions have been discovered. Those problems were addressed by the scientific society using different formal methods, such as Petri Nets (Dijkman et al., 2007) or Abstract State Machine (ASM) method (Börger and Thalheim, 2008a; Börger and Sörensen, 2011). Much effort has been put into providing a formal layer, enhancing the existing informal meta-model for the BPMN standard. We base our work on the ASM approaches stated above. The ASM method (Börger and Stärk, 2003) evolved from Evolving Algebras (Gurevich, 1995). We use ASMs to formalize the BPMN meta-model. Our refinements follow the process execution conformance (Object Management Group (OMG), 2011, chap. 2.2) and formalize the particular part, we call behaviors.

The need for a formal technique to model workflows has already been considered in approaches before. E.g., formalizing existing workflow models by mapping them to well established formalism, i.e., Petri Nets (van der Aalst, 1998) further evolved into Yet Another Workflow Language (YAWL) trying to satisfy the needs of universal organizational theory and standard BPM concepts (van der Aalst and ter Hofstede, 2003). About the same time BPMN emerged and became a popular, widely-used standard in the area of BPM, among others, because of its intuitive graphical notation and mapping to Business Process Execution Language (BPEL) (Organization for the Advancement of Structured Information Standards (OASIS), 2007).

In this paper we will first describe the *enabling token set* concept in section 3, which is essential for activation of flow nodes across instances in one step. We continue with introducing the behavioral decomposition in section 4 and show three main behaviors: the gate behavior in section 4.1, the merge behavior in section 4.2 and the split behavior in section 4.3. We will discuss all possible merge and split behavior types, i.e., parallel, exclusive, inclusive and complex. Finally, these behaviors are used to define the gateway transition rules in section 5. Furthermore, we give an example of how such a behavior-based ground model (the blue print of the designed system (Börger and Stärk, 2003)) can be easily extended by another element in section 6. Section 7 concludes our results.

2 MOTIVATION

The Business Process Model and Notation (BPMN), in current version 2.0, (Object Management Group (OMG), 2011) is well established for describing business processes. With September 2013 BPMN has become an ISO/EC-Standard (reference number 19510:2013). Still the standard has some ambiguities and inconsistencies. While participating in a

Kubovy J., Auer D. and Küng J..
Behavior-based Decomposition of BPMN 2.0 Control Flow.
DOI: 10.5220/0004862302630271
In *Proceedings of the 16th International Conference on Enterprise Information Systems* (ICEIS-2014), pages 263-271
ISBN: 978-989-758-029-1

project with the focus to formally describe the BPMN standard, we dealt with those inconsistency problems and improved the meta-model structure to improve reusability. Those problems appear in both, the vertical refinements on one hand and the horizontal extensions to BPMN on the other. When formalizing the BPMN meta-model, duplicates appeared to be the reason for many of the inconsistencies in the existing standard. Thus, we try to extract the common behavior patterns, resulting in a revised, more compact, and graspable meta-model.

This work is based on process diagram as described in the BPMN 2.0 specification. This particular notation is chosen since it is popular between process designers and modelers and is the most commonly used graphical representation for business processes (Kubovy et al., 2012). We believe that the usability of our outputs will be supported by the fact that the potential reader does not have to work him/her-self into a completely new process modeling notation.

We start with the BPMN modeling practices, which contain among others gateway pairing or restriction of sequence flows on one flow node side (Freund et al., 2010). This may improve readability of simple diagrams, increase clearness and reduce ambiguity of even complex diagrams, but increases complexity of the diagrams in general since the amount of flow nodes in a process diagram dramatically bursts. By simplifying the meta-model, we achieve an unambiguous, clear, correct and more graspable meta-model, which is important for conformance and deployment. But the complexity growth of such process models may be harder to maintain during development and lower sustainability of processes based on such a meta-model.

