DENNIS WOLF

INVESTIGATING EVENT SUBSCRIPTION MECHANISMS IN BPMN

INVESTIGATING EVENT SUBSCRIPTION MECHANISMS IN BPMN

DENNIS WOLF

< Any Subtitle? >

August 2017 – version 1

Dennis Wolf: *Investigating Event Subscription Mechanisms in BPMN,* < Any Subtitle? >, © August 2017

ABSTRACT

Business Processes have become an essential tool in organizing, documenting and executing company workflows while Event Processing can be used as a powerful tool to increase their flexibility especially in destributed scenarios. The publish-subscribe paradigm is commonly used when communicating with complex event processing platforms, nevertheless prominent process modelling notations do not specify how to handle event subscription.

At the example of BPMN 2.0, the first part of this work illustrates the need for a flexible usage of event subscription in process models and derives new requirements for process modelling notations. An assessment of the coverage of these requirements in BPMN 2.0 is presented and shortcomings are pointed out.

Based on the identified requirements, this work presents a new concept for handling event subscription in business process management solutions, predominantly built on the notion of event buffers. The concept includes an extension to the BPMN meta model, specifies the semantics and API of a new event buffering module and describes the changes necessary to the behaviour of the process engine.

For evaluation purposes, the concept has been implemented as a reusable Camunda Process Engine Plugin that interacts with the academic Complex Event Processing Platform UNICORN.

ZUSAMMENFASSUNG

Kurze Zusammenfassung des Inhaltes in deutscher Sprache...

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

1 INTRODUCTION

2 BACKGROUND

- putting cep queries into bpmn models => heiko's thesis or other related work?

- throughout the chapters we are working with message events that are received via pub/sub from an external cep platform

- BPMN: always 2.0

- esper: probably 5.3

3 PROBLEM STATEMENT

This section will further define the problem and derive formal requirements to event subscription mechanisms

3.1 MOTIVATING EXAMPLES

- one example for independent. The subscription does not depend on a prior precess result, the subscription can be done even before process instantiation

- one example for a process that uses an intermediate event that depends (subscription-wise) on the result of a previous step in the process.

==> If the event occurs at a certain time, the process gets delayed unnecessarily or even run into a deadlock

3.2 EVENT OCCURRENCE SCENARIOS

Given the motivating examples, I am deriving a generic set of event occurrence scenarios. Each of these scenarios can occur in the real world and process implementations need to be capable of handling them to avoid negative effects.

TIME OF EVENT OCCURRENCE The most important variable to consider is the time of event occurrence. According to the BPMN specification, it is possible to catch an event if it occurs after the event element is enabled. As shown before, it is often impossible to control occurrence time and events do occur outside of these time windows. We specify the possible event occurrence times in relation to the life cycle of a process that utilizes a BPMN Intermediate Event

ref process lifecycle

.

Figure 1 shows the life cycle steps of a process and an instance from the deployment of the process until the undeployment and uses a timeline to illustrate that an event might occur at any time during this cycle. More precisely, an event is always considered to occur before or after a life cycle step or in between two consecutive steps.

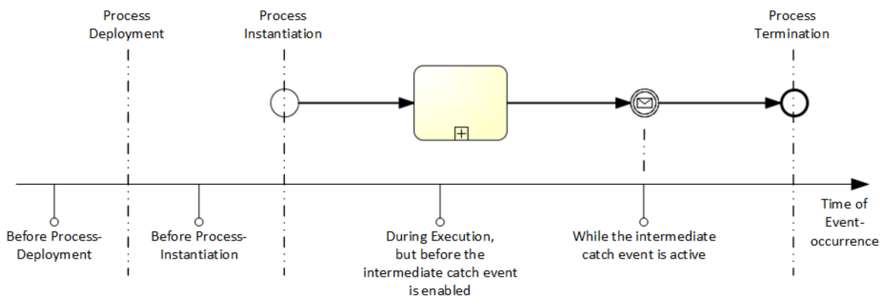

Process Deployment
Process Instantiation
Process Termination
Before Process-Deployment
Before Process-Instantiation
During Execution, but before the intermediate catch event is enabled
While the intermediate catch event is active
Time of Event-occurrence

Figure 1: Possible event occurrence times in relation to a process execution life cycle

How about system-deployment/process engine start and process undeployment? show in illustration, but say in text that we simplify this for now. after undeployment is essentially before deployment of a new process; Before Engine start is also before pr. deployment and we presume that an engine is running and does not stop.

Given the relevant life cycle steps, *process deployment*, *process instantiation* and *Event enablement*, the following occurrence scenarios are distinguished in this work:

A. O1 After the enabling of the BPMN event (BPMN default)

B. O2 The event does not occur

C. O3 Between Process instantiation and the enabling of the BPMN event

D. O4 Between Process deployment and process instantiation

E. O5 Before Process deployment

add a back reference to the examples? In example XY, events can occur before... whereas in example...

