

Notification Concept for BPMN Workflow Interpreter using the ASM method

Jan Kubovy, Josef Küng

Institute for Application Oriented Knowledge Processing (FAW)
Johannes Kepler University in Linz
{jkubovy,jkueng}@faw.jku.at
<http://www.faw.jku.at>

Abstract. In this paper we focus on filling the gap between the formal Business Process Model and Notation (BPMN) Abstract State Machine (ASM) ground model[1, 3] and a Workflow Interpreter (WI) implementation. For that purpose we use an execution *context* concept and *notification* concept, a refinement of triggers (event definitions)[4].

1 Introduction

The motivation for this paper is to fill the gap between abstract Business Process Model and Notation (BPMN) Abstract State Machine (ASM) ground model defined in [1, 3] and the Workflow Interpreter (WI). The basic idea of how a WI can be seen in [3] as `rule WorkflowTransitionInterpreter`, which is firing the `rule WorkflowTransition : flowNodes`. This abstract rule handles the traversal of a token through all the instances of all the processes deployed to the Workflow Engine (WE), which the WI is the core part of. The `WorkflowTransition : flowNodes` is further refined in [1] for the concrete flow node types such as activities, gateways and events. The communication between the process run and the WI is refined in this paper by defining the rules and locations left abstract in [1] as the communication between processes using messages or signals.

Starting with defining the execution *context tree* in section 2, where the root of that tree is the *static context* present only once. Its immediate children are *root contexts* created for each running top-level process. The rest of the *context tree* is formed by *sub contexts* which are created for every new activity instance. After building a *context tree*, *notifications*, defined in section 3, can carry triggers to their destination through it. These *notifications* are ordered according to their time of occurrence and are processed in that order. *Implicit triggers*, defined in section 4, that are triggered when certain conditions occur in the workflow, automatically throw corresponding *notifications* in such case. Inter-process communication is in [4] possible using messages and signals. For those we define in section 5 the creation of the corresponding *notifications*. Finally, in section 6 and 7, the rest of the forwarding concepts[4], the *publication* concept to forward the *notifications* down the *context tree* and the *propagation* to forward the *notification* up, are formally defined.

2 Context

The environment communicates with the WI through existing contexts. There are three types of context (*Static Context*, *Root Context* and *Sub Context*) but only two (*Static Context* and *Root Context*) are allowed to be accessed by the environment. By environment we mean here the rest of the WE of an arbitrary execution platform.

Static Context exists only once and is created as soon as the WE is started. All existing deployments expose their defined top-level start events of their `rootProcessOfDeployment : Deployments → Processes`, which are those with no defined trigger or with “Message”, “Timer”, “Conditional” or “Signal” trigger [4], to the *static context* waiting for a corresponding *notification* to be fired (see section 3).

Root Context is a context created for each new `rootProcessOfDeployment → Processes` instance. The environment can communicate with such instance by sending *notifications* to existing events or by evaluating `Completed : Instances → Boolean[1]` to `true` as soon as the corresponding task is finished. The rules for finishing the different task types will be defined further in this paper. The instance of a context, for which it was created, can be obtained from the `instanceOfContext : Contexts → Instances` location.

Sub Context is a context created for every new activity instance inside a running process instance. This context is not visible to the environment but may populate some uncaught events to the `parentContext : Contexts → Contexts`. Notifications sent by the environment to the *root context* may be populated down to the *sub contexts*.

3 Notifications

We define a *notification* as an object similar to a token but carrying event definition or trigger through the *context tree*. The environment puts all *notifications* to the `notifications → List`, an ordered list, where the *notifications* are ordered by their `monitored timeOfOccurrence : Notifications → Time`. Such timestamp is created by an arbitrary *time authority machine*, i.e. neither by the environment nor by the WI machine, which can only read this value. Therefore, for both it is a monitored dynamic location[2].

Notifications fire a concrete trigger (or event definition see [4, sec. 10.4.5]) of a defined event. It specifies the `shared contextOfNotification : Notifications → Contexts` where the *notification* happened. Since a *notification* can be created by both, the environment and the WI this function is shared.