One possible solution is to introduce containers (for the use in graphical notations) or macros (for the use in formal descriptions). We can demonstrate this, e.g, by restricting all flow nodes except gateways to only one incoming and one outgoing sequence flow. Handling the splitting and merging behavior will then remain only in one place, the gateway. Still, during the modeling of a process diagram we will not want to place a gateway in front or after every flow node merging or splitting more sequence flows. Thus, containers may be introduced to model extended modeling elements. Such containers will then hold multiple elements from the meta-model and establish a complex element keeping the model clean from problems, e.g., duplicate definitions, but enables the advantages of simple or basic (read small and graspable) modeling elements. This way we may concentrate on the essential purpose of the different flow node types and compose the complex flow elements, defined in the BPMN standard using such *simple flow elements*. An example of a *BPMN activity* as a container using a *simple activity* and two *simple gateways* is depicted in figure 1). Here we define a *Simple Activity* limited to exactly one incoming and one outgoing sequence flow with no merging nor splitting behavior. The possibility of multiple incoming or outgoing sequence flows, as described in the BPMN 2.0 standard, is enabled with the two gateways in figure 1. The first is a *simple exclusive merge gateway*, allowing only to merge the flow using XOR-Join and the second is *simple parallel split gateway* allowing only to split the flow to parallel paths. In the BPMN 2.0 standard both, the activity and the exclusive gateway, define the exclusive behavior and this leads to duplication in the standard and may also lead to duplication in the implementation of a workflow engine. Similarly for parallel split behavior of activity and parallel gateway. We identify such behaviors, which can be extracted and defined separately. We compose the complex flow elements using such behaviors. An advantage of this approach will be demonstrated by example in section 6 by extending the BPMN 2.0 meta-model with a *sole exclusive gateway*.

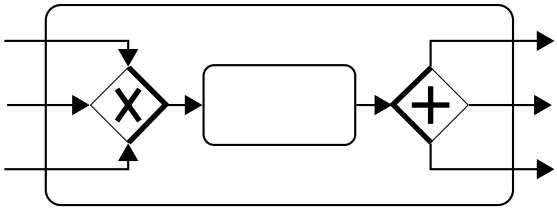


Figure 1: A BPMN activity composed using a container with two *simple gateways* and one *simple activity*.

3 ENABLING TOKEN SET

A token set is a set of tokens of the same instance, from distinct incoming sequence flows of one flow node. This set may potentially fire the given flow node. However, this has to be decided by a fire condition we will discuss later in this paper. We use `enablingTokenSets` shown in figure 2 to return sets of such token sets. In every token set all tokens have to belong to one instance. No two tokens residing in the same sequence flow can be present in the same token set. For example, there will be a flow node with three incoming sequence flows SF_1, SF_2 and SF_3 on which tokens from two different instances, I_A and I_B, will reside. The token distribution is the following: sequence flow SF_1 contains two tokens of instance I_A and one token of instance I_B, sequence flow SF_2 contains one token of each instance and sequence flow

SF_3 contains two tokens of instance I_B. The following four sets of tokens will be returned: Set 1 will contain two tokens of instance I_A from sequence flow SF_1 and SF_2. Set 2 will contain one token of instance I_A from sequence flow SF_1. Set 3 will contain three tokens of instance I_B from all the three incoming sequence flows. And last set 4 will contain one token of instance I_B from the sequence flow SF_3. Each of the token sets may or may not fire the given flow node, depending on the `MergeBehavior` defined later in this paper. The order of the token sets in the `enablingTokenSets` and the order of tokens in one set is irrelevant.

```
derived enablingTokenSets(node) =
 return res in
  local selected, all, set in
   res := {}
   set := {}
   all := ([ t | t is_in TOKENS holds
    sequenceFlowOf(t) is_in incoming(
        node)])
   while all != {} do
    selected := undef
    if set={} then choose selected
        is_in all
    else choose selected is_in all
        with
     forall token is_in set holds
      sequenceFlowOf(token) !=
          sequenceFlowOf(selected)
        and instanceOf(token) =
            instanceOf(selected)

    if selected != undef
     add selected to set
     remove selected from all
    if selected = undef or all = {}
     add set to res
     set := {}
```

Figure 2: Derived function for retrieval of enabling token sets for possible flow node activation.