For a flexible and efficient use of events in business processes, it must be possible to use events that occur in any of these phases. To make sure that an event can be caught, no matter at which time during the phase it occurs, the subscription to the CEP platform must happen at the beginning of the occurrence phase. It follows that the event subscription must be possible at system start, at process deployment, at process instantiation, at any time during process execution and when the BPMN Event element is enabled.

EVENT SUBSCRIPTION DEPENDENCIES It is important to note that the subscription to an event source can depend on additional context information or process data. This can be a severe limitation to the possible subscription time.

ref to process model

shows a logistics process that uses event data about the GPS position of a certain truck to keep the estimated time of arrival of the transport updated. Whenever it receives an updated GPS position, the ETA is re-calculated; once the *arrival*-event has been received, the process finishes.

this example is not good, because we are not interested in a gps event that occurs earlier. Find an example where you would like earlier events, but subscription is not possible

Before the subscription to that specific truck gps event can happen, the process must determine the *truckId* to use in the event query. Only when the *truckId* is available, the subscription can be executed. This example illustrates how a query filter expression can depend on context data, but it might as well be the event source itself that differs depending on the particular execution.

there could be an xor gateway and following two different events and only one of them can get executed

solution would be to listen to all gps, but potentially too much data. Decision must be made cautiously! <= Where should I mention this? maybe later in the concept

3.3 REQUIREMENTS DEFINITION

The previous sections have exemplified how the execution semantic offered by the BPMN specification limits users in the use of events in business processes. Now these shortcomings are formalized into an additional set of requirements that must be met by a process execution environment to enable event handling in the extended set of event occurrence scenarios. The formal requirements will later be used to evaluate the capabilities of current Process Management Solutions and to develop a new concept to handling event subscription in business processes.

ref to chapters

R1: FLEXIBLE EVENT SUBSCRIPTION TIME

R1.1: Explicitness: For each event that is used in a business process, it must be possible to derive the time of event subscription from the process model. The time of subscription may either be explicitly stated or defined implicitly.

R1.2: Flexibility: The time of subscription can be influenced to catch events according to any of the event occurrence scenarios O1, O2, O3, O4. In other words, the process model defines the earliest acceptable time for an event occurrence to be considered in the process execution. The necessary options are *since system start*, *since process deployment*, *since process instantiation*, from an arbitrary but *explicit time during process execution*, or *since enabling of the Event Process Element*.

limited by subscription dependencies

change the options back to the times of subscription. Mention that the subscription is necessary before the time of event occurrence, but too early subscription is also a problem.

R2: AUTOMATIC SUBSCRIPTION HANDLING

R2.1: Subscription The subscription to event sources is handled implicitly by the process execution environment as defined by the process model.

R2.2: Removal of Subscription The removal of a subscription from the system is handled automatically as soon as a subscription becomes unnecessary.

R3: EVENT BUFFERING

To make all events since the subscription time available during process execution, matching events need to be stored temporarily.

buffer policies and scope?

4 ASSESSMENT OF CURRENT BUSINESS PROCESS MANAGEMENT SOLUTIONS

The lack of flexibility in handling event subscription in business processes has been outlined in the previous chapters and a set of extended requirements to process management solutions have been presented. In this section I take a closer look at the capabilities of current solutions with regards to the event occurrence scenarios to get a better understanding of the issues that arise when working with event subscription in business processes. The assessment will be carried out using BPMN and Camunda, a state-of-the art and widely adopted business process engine. The main goal is to identify and illustrate the shortcomings of the current process technology stack. These shortcomings will be referenced in addition to the presented requirements to develop a more refined subscription handling model in the following chapter.

"subscription handling model"?

which functionality should be evaluated exactly?: all occurrence scenarios, but no buffer policies. The buffer will always store the last version of the event and also deliver that version.

4.1 BPMN MODELS IN PRESENCE OF THE EVENT OCCURRENCE SCENARIOS

Chapter X has revealed that processes can run into deadlocks if events do not occur at the right time

.

Figure 2 shows a generalized process that uses an Intermediate Catch Event just before process termination. In this section I first describe for each Event Occurrence Scenario how this simple event implementation behaves in presence of the given scenario. I then evaluate if it is feasible to create a BPMN model that is free from deadlock in these situations.