A *notification* may also define a concrete flow node the *notification* is meant for. This is optional and `shared nodeOfNotification : Notifications → FlowNodes` may be left *undef*. The case we want to assign a concrete event

to a *notification* is e.g. in the case when the *notification* is created by the environment to start a new process instance. Using this location we can select a concrete event with undefined (none) trigger to start the process. Other *notifications* may not specify a concrete flow node. Concrete flow node will be assigned by a *context* as soon as the *notification* reaches the *context* where an appropriate flow node is present. For example if a error event happens a corresponding *notification* is created but the catching event is not know yet and therefore the **shared nodeOfNotification : Notifications \rightarrow FlowNodes** is *undef*. The WI then searches for the catching event using the *propagation* concept[4] in this case, further defined in section 7. This way we implement the different forwarding concepts[4, sec. 10.4.1] of event triggers. Eventually, every *notification* will be assigned to a concrete flow node or removed from the **notifications \rightarrow List**. See section 6 and 7 for details about the assignment of flow nodes to *notifications* and their removal from the **notifications \rightarrow List**.

The trigger of a *notification* may be defined using the **shared triggerOfNotification : Notifications \rightarrow Triggers** location. If this location is left *undef* it represents the “None” trigger.

The **notifications \rightarrow List** is processed by the rule **ProcessNotificationList** shown in listing 1. Since the list is ordered by **monitored timeOfOccurrence : Notifications \rightarrow Time** the **select** construct will always return the oldest *notification*. This way is assured that older *notifications* will be processed before newer ones. The rule **ProcessNotificationList** will wait till the selected *notification* is assigned to a concrete flow node (see section 6 and 7). Unassigned *notifications* have to be first forwarded by other rules defined further in this paper.

Listing 1. ProcessNotificationList

```

rule ProcessNotificationList =
   $\forall$  notification  $\in$  notifications |
    nodeOfNotification(notification) = undef do
      ForwardNotification(notification)

  select notification  $\in$  notifications do
    if nodeOfNotification(notification)  $\neq$  undef then
      let trigger  $\leftarrow$  triggerOfNotification(notification),
        node  $\leftarrow$  nodeOfNotification(notification) in
        if flowNodeType(node)  $\in$  Events then
          TriggerOccurs(trigger, node)  $\leftarrow$  true
        else if flowNodeType(node) = "Receive"
          Received(trigger, node)  $\leftarrow$  true
        remove notification from notifications
    else
      skip

```

4 Implicit triggers

Concept of implicitly thrown events is defined here to enable the WI control also the implicit events. The rule **ThrowImplicitNotification**, shown in listing 2, is then responsible for observing conditional and timer triggers and throw corresponding *notifications* if their conditions were met. The assumption made is that every timer can be generalized as a conditional trigger and the attributes: `timeDate`, `timeCycle` and `timeDuration`[4, table 10.101] can be expressed by a `condition`[4, table 10.95].

Listing 2. rule **ThrowImplicitNotifications**

```
rule ThrowImplicitNotifications =  
   $\forall$  process  $\in$  Processes do  
     $\forall$  event  $\in$  flowNodes(process) |  
      flowNodeType(event)  $\in$  { "Conditional", "Timer" } do  
         $\forall$  instance  $\in$  instances(process) do  
          if eventCondition(event, instance) = true then  
            let context  $\leftarrow$  choose {context | context  $\in$  Contexts  
               $\wedge$  instanceOfContext(context) =instance} in  
              let n  $\leftarrow$  new Notifications in  
                contextOfNotification(n)  $\leftarrow$  context  
                nodeOfNotification(n)  $\leftarrow$  event  
                triggerOfNotification(n)  $\leftarrow$  trigger(event)
```

As the `nodeOfNotification : Notifications \rightarrow FlowNodes` is defined with the creation of the *notification* this forwarding concept is in [4] referred to as “direct resolution”.

5 Message and signal pools

For the purpose of inter process communication the BPMN standard defines Messages and Signals[4]. The main difference is that a Message specifies a target but Signal broadcasts to all Signal Catch events. The second most significant difference is that Signals just trigger the corresponding catch events but Messages usually carry more complex content and also may call a Service operation[4, sec. 8.4.3, fig. 8.30, 10.89].