4 BEHAVIORAL DECOMPOSITION

For decomposing we use a slightly different approach than containers discussed in section 2, still targeting the same issues. We define behavior patterns, similar to (Börger and Sörensen, 2011), and then reuse such patterns as building blocks to define the BPMN elements. This has, among others, the advantage that the intermediate step of defining *simple flow nodes* is not necessary. Furthermore, we can dynamically switch those building blocks using for example polymorphic or other suitable techniques. This allows to define one

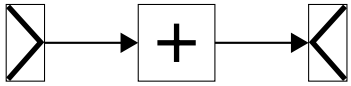


Figure 3: Parallel gateway decomposition using behavioral building blocks.

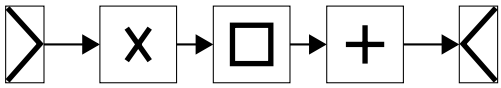


Figure 4: Activity decomposition using behavioral building blocks.

element instead of multiple container elements forming all possible combinations. E.g., the activity example in figure 1 would result in more than one container if we would also consider activities with no incoming or outgoing sequence flows or any combinations of the above. An example of a parallel gateway behavior decomposition is shown in figure 3. Using such behavioral building blocks will result in figure 4 to model the discussed BPMN activity. This time we do use much simpler building elements to compose the resulting flow node than using containers in figure 1. The reuse of such building block elements increases and the resulting meta-model definition will decrease in size and complexity (a behavior is defined only once). This is the core decomposition idea we will use in this paper to formally model the BPMN elements. This introduction should sketch the abilities and advantages of behavioral decomposition. This also comprises with the requirements capture of a system (Börger and Stärk, 2003), which is BPMN in our case, as we describe the behavior of each modeled element and split it into small parts based on similarities, observation and best practices (Freund et al., 2010).

For depicting behaviors, we use a square or rectangle (half as wide as high) both with sharp corners. This was chosen not to collide with any BPMN symbols. Inside the squares resides a BPMN symbol identifying the simple behavior. We use the designation “simple” for flow elements which deal with only the core of their purpose. An activity behavior, for example, does not deal with multiple incoming or outgoing sequence flows, just with performing an activity - dealing with the activity life-cycle. Similarly, the simple gateway behavior just deals with decision making, e.g., how the flow is merged or split. The quantity of incoming and outgoing sequence flows is realized by the input or output behavior as shown in Figure 3 and 4 as join and split behavior. In the next sections we will show, how this can be implemented.

```
derived canPassThrough(node, sequenceFlow, behavior) =
  return res in
    let otherSequenceFlows := outgoing(node) \ sequenceFlow in
      if gateCondition(sequenceFlow) = "DEFAULT" then
        res := forall outSequenceFlow is_in otherSequenceFlows holds
          gateCondition(outSequenceFlow) = FALSE

      else if behavior = "EXCLUSIVE" then
        res := gateCondition(sequenceFlow) = TRUE
            and forall outSequenceFlow is_in otherSequenceFlows holds (
          gateCondition(outSequenceFlow) != "DEFAULT"
            and gateCondition(outSequenceFlow) = FALSE)

      else if behavior = "INCLUSIVE" then
        res := gateCondition(sequenceFlow) = TRUE
```

Figure 5: Derived function determining if a token residing on a given sequence flow can pass through the given flow node.

4.1 Gate Behavior

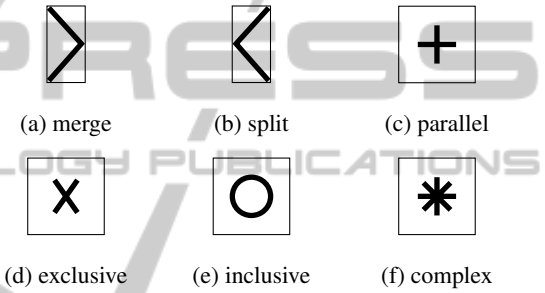
A gate is a mechanism, which somehow controls the flow. As in the real world, a gate has a guard watching the gate and letting in only those who fulfill certain conditions modeled by the `gateCondition` in figure 5.

A simple gate can be seen as a condition, which has to hold in order to let a token pass through. This mechanism will be further used in controlled flow, which is realized in the BPMN 2.0 specification using conditional sequence flows, gateways, but also other flow nodes, which implicitly allow control of the sequence flow. Since all flow elements share this behavior, it is extracted as a building block to allow reuse further in this paper. To notate this behavior a small diamond shape is used, shared for both the conditional sequence flows and gateways (Object Management Group (OMG), 2011).