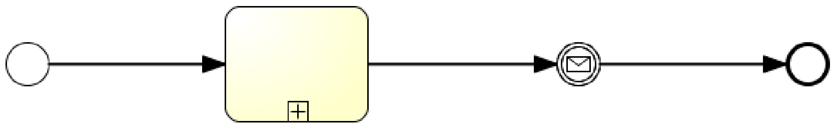


Figure 2: Abstract Process using an Intermediate Catch Event

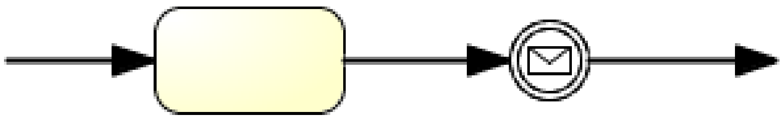


Figure 3: Standard Intermediate Catch Event

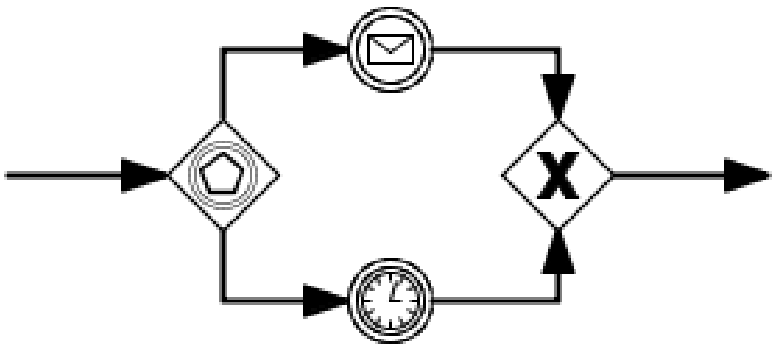


Figure 4: Intermediate Event with a parallel Timer Event

SCENARIO O1: THE EVENT OCCURS AFTER THE ENABLING OF THE BPMN EVENT The first scenario represents the most simple case, that is also natively supported by the BPMN 2.0 specification. When the event occurs after the Event element has been enabled, the event will be received and the process can proceed normally. The use of a standard Intermediate Catch Event does suffice to cover this situation.

SCENARIO O2: THE EVENT DOES NOT OCCUR In certain situations an event might not occur at all. Given a basic event implementation like in Figure 2, the process flow will get to a halt once it reaches the Intermediate Catch Event and will not be able to proceed. While, depending on the process design, this might be the desired behavior, in many situations this is not acceptable.

Let's consider a process that is supposed to wait for approval for a certain amount of time and trigger an additional request if the approval has not been issued before the deadline. Figure 4 shows how this behavior can be implemented using an Event-based Gateway which puts a Timer Event in parallel to the Intermediate Catch Event. This extension will make sure that a process does not run into a deadlock state if the expected event does not occur.

I mention an example, but that example is not exactly illustrated in the process

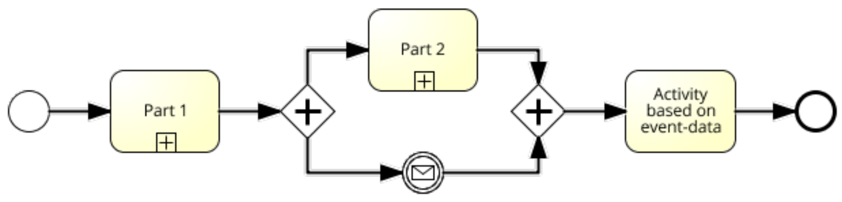


Figure 5: Event Element in parallel process flow

according to the spec: what exactly will happen to the active catch event once the timer fires?

SCENARIO O3: OCCURRENCE BETWEEN PROCESS INSTANTIATION AND THE ENABLING OF THE BPMN EVENT In case the event occurs during process execution, but before the BPMN event element is enabled and thus listening for events, the occurrence will not be considered in the execution. The process will get stuck at the Event Process Element as if the event did not happen at all. To avoid a deadlock in this scenario, a solution is to execute the Intermediate Catch Event in parallel to the rest of the process flow using a Parallel Gateway. This is illustrated in Figure 5. The time of subscription to the event can be controlled by the position of the parallel split: To implement an event subscription right after process instantiation, the Parallel Gateway has to be the first element after the Start Event (that means *Part 1* in the illustration is empty). To implement event subscription at a specific point during process execution, part of the process can execute before reaching the Parallel Gateway. In Figure 5, the event may occur at any time during the execution of the collapsed sub-process *Part 2*.

SCENARIOS O4 AND O5: BEFORE PROCESS INSTANTIATION Any Events that happen before process instantiation will not be considered in a standard Intermediate Catch Event. That applies to both scenarios, the occurrence between deployment and instantiation (*O4*) and an occurrence time before the deployment of the process in the Process Engine (*O5*).