The source and the target event of a Message are linked by a Message flow defined in collaboration[4, ch.9], but this is out of the execution scope. First, collaboration is not required for **BPMN Process Execution Conformance** nor for **BPMN BPEL Process Execution Conformance** [4, p. 109]. Second, implementation of message flows will require loading all communicating processes into the WI which is not always possible, since different processes may be running on different WE and still be communicating. For this purpose we define a `messagePool \rightarrow Set` and a `signalPool \rightarrow Set`. Messages or Signals arriving from the environment are converted to notifications (see section 3) as defined in the rule **ProcessMessagePool**, shown in listing 3, and similarly for Signals in the rule **ProcessSignalPool**, shown in listing 4.

Listing 3. rule ProcessMessagePool

```

rule ProcessMessagePool =
   $\forall$  message  $\in$  messagePool do
    let notification  $\leftarrow$  new Notifications,
    let trigger  $\leftarrow$  new triggers("Message") in
      messageOfTrigger(trigger)  $\leftarrow$  message
      triggerOfNotification(notification)  $\leftarrow$  trigger
      contextOfNotification(notification)  $\leftarrow$  "StaticContext"
    remove message from messagePool

```

Listing 4. ProcessSignalPool

```

rule ProcessSignalPool
   $\forall$  sig  $\in$  signalPool do
    let notification  $\leftarrow$  new Notifications,
    let trigger  $\leftarrow$  new triggers("Signal") in
      signalOfTrigger(trigger)  $\leftarrow$  signal
      triggerOfNotification(notification)  $\leftarrow$  trigger
      contextOfNotification(notification)  $\leftarrow$  "StaticContext"
    remove sig from signalPool

```

6 Event publication

Publication forwarding concept defined in [4] applies for message and signal events. The creation of *notifications* for such events coming from the environment is shown in section 5. Here, the publication of those *notifications* is defined in listing 5.

As communicating processes can be running on different WE the proposal is to define the message flow as a matching concept based on the name of the message (see [4, figure 10.89]). After a message event with the corresponding name is found, it will be fired and the *notification* will be consumed. Similarly for signals[4, figure 10.93] but with the exception that the corresponding *notification* will not be consumed which allows multiple signal catch events to catch the signal. For this purpose we define $\text{name} : \text{Message} \rightarrow \text{String}$ and $\text{name} : \text{Signal} \rightarrow \text{String}$.

Listing 5. rule PublishNotification : Notifications

```

rule PublishNotification(n) = ForwardNotification(n) where
  let context  $\leftarrow$  contextOfNotification(n) in
     $\forall$  node  $\in$  flowNodes(context) |
      flowNodeType(node)  $\in$  {"Message", "Signal", "Receive"} do

      if flowNodeType(node)  $\in$  {"Message", "Signal"}
         $\wedge$  name(trigger(node)) = name(triggerOfNotification(n))
         $\vee$  flowNodeType(node) = "Receive"

```

```

    ∧ name(message(node)) = name(triggerOfNotification(n))
  then
    if flowNodeType(node) ∈ {"Message", "Receive"} then
      nodeOfNotification(n) ← node
    else if flowNodeType(node) = "Signal" then
      let duplicate ← Clone(n) in
        nodeOfNotification(duplicate) ← node

    // Publish only yet unassigned notifications
    if nodeOfNotification(n) = undef then
      ∀ child ∈ contexts | parent(child) = context do
        let duplicate ← Clone(n) in
          contextOfNotification(duplicate) ← child
      remove n from notifications // lifetime ends

```

The abstract rule **Clone : Notifications** duplicates the original *notification* with preserving the original **timeOfOccurrence : Notifications**. Creating a new *notification* with the **new** construct would generate new timestamp.

Additionally the fact that all messages in a process have unique names is checked as shown in listing 6.

Listing 6. constraint UniqueMessageNames

```

∀ process ∈ Processes holds
  ∀ first ∈ flowNodes(process) |
    flowNodeType(first) = "Message" holds
    # second ∈ flowNodes(process) |
      flowNodeType(second) = "Message"
      ∧ name(second) = name(first)

```

7 Event propagation

Additionally, events from running process instances may be propagated [4, sec. 10.4.1] up to their innermost *context* containing an event which can catch them. If no such event is defined those *notifications* may be propagated up to their *Root Context* or to the common *Static Context*, e.g. error, escalation, cancel or terminate.