4.2 Merge Behavior

A flow node may have one or more incoming sequence flows. In case of multiple incoming sequence flows such a flow node needs some kind of merging behavior. For this the symbols shown in figure 6a (van der Aalst, 1998) in conjunction with the different types - parallel shown in figure 6c, exclusive in figure 6d, inclusive in figure 6e and complex in figure 6f. For the different merging behavior types, we use the same symbols as in BPMN 2.0 inside the square symbolizing a behavior.

A merge behavior in general identifies possible `enablingTokenSets` of a flow node, each able to fire the given flow node. The given flow node can fire once or even multiple times in some cases. A token set in this sense is a combination of tokens residing on distinct incoming sequence flows of the given flow



(a) merge (b) split (c) parallel

(d) exclusive (e) inclusive (f) complex

Figure 6: The different used behaviors.

node with a distribution possibly able to fire the given flow node. There are no tokens in one set, which share the same incoming sequence flow and instance of a process or sub-process. A set of such sets will be used further in this paper to handle all possible, even multiple, fires across the process or sub-process instances. Every time a flow node fires, all `firingTokens` of the flow node will be consumed. Such `firingTokens` are a union of all tokens in all `enablingTokenSets`, which can actually fire the given flow node. The `rule MergeBehavior` shown in figure 7 defines the abstract behavior pattern for merging multiple sequence flows of a flow node. This rule returns all relevant instances of a (sub-)process containing the given flow node, in which the flow node fired. The size of the set of these `firingInstances` identifies the number of fires and each item in this set is a process instance, in which the given flow node fired. Any tokens, which contributed to any of the fires returned by the abstract `derived firingTokens` function, has to be in one of the instances returned by the `firingInstance` function and has to be consumed.

Parallel Merge Behavior: firing a flow node if there is a token on every incoming sequence flow of that flow node (Object Management Group (OMG),

```
rule MergeBehavior(node) =
  return res in
    forall token in firingTokens(node
        ) do
      ConsumeToken(token)
    res := firingInstances(node)
```

Figure 7: Rule defining the behavior for merging multiple sequence flows.

2011, Tab. 13.1). The possibility of multiple instances has to be taken into account. This is realized by the mentioned `enablingTokenSets`, where each token set only contains tokens from the same instance and different incoming sequence flows. The `rule ParallelMergeBehavior` has to check if each set contains a token for all incoming sequence flows. It (see figure 8) returns tokens from incoming sequence flows if and only if all of the incoming sequence flows have a token realized using `enablingTokenSets`. Therefore a token will be returned from `firingTokens` if and only if such a token is in a token set containing a token for each incoming sequence flow. The quantity of `firingInstances` represents the minimum quantity of tokens in one of the incoming sequence flows for a certain instance. Using the `enablingTokenSets` it is possible to separate multiple tokens from the same instance.

```
rule ParallelMergeBehavior(node) =
 MergeBehavior(node) where
  firingTokens(node) =
   return res in
    res := ([ t | t is_in tokenSet,
        tokenSet is_in
        enablingTokenSets(node) holds
      forall sf is_in incoming(node)
       exists tokenInSameSet is_in
           tokenIn(sf)
         and tokenInSameSet is_in
             tokenSet ])

  firingInstances(node) =
   return res in
    choose sf is_in incoming(node) in
     res := ([ i | i is_in INSTANCES
         holds
       exists t is_in firingTokens(
           node)
         and sequenceFlowOf(t) = sf
         and instanceOf(t) = i ])
```

Figure 8: Rule defining the behavior for parallel merging of multiple sequence flows.