To create a Process Model that allows to catch an event before the process instance exists, three new elements are introduced: (1) An additional *Auxiliary Buffering Process* that can catch an incoming event, (2) an *Event Buffer*, a temporary data-store that keeps event data until required by the *Original Process*, (3) an *Auxiliary Event Delivery Process*, that retrieves events from the buffer and makes them available to the *Original Process*. Figure 6 reveals the interaction of the *Original Process*, the two auxiliary processes and the data-store. To start listening for an event, the *Auxiliary Buffering Process* has to be instantiated through

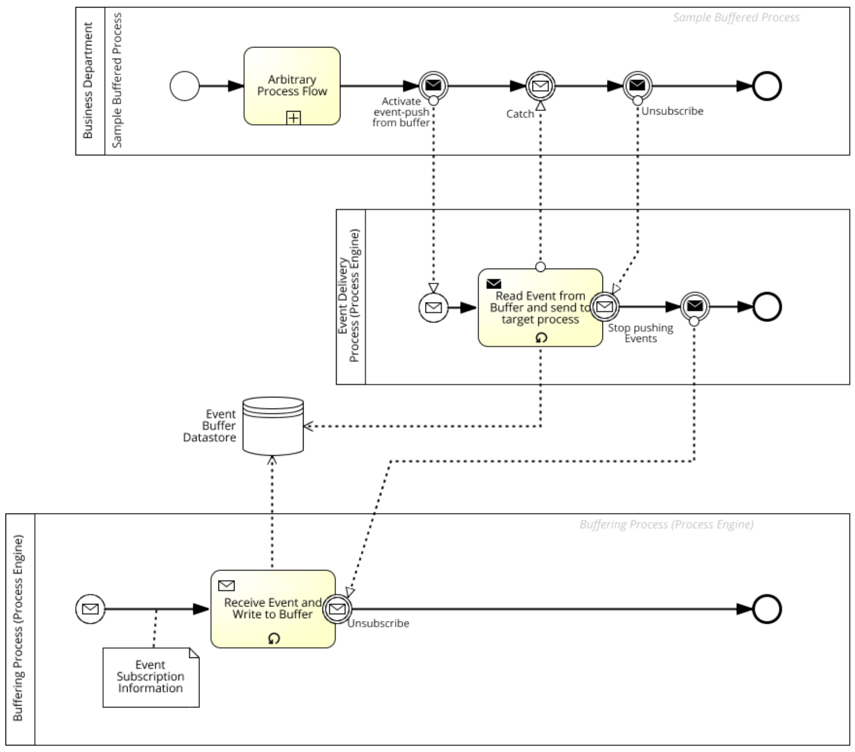

Sample Buffered Process
Business Department
Sample Buffered Process
Arbitrary Process Flow
Activate event-push from buffer
Catch
Unsubscribe
Event Delivery Process (Process Engine)
Read Event from Buffer and send to target process
Stop pushing Events
Event Buffer Datastore
Buffering Process (Process Engine)
Buffering Process (Process Engine)
Receive Event and Write to Buffer
Unsubscribe
Event Subscription Information

Figure 6: Event Buffering through an auxiliary Buffering Process

the a message start event containing the information necessary for the event subscription. The process starts listening for the event and writes the received event to the temporary data-store. The given process design is able to handle multiple event occurrences, because the receiving activity is looping. The buffering process terminates once the *Unsubscribe* event is received.

how should the ABP be started? manually?

The *Original Process* can be started any time after the buffering process. In Figure 6, the Intermediate Catch Event has been explicitly split into three events: An initial Send Event to request events, a Catch Event to receive and a final Send Event to signal that no events shall be received anymore. The initial Send Event instantiates the *Auxiliary Event Delivery Process*, which tries to read from the Event Buffer and deliver the event to the Original Process. Once there is data available in the buffer, it is sent by the sending activity. The central looping activity will retry reading from the buffer until data becomes available and will only be terminated once the *Stop*-event occurs. The Original Process can receive the event using a standard Intermediate Catch Event even when the event occurs before the instantiation of *Original Process*, so it handles scenario O4. Moreover the *Auxiliary Buffering Process* is not bound to a specific event, it works generically with any event information that is passed on to it. For that reason it is also not bound to a specific process deployment and can buffer events even before a process has been deployed, so it handles scenario *O5*. Given that the buffering process can alternatively be started using an explicit Message Send Event during process execution and the process does not stop listening until the Original Process has received the event, scenarios *O1* and *O3* are also supported.

what exactly is passed around | how many instances of each process | overwrite or append to buffer | this is only one solution to do this, one that requires minimal changes in the original process

4.2 IMPLEMENTION OF EARLY EVENT SUBSCRIPTION USING STANDARD CAMUNDA

The previous chapter has shown that that it is possible to create BPMN models to match each of the Event Occurrence Scenarios, though for the scenarios *O4* and *O5* the solution becomes increasingly complex. In the next step I investigate the capabilities of Camunda, a modern and actively developed Business Process Engine that is available under an open source license. Camunda shall be used without any code customization, that means as offered on the website. The solution presented for the last two scenarios has proven capable enough to handle all Event Occurrence Scenarios, therefor the goal is to implement this solution. It will be necessary to create the two auxiliary processes and a data-store in addition to the original process that makes use of the event buffering.