The corresponding *notifications* are created in the refined rule **Throw : FlowNodes × Instances[1]** shown in listing 7.

Listing 7. rule Throw : FlowNodes × Instances

```

rule Throw(node, instance) =
  let notification = new Notifications in
  choose context ∈ Contexts |
    instanceOfContext(context) = instance do
      contextOfNotification(notification) ← context

```

```
triggerOfNotification(notification) ← trigger(node)
```

If such a *notification* is not caught by any enclosing activity the process instance will terminate in case of error and terminate trigger[4, tab. 10.88]. In case of other triggers nothing will happen as this is defined as the common behavior in [4] and may be refined in a concrete implementation, as shown in listing 8.

Listing 8. rule PropagateNotification : Notifications

```
rule PropagateNotification(n) = ForwardNotification(n) where
  let context ← contextOfNotification(n) in
    let i ← instanceOfContext(context) in
      let boundary ← { b |
        b ∈ boundaryNodes(instantiatingFlowNode(i)) |
        flowNodeType(boundary) ∈ BoundaryEventTypes
        ∧ trigger(boundary) = triggerOfNotification(n) } in

        if |boundary| = 1 ∧ flowNodeType(boundary) ≠ "Cancel"
          nodeOfNotification(n) ← boundary
        else if |boundary| = 0
          local parent ← parentContext(context) in
            if parent ≠ undef
              contextOfNotification(n) ← parent
            else // No parent – Static Context
              if triggerOfNotification(n) = "Terminate"
                ∨ triggerOfNotification(n) = "Error"
                TerminateProcess
          else
            // Exception may be thrown in concrete impl.
```

8 Summary

The *notification* concept, shown in this paper, allows the Workflow Interpreter (WI) to control and observe the processes deployed to it. It implements the communication concept using messages or signals with other processes. It also enables the WI to instantiate new processes and to react to some events, e.g. unhandled error, escalation or terminate events. The *context* concept, defined in section 2, is used as a communication medium to forward the *notifications* to different parts of running processes and activities in the WI.

The event forwarding concepts[4] were formally defined in section 4, 6 and 7. The link events were left out since they are used only to break sequence flows to solve some graphical presentation limitations in the same process level[4, section 10.4.5] and this can be handled by the rule `WorkflowTransition : FlowNodes`[3].

In section 5 we have shown the creation of *notifications* for incoming messages and signals using pools. One can see the similarity between a message and a signal. The two apparent differences are that a message specifies a concrete target (i.e. catching event) and carries a more complex payload[4, figure 8.30, table 8.48], while a signal can be caught by any, and possibly more than one, catching event and does not carry any additional payload[4, figure 10.93]. Since we cannot model a message flow across different processes possibly running in different WIs we match the message target in section 6 using the name of the message. Another possibility is to define `messageFlowNameOfMessage` and match the target flow node by that parameter, since a message flow also specifies a name[4, figure 8.30]. We chose to match the target by the name of the message and not message flow to demonstrate the similarity between messages and signals and because message flow is a part of collaboration[4, section 9] which is not necessary to claim neither *Process Execution Conformance*[4, section 2.2] nor *BPEL Process Execution Conformance*[4, section 2.3].

The concepts presented in this paper, the *context concept* and the *notification concept*, are the core concepts currently used for our work-in-progress abstract specification of a Workflow Engine (WE).

Bibliography

- [1] Börger E, Sörensen O (2011) BPMN core modeling concepts: Inheritance-based execution semantics. In: Embley D, Thalheim B (eds) Handbook of Conceptual Modeling: Theory, Practice and Research Challenges, Springer Verlag
- [2] Börger E, Stärk R (2003) Abstract state machines: a method for high-level system design and analysis. Springer
- [3] Börger E, Thalheim B (2008) 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, Springer-Verlag, ABZ '08, pp 24–38
- [4] OMG (2011) Business process model and notation (BPMN) 2.0. www.omg.org/spec/BPMN/2.0