Exclusive Merge Behavior: firing a flow node if there is at least one token on at least one incoming sequence flow of that flow node (Object Management Group (OMG), 2011, Tab. 13.2). Taking multiple instances into account, this behavior will fire the given flow node for every token residing in any directly incoming sequence flow in any instance. The `rule ExclusiveMergeBehavior` shown in figure 9 refines the `rule MergeBehavior`. This behavior fires the given flow node for any token that is present on any of the incoming sequence flows. No matter if only one or more tokens from the same instance are present on the directly incoming sequence flows, all will be returned. It may also fire multiple times in the same instance. Therefore the returned set of instances from `firingInstances` may contain duplicate instances.

```
rule ExclusiveMergeBehavior(node) =
 MergeBehavior(node) where
  firingTokens(node) =
   return res in
    res := { t | t is_in TOKENS holds
     exists sf is_in incoming(node)
      t is_in tokensIn(sf) }

  firingInstances(node) =
   return res in
    res := { i | i is_in INSTANCES
        holds
     exists t is_in firingTokens(node
         )
      instanceOf(t) = i }
```

Figure 9: Rule defining the behavior for exclusive merging of multiple sequence flows.

```
rule InclusiveMergeBehavior(node) =
 MergeBehavior(node) where
  firingTokens(node) =
   return res in
    res := ([ t | t is_in tokenSet,
        tokenSet is_in
        enablingTokenSets(node) holds
      UpstreamToken(node,incoming(node
         ),tokenSet,instanceOf(t))={}
        ])

  firingInstances(node) =
   res := ([ i | i is_in INSTANCES
       holds
     exists tokenSet is_in ([ ts | ts
         is_in enablingTokenSets(node)
         holds
       exists t is_in firingTokens(
           node)
         and t is_in ts ])
      exists token is_in tokenSet
       and instanceOf(token) = i ])
```

Figure 10: Rule defining the behavior for inclusive merging of multiple sequence flows.

```
rule ComplexMergeBehavior(node) =
 MergeBehavior(node) where
  let firingTokensForNode := {}, firingInstancesForNode := in
   forall tokenSet is_in enablingTokenSets(node) do
    choose tok is_in tokenSet do

     // Waitig for start state:
     if waitingForStart(node, instanceOf(tok)) then
      if activationCondition(node, instaceOf(tok)) then
       ignoreDuringReset(node, instanceOf(tok)) := { sf | sf is_in incoming(node)
            holds
        exists t is_in tokenSet and sequenceFlowOf(t) = sf }
       waitingForStart(node, instaceOf(tok)) := FALSE
       forall t in tokenSet add t to firingTokensForNode
       add instanceOf(tok) to firingInstancesForNode

     else // Waitig for reset state:
      let relevant := (incoming(node) \ ignoreDuringReset(node, instanceOf(tok)))
          in
       if UpstreamToken(node, relevant, tokenSet, instanceOf(tok)) = {} then
        ignoreDuringReset(node) = undef
        waitingForStart(node, instanceOf(tok)) := TRUE
        forall t in tokenSet add t to firingTokensForNode
        add instanceOf(tok) to firingInstancesForNode
   derived firingTokens(node) = firingTokensForNode
   derived firingInstances(node) = firingInstancesForNode
```

Figure 11: Rule defining the behavior for complex merging of multiple sequence flows.

Inclusive Merge Behavior: firing the flow node if there is at least one token on at least one incoming sequence flow and there are no `UpstreamTokens` (Völzer, 2010) related to that flow node (Object Management Group (OMG), 2011, Tab. 13.3). The `rule InclusiveMergeBehavior` shown in figure 10 refines the `rule MergeBehavior` and fires the given flow node if at least one token is present on any of the directly incoming sequence flows and there is no token in the process, which "may arrive" (Völzer, 2010). This is defined using the mentioned `UpstreamToken` rule. If no such upstream token exists, this behavior will fire. It will return the contributing tokens from all `enablingTokenSets`, which fired the flow node. All possible instances of those tokens will be return by the `firingInstances` function.

Complex Merge Behavior: defines two activation behaviors: first, when the flow node is in `waitingForStart` state, and second, if it is not in that state. In the first case the flow node gets fired if the `activationCondition` holds and in the second one it is fired if all `Upstream Tokens` from the first phase arrive (Object Management Group (OMG), 2011, Tab. 13.5), i.e., same as `InclusiveMergeBehavior` without `sequenceFlowsToIgnoreDuringReset`, which are those that fired the flow node in the `waitingForStart` state. The `rule ComplexMergeBehavior` shown in figure 11 takes, compared to other merge behaviors, the mentioned internal state into account. The internal state is updated by this rule only. It may be read for the purpose of outgoing sequence flow conditions but not changed anymore. Firing a flow node using this `ComplexMergeBehavior` in the `waitingForStart` state is straight forward: only the `activationCondition` needs to be fulfilled. In the "waiting for reset" state, the firing condition is much the same as in the `InclusiveMergeBehavior` shown in figure 10. The slight difference is that the set of relevant sequence flows observed for a token, which "may arrive", does not include the sequence flows, which contributed to the activation in the first phase (Object Management Group (OMG), 2011). The set of `firingTokens` only holds those, which contribute to fire one of the `firingInstances`. All other tokens residing on any of the directly incoming sequence flows of the flow node will not be included in that set.