Two generic sample processes have been modeled for demonstration purposes. Figure 7 shows a simple process with an explicit subscription activity to represent the listening to the event after process instantiation but before reaching the Catch Event (Scenario *O3*). It follows a sample activity that takes 15 seconds (implemented using a *Script Task*), the Intermediate Catch Event and another Script Task that displays the content of the received message. The example for scenarios *O4* and *O5* (see Figure 8) comprises the following elements: After the start event follows an Intermediate Catch Event, then an activity that prints the message of the event to console and last the Process End Event. Both figures show the processes as modeled in the Camunda Modeler.

AUXILIARY BUFFERING PROCESS The task of this process is to subscribe to a CEP Platform using a provided event query and start listening for events. Any incoming event must be stored in a data-store (*Event Buffer*). UNICORN, an Esper-based academic event processing platform, will be used in this example. A local MySQL database has been chosen for buffering the event data because it's freely available, quick to set up, offers standardized access via SQL queries and Java

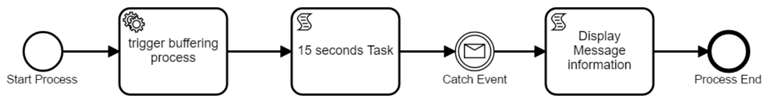


Figure 7: Generic Example Process in Camunda for Occurrence Scenario *O3*

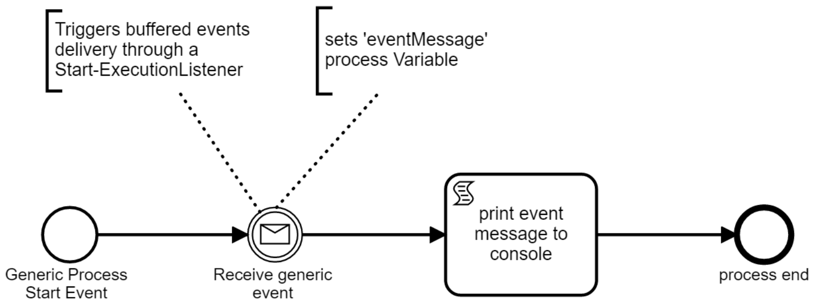

Triggers buffered events delivery through a Start-ExecutionListener
sets 'eventMessage' process Variable
Generic Process Start Event
Receive generic event
print event message to console
process end

Figure 8: Generic Example Process in Camunda for Occurrence Scenarios *O4* and *O5*

connectors and will persist data to the local harddrive by default. As UNICORN also requires an SQL database, the MySQL instance can be used in both cases.

Figure 9 shows the final Buffering Process modeled in the Camunda Process Modeler. The process can be instantiated by issuing a *Buffering Task* message. This message must contain three data fields: *processDefinitionId*, to know which process definition the buffered messages belong to; *messageName*, the name of the message event within the process; *query*, the event query in the Esper Query Language. Camunda will make the message data automatically available in the process instance as process variables, so they can be used during the execution of the Buffering Process. After instantiation, the process reaches the activity *Subscribe to Event Source*, a *Java Service Task* that executes a HTTP call to the UNICORN platform. That call registers the event query in UNICORN.

it tells unicorn its own instanceId, so that unicorn can correlate events to that exact instance

Afterwards, the process reaches the receiving activity *Wait for unsubscribe event* that will terminate the process as soon as the *Unsubscribe* event has been received. As long as this activity is active, events can be received through the attached Non-Interrupting Boundary Event. Incoming events have a field *eventBody*, which contains the

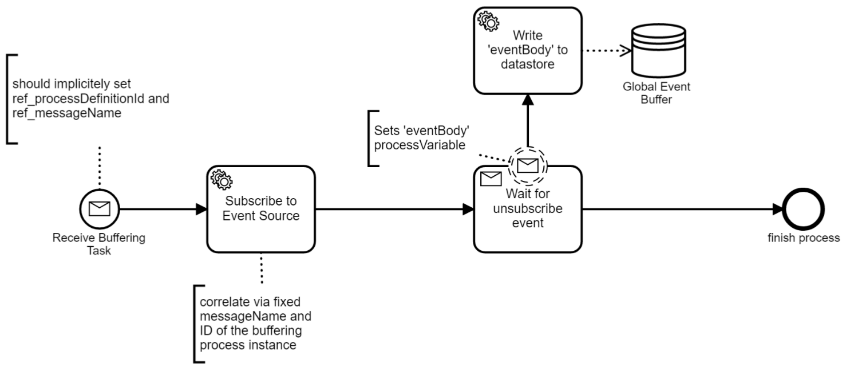

should implicitely set ref_processDefinitionId and ref_messageName
Receive Buffering Task
Subscribe to Event Source
correlate via fixed messageName and ID of the buffering process instance
Sets 'eventBody' processVariable
Write 'eventBody' to datastore
Global Event Buffer
Wait for unsubscribe event
finish process

Figure 9: Auxiliary Buffering Process in the Camunda Modeler

event information and becomes available through a process variable with the same name.

does the message from UC have a field 'eventBody'?

The boundary event triggers the service task *Write eventBody to datastore*, which takes the data from the process variable and writes it to the MySQL Database Instance (*Global Event Buffer*).

AUXILIARY EVENT DELIVERY PROCESS The delivery process (see Figure 10) reads the latest data from the buffer and sends it to the process instance. It can be started with a message that contains the *processInstanceId* and the *processDefinitionId* of the requesting process and the *messageName* of the Message Event that is requested from the buffer. A *Delay* Timer Event has been inserted to make sure that the receiving process is already in listening state, the execution happens asynchronously. It follows the service task *Retrieve event from buffer*, which executes Java code to read from the MySQL Database *Event Buffer* and store the event information in a process variable named *eventMessage*. The content of that process variable is sent to the *Original Process* in the Send Event, afterwards the execution is finished.

INTERACTION OF THE PROCESSES In this implementation of flexible event subscription, the action of subscribing to the event source and the reception of events in the Original Process are splitted into two separate parts, each supported by an auxiliary process. To initiate the subscription at the event source, the Auxiliary Event Buffering process has to be started. For scenario *O3*, this happens through an extra activity (*Trigger Buffering Process*) during process execution, so that events after process instantiation are received by the Buffering Process. In scenarios *O4* and *O5*, the subscription and thus the instantiation of the Buffering Process must happen before the instantiation of the Target Process. As there is no such mechanism in the standard Camunda Process Engine, the Buffering Process must be started by

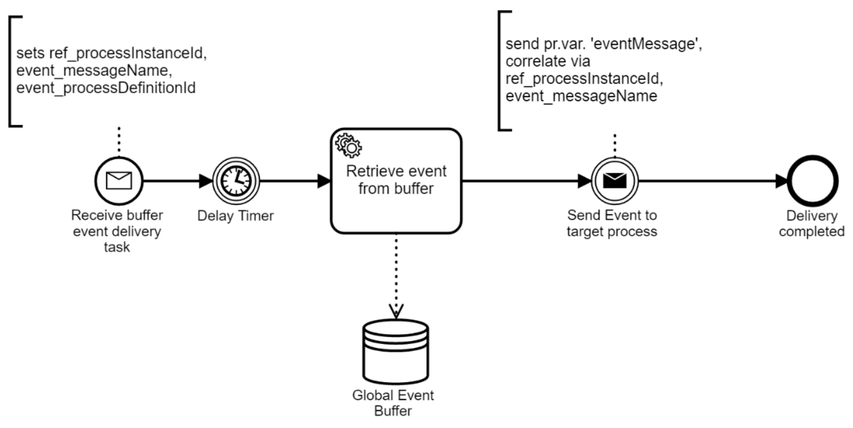

sets ref_processInstanceId, event_messageName, event_processDefinitionId
send pr.var. 'eventMessage', correlate via ref_processInstanceId, event_messageName
Receive buffer event delivery task
Delay Timer
Retrieve event from buffer
Send Event to target process
Delivery completed
Global Event Buffer

Figure 10: Auxiliary Event Delivery Process in Camunda Modeler

hand, providing the *processDefinitionId*, the *messageName* and the *eventQuery*.

Now that the *Buffering Process* is running, any events matching the query will be stored to the buffer. When the Target Process reaches the Catch Event, a request for buffered events is sent as a message to trigger the *Auxiliary Event Delivery Process*. This message is sent using a short piece of Java code that gets executed when the Catch Event is reached. The code is invoked by a Start ExecutionListener attached to the Catch Event. ExecutionListeners are offered by Camunda to execute own Java programs before or after relevant events during process execution, like the execution of an element in the process. While the Original Process will now start listening for the desired events, the Event Delivery Process will send the buffered events as messages to the Original Process.

If no events have been received yet, all the involved processes remain active: the Buffering Process will keep listening for an external event. The Delivery Process will send an event to the Original Process as soon as there is one in the buffer. The Catch Event in the Original Process will keep listening for an Event.

the termination of the processes is not yet implemented in Camunda

get wording straight: Original process, target process, requesting process, main process | also always italic or never

Fehlende Abbildung

overview of the 3 processes, mysql and UNICORN note that this is an investigative implementation that matches exactly the given use-case and is not meant to be used in production. It is neither flexible nor robust enough for that purpose, but suits very well in understanding the capabilities and the shortcomings of BPMN and Camunda when it comes to handling the Event Occurrence Scenarios

4.3 DISCUSSION

The goal of this chapter was to get a better understanding of the capabilities of the tools when it comes to covering all event occurrence scenarios. Even though is has proven possible to to implement a flexible event subscription time using standard BPMN 2.0 and Camunda, the success comes at a cost. The downsides of the presented approach are presented in the following.

It was necessary to create two generic auxiliary processes for event buffering, to connect to a MySQL data-store and use ExecutionListeners to execute custom Java code in Camunda to cover all scenarios, *O1* to *O5*.

give them short names for reference?