Here we can already see (in case of the "waiting for reset" state) a reuse of previously defined parts, which decreases the specification volume and enables consistency in contrast to the original specification.

4.3 Split Behavior

Flow nodes may also have one or more outgoing se-

```
rule SplitBehavior(node, instance) =
 forall sequenceFlow is_in
     allowedOutgoing(node)
  ProduceToken(sequenceFlow, instance
      )
```

Figure 12: Rule defining the behavior for splitting multiple sequence flows.

quence flows. Such a flow node then needs some kind of split behavior. For a splitting behavior the symbols shown in figure 6b (van der Aalst, 1998) in conjunction with the different types - parallel shown in figure 6c, exclusive in figure 6d, inclusive in figure 6e and complex in figure 6f. The `rule SplitBehavior`, shown in figure 12, defines the abstract behavior for splitting the flow into multiple sequence flows of a flow node. This rule produces a token on all `allowedOutgoing` sequence flows for the given flow node. According to BPMN 2.0 specification, we distinguish three different split behaviors: parallel, exclusive, and inclusive split behavior.

Parallel Split Behavior: defining all existing directly `outgoing` sequence flows as `allowedOutgoing` sequence flows and producing a token on each of them. The `rule ParallelSplitBehavior` (see figure 13) refines the `rule SplitBehavior`. This is used as a default splitting behavior in BPMN 2.0 for any flow node except exclusive, inclusive, complex and event-based gateways.

```
rule ParallelSplitBehavior(node,
    instance) =
 SplitBehavior(node, instance) where
  allowedOutgoing(node)=outgoing(node
      )
```

Figure 13: Rule defining the behavior for parallel splitting of multiple sequence flows.

Exclusive Split Behavior: determining if a token `canPassThrough` a flow node using a specific outgoing sequence flow and the *"EXCLUSIVE"* behavior, producing a token on exactly one directly `outgoing` sequence flow, where the `gateCondition` allows such a passage. The `rule ExclusiveSplitBehavior` shown in figure 14 refines the `rule SplitBehavior`, which is used in exclusive gateways.

Inclusive Split Behavior: determining if a token `canPassThrough` a flow node using a specific outgoing sequence flow and the *"INCLUSIVE"* behavior, producing a token on any subset of `outgoing` sequence flows, where the `gateCondition` allows such a passage. The `rule InclusiveSplitBehavior`

```
rule ExclusiveSplitBehavior(node,
    instance) =
 SplitBehavior(node, instance) where
  allowedOutgoing(node) := ([ sf |
     sf is_in outgoing(node) with
   canPassThrough(node,sf,"EXCLUSIVE
       ")])
```

Figure 14: Rule defining the behavior for exclusive splitting of multiple sequence flows.

```
rule InclusiveSplitBehavior(node,
    instance) =
 SplitBehavior(node, instance) where
  allowedOutgoing(node) := ([ sf | sf
      is_in outgoing(node) with
   canPassThrough(node,sf,"INCLUSIVE
       ")])
```

Figure 15: Rule defining the behavior for inclusive splitting of multiple sequence flows.

shown in figure 15 refines the `rule SplitBehavior`, which is used in inclusive and complex gateway.

5 GATEWAY TRANSITIONS

Now the required building blocks for gateways are available (we discuss here only non-event-based gateways). The `GatewayTransition` can be built along with all the specific gateway type transition rules. The `DataBasedGatewayTransition` shown in figure 16 defines a transition for non-event-based gateways by refining the `WorkflowTransition` (Börger and Thalheim, 2008b; Börger and Thalheim, 2008a). It gathers relevant `enablingTokenSets`. If the concrete `MergeBehavior` allows to fire the given flow-based gateway, it will produce tokens on all `allowedOutgoing` sequence flows realized by the concrete `SplitBehavior`.