NO AUTOMATIC SUBSCRIPTION HANDLING

In the presented process models, separate process elements had to be added to handle event subscription and initiate event delivery. That conflicts with requirement *R2*, which states that the subscription and un-subscription must be automatically handled by the process engine. For the scenarios *O4* and *O5* the Buffering Process has to be triggered manually, because it must be executed before the target process is running. Camunda does not handle external event subscription itself, especially not before the process is running.

MORE COMPLEX PROCESS MODELS

As additional process elements have to be added to handle event subscription and delivery, the models become more complicated and are less concentrated on the business case.

there could be a requirement that states that there should be no additional process elements unless there is an explicit subscription time

BUFFERING IS AN IT TASK

The auxiliary processes are not business tasks and are thus not suited to be modeled in BPMN. Desired functionality can be put into Camunda BPMN models thanks to its flexibility to use Java code in Service Tasks or Event Listeners, but naturally the full functionality of the Event Buffer cannot be expressed using BPMN.

what exactly is the issue here?

ADDITIONAL LOAD ON THE PROCESS ENGINE

Because of the aux processes, two additional processes have to be deployed in the process engine and are potentially running in parallel to any given process instance. For each Event Element used in a process the engine has to run an instance of the Buffering Process and, eventually, an instance of the Buffer Delivery Process. That puts additional load on the process engine, which might prevent business critical processes from executing delay-free.

Even when the number of deployed and running auxiliary processes can be reduced through further optimizations there remains an event-management overhead as every event has to be handled twice: once when it is stored in the buffer and once when it's delivered to the target process.

HIDDEN PERFORMANCE LIMITATIONS OF THE PROCESS ENGINE

Given the large amount and high frequency in that events can occur in reality, optimal performance is required for an event-buffering module. Running essential parts of the buffering within the process engine might pose performance limitations that cannot be influenced without tempering with the process engine code.

5 FLEXIBLE EVENT SUBSCRIPTION IN BPMN

Introduce a new concept. three pillars: bpmn-x, process engine behavior, event handling api

5.1 BPMN EXTENSION

this is the explanation for the process designer

Given the additional requirements and the shortcomings identified in the previous sections, the following two chapters present an extension to the BPMN event handling model. At first, an extension to the Business Process Model and Notation (BPMN) is described, which aims at providing the Process Designer with more flexible Event Handling capabilities according to Requirement *R1*. Afterwards, Chapter 6 clarifies the changes necessary to the event handling platform and the process engine to cover Requirements *R2* and *R3*. While the presented concepts are kept as general as possible, they are grounded in an analysis of the Esper-based CEP Platform Unicorn and the open-source process Engine Camunda.

To allow the flexible use of event subscription in BPMN models, a number of additional attributes must be added to the model. The extension should cover the Intermediate Catch Event, the Boundary Catch Event and the Receive Task, all three can be used to model the receiving of messages in BPMN.

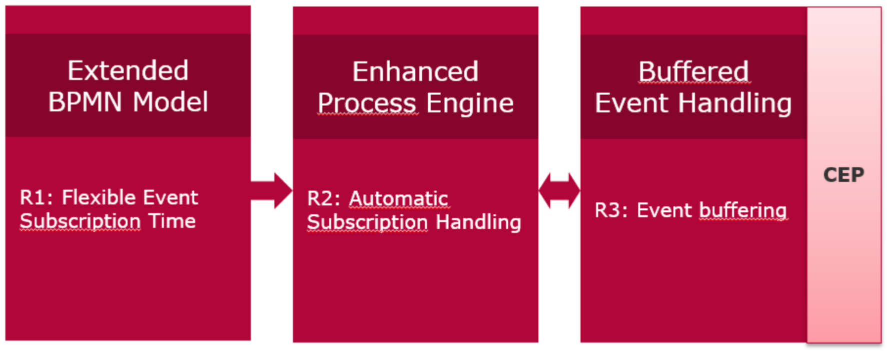


Figure 11: The concept for flexible event subscription involves three modules: A BPMN extension, enhanced process engine behavior and buffered event handling.

Is it ok to reference the receiveTask as well? It might have different execution semantics as it doesn't reference the MessageReceiveEvent, but the Message directly.

To cover all three elements, the extension will be attached to the BPMN type *tMessage*.

a new <element> or how do we extend?

According to the plain specification, the message type comprises an attribute *name*, the name of the message, and *itemRef*, the reference to a BPMN *Item* definition.

explain 'item'?