The `ParallelGatewayTransition` shown in figure 17, the `ExclusiveGatewayTransition` in figure 18, and the `InclusiveGatewayTransition` in figure 19 are then simply composing the existing behavior patterns. This will be that simple for other possible flow node transition rules as well, e.g., for activities or events. Other flow nodes will of course refine a different rule than the `DataBasedGatewayTransition` used for non-event-based gateways where their specific behavior and additional constrains may be defined.

More difficult is the case of `ComplexGatewayTransition` shown in figure 20. There is a specific `ComplexMergeBehavior` but no specific `SplitBehavior`. This is because the complex gate-

```
rule DataBasedGatewayTransition(node)
    =
 let firingInstances := MergeBehavior
    (node) in
  rule WorkflowTransition(node) where
   derived controlCondition(node) =
    return res in
     res := firingInstances != {}

   derived eventCondition =
    return res in
     res := true

   rule ControlOperation(node) =
    forall instance is_in
        firingInstances
     SplitBehavior(node, instance);

   rule EventOperation(node) =
    skip
```

Figure 16: Rule DataBasedGatewayTransition.

```
rule ParallelGatewayTransition(node)
    =
 DataBasedGatewayTransition(node)
    where
  rule MergeBehavior(node) =
     ParallelMergeBehavior(node)
  rule SplitBehavior(node) =
     ParallelSplitBehavior(node)
```

Figure 17: Rule ParallelGatewayTransition.

```
rule ExclusiveGatewayTransition(node)
    =
 DataBasedGatewayTransition(node)
    where
  rule MergeBehavior(node) =
     ExclusiveMergeBehavior(node)
  rule SplitBehavior(node) =
     ExclusiveSplitBehavior(node)
```

Figure 18: Rule ExclusiveGatewayTransition.

```
rule InclusiveGatewayTransition(node)
    =
 DataBasedGatewayTransition(node)
    where
  rule MergeBehavior(node) =
     InclusiveMergeBehavior(node)
  rule SplitBehavior(node) =
     InclusiveSplitBehavior(node)
```

Figure 19: Rule InclusiveGatewayTransition.

way split behavior is the same as for inclusive gateway (Object Management Group (OMG), 2011).

The above described behaviors enclose the formal description of the meta-model for basic control flow

```
rule ComplexGatewayTransition(node) =
 DataBasedGatewayTransition(node)
    where
  rule MergeBehavior(node) =
     ComplexMergeBehavior(node)
  rule SplitBehavior(node) =
     InclusiveSplitBehavior(node)
```

Figure 20: Rule ComplexGatewayTransition.

definition. In the next section we will demonstrate on an example, how this ground model can be extended. This demonstrates an appropriate decomposition of the meta-model, presented in this paper, which enables an easy and correct horizontal extension.

6 EXAMPLE FOR EXTENDING THE GROUND MODEL

We will demonstrate how simple it is to extend the ground model, sketched in the previous sections of a different behaviors and flow node types. Let's look closer at the exclusive gateway. What should happen, if there are more than one tokens distributed on any of the incoming sequence flows. We defined here that each of these tokens will fire the given gateway, resulting in possible multiple fires in one instance and step. This behavior is based on the BPMN 2.0 specification (Object Management Group (OMG), 2011), e.g., the exclusive merging behavior for exclusive gateways and activities. But this might be also seen as a possible ambiguity. In some cases we might want to use the exclusive gateway in situations, where more than one token is present on any distinct incoming sequence flow is not permitted. There is no way to do that with the currently existing flow nodes and behaviors defined above. Therefore, we decided to extend the ground model.

For this purpose, we define a specialized `MergeBehavior` for the new *sole exclusive gateway*. Such a gateway is allowed to have only one incoming sequence flow containing a token. Thus, more than one incoming sequence flow containing a token is considered as a violation of the `rule SoleExclusiveMergeBehavior` shown in figure 21 and will raise an `AlternativePathViolation` error. On the other hand multiple tokens in the same incoming sequence flow will not raise such an error. This is considered as correct behavior and will fire the given flow node for each token residing in such a sequence flow.

We do not discuss here if more tokens on the same incoming sequence flow is a correct interpretation of an exclusive gateway or not. We allow this

```
rule SoleExclusiveMergebBehavior(node
    )=
 constraint AlternativePathViolation
     =
  forall set is_in enablingTokenSets(
      node) holds
   |set| = 1

 ExlusiveMergeBehavior(node)
```

Figure 21: Rule SoleExclusiveMergeBehavior.

since more than one tokens can be produced on the same sequence flow for example by an activity if the `completionQuantity` attribute is greater than 1 (Object Management Group (OMG), 2011, Tab. 10.3). The goal of *sole exclusive gateway* is to correctly merge alternative paths - distinct incoming sequence flows to a given flow node. The aim of this paper is also not to show this particular behavior as a problem.

The goal of this example is to show that if such a behavior needs clarification, it can be easily introduced by extending the ground model. We leave the discussion about correct behavior of an exclusive gateway to future work.

7 CONCLUSION

We showed an approach of a behavioral decomposition of the BPMN 2.0 control flow concept. The decomposition of the meta-model is demonstrated on gateways, since we started with the merge behavior of incoming sequence flows and split behavior of outgoing sequence flows. The split and merge behaviors, which are shared across other flow node types too, such as activities or events, can be simply reused in their transition rules as discussed in section 5. The different flow node types can then be defined by composing different behavior patterns as shown for activities in figure 4. The goal this decomposition is to define a behavior once in a way it is reusable across the defined system. This enables that such a model is better graspable and understandable by the target audience.

Also such a resulting ground model is easily extendable with other behaviors. This was shown on a theoretical extension example in section 6, where the discussed ground model was extended by a *sole exclusive gateway* type. We leave the discussion about the correct behavior of exclusive gateways to some future work.

ACKNOWLEDGEMENTS

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

REFERENCES

Börger, E. and Sörensen, O. (2011). BPMN Core Modeling Concepts: Inheritance-Based Execution Semantics. In Embley, D. W. and Thalheim, B., editors, *Handbook of Conceptual Modeling: Theory, Practice and Research Challenges*, chapter 9. Springer-Verlag.

Börger, E. and Stärk, R. F. (2003). *Abstract State Machines - A Method for High-Level System Design and Analysis*. Springer-Verlag.

Börger, E. and Thalheim, B. (2008a). A method for verifiable and validatable business process modeling. *Advances in Software Engineering, LNCS*, 5316:59–115.

Börger, E. and Thalheim, B. (2008b). Modeling Workflows, Interaction Patterns, Web Services and Business Processes: The ASM-Based Approach. In *Proceedings of the 1st international conference on Abstract State Machines, B and Z*, pages 24–38. Springer Berlin / Heidelberg.

Dijkman, R. M., Dumas, M., and Ouyang, C. (2007). Formal Semantics and Analysis of BPMN Process Models using Petri Nets.

Freund, J., Rücker, B., and Heinninger, T. (2010). *Praxishandbuch BPMN Incl. BPMN 2.0*. Carl Hanser Verlag München Wien.

Gurevich, Y. (1995). Evolving Algebras 1993: Lipari Guide. *Specification and Validation Methods*, pages 231–243.

Kubovy, J., Geist, V., and Kossak, F. (2012). A Formal Description of the ITIL Change Management Process Using Abstract State Machines. *2012 23rd International Workshop on Database and Expert Systems Applications*, 0:65–69.

Object Management Group (OMG) (2011). Business Process Model and Notation (BPMN) 2.0. www.omg.org/spec/BPMN/2.0/.

Organization for the Advancement of Structured Information Standards (OASIS) (2007). Web Services Business Process Execution Language Version 2.0. https://www.oasis-open.org/standards.

van der Aalst, W. M. and ter Hofstede, A. H. (2003). YAWL: Yet Another Workflow Language. *Information Systems*, 30:245–275.

van der Aalst, W. M. P. (1998). The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66.

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*, BPM'10, pages 294–309, Berlin, Heidelberg. Springer-Verlag.