In the following, the required additional attributes will be explained one after the other. The goal is to retain a stand-alone model that contains all information necessary to execute the subscription to the event source.

add table overview of all extension fields with some details. Look that it doesnt double with the table-todo in automatic-subscription

5.1.1 *Adding basic subscription information*

For a basic event subscription, an event query, the platform address and optionally authorization information of the CEP Platform is required.

as explained in background

We work with the following simplified process

Assuming that there is one CEP platform for all events and processes, the latter two, i. e. the basic platform information, are configured centrally for the current process execution environment. Hence they don't need to be specified for every given message element.

that means that in the implementation we need additional configuration values -> implementation chapter

The event query instead needs to be specified for every message and is added to the model as an extension attribute *eventQuery* of type string, which should contain the full query as interpretable by the CEP platform. A similar approach has been taken by X and Y, who aim at enriching BPMN models with subscription information without considering the time of subscription specifically.

Find this source; They add fields ... to element ..., their primary goal is ...

Given this fundamental part of the BPMN extension, it is possible to execute the subscription, but the time of subscription cannot be influenced.

5.1.2 *The time of event subscription modeled in BPMN*

This section specifically addresses the requirement *R1.1*, aiming to provide a flexible event subscription time to be selected for each BPMN message when designing the event-driven process. Two different tools are to be offered to support all subscription times demanded by *R1.1*: Firstly, the subscription can happen implicitly at certain process-related points in time. Alternatively, the subscription can be modeled explicitly as a flow-element in the process. It is the task of the process designer to elaborate the correct time of subscription necessary for her use case.

The subscription will be executed automatically by the system based on the information given in the BPMN model. Further information on the exact execution flow is provided in chapter XY.

add reference. in automatic subscription handling: say for each of the cases when exactly the subscription is executed.

Any event message that occurs before reaching the event element but after the time of subscription will be kept in a buffer by the system. In its simplest version, the buffer is of length 1, that means it stores exactly one message received from a CEP platform. It always stores the latest message. When a newer message arrives, the old one is replaced in the buffer.

By default, there is no interference between the buffers of different messages, process instances or processes. Each buffer instance will contain the latest information as if it was the only buffer in the system. Performance improvements to avoid duplicate buffer content will be managed by the system without explicit action by the user. Section ... later introduces a shared, more complex usage scenario of the event buffers.

maybe make a separate paragraph somewhere here?

Modern event query languages are feature-rich and offer a large set of expressions to filter events from incoming streams.

ref background

The introduced basic event buffer can be used in connection with any desired event query and will store the latest output of that query. These two features together suffice to implement even more complex use-cases: Query windows of length *n* can be used to keep multiple events in the buffer, filter expressions allow to keep a subset of all events based on their attribute values, multiple streams can be joined together.

As soon as the process flow reaches an event element, the latest CEP message is retrieved from the buffer. It is not consumed, that means a second event element that references the same BPMN Message will reuse the information from the buffer. If no information is available in the buffer, the flow element will remain in the waiting state until a message is received. Then, the process flow proceeds as usual.

reorganize this chapter, maybe there should be an extra part for the buffer introduction/definition

SUBSCRIPTION TIME AS PART OF THE BPMN MESSAGE ELEMENT

To provide the Process Designer with a simple but powerful tool to influence the time of event subscription, a field *subscriptionTime* is added the BPMN message element. The field is implemented as an enumeration and can have one of the following four values: *System Start*, *Process Deployment*, *Process Instantiation*, *Event reached*. The last option is the default option according to the BPMN specification.

enumeration is fine?

Note that a *subscriptionTime* set to *Event reached* will remain without effect if an explicit subscription task for the same event was executed before the event is reached.

what does system start mean?

For each of the options: Define exactly (according to BPMN spec or standard literature), when in the flow the subscription is executed.

In motivating Example Ex, it is necessary to issue the subscription as early as possible, to make sure that data is available and the process execution is not delayed.

which example to reference?

Using the BPMN extension, the use case can be implemented by defining the event query and the subscription time in the BPMN model.

show what that would look like in the example. Maybe some XML?

THE EXPLICIT SUBSCRIPTION TASK

As an alternative to specifying the subscription time using the extension field *subscriptionTime*, a new kind of activity is added to BPMN. It can be used to place the action of subscribing to an event source anywhere in the process flow. The activity is called *Explicit Subscription Task* and implemented by extending the BPMN Service Task. A field *messageId* is added to the service task to establish a reference between the activity and the message definition. As introduced in section Section 5.1.1, the extended BPMN Message definition contains the information necessary to issue the subscription to an event source. Once the Explicit Subscription Task is activated, the subscription for the referenced message is issued.

Modeling the event subscription in an explicit task can be necessary when the subscription depends on the result of another activity. In that case, the subscription cannot be issued on process instantiation, but early subscription might be possible using the explicit subscription task. Apart from this particular use case, the explicit subscription task enables the Process Designer to place the subscription flexibly in the process flow and give her full control over the time of subscription.

variation in notation from other paper, idea is similar

As an improvement to the options for subscription time, there could be an option "ASAP", so that the process engine issues the subscription automatically as soon as the required process data becomes available

If both tools, the extension field *subscriptionTime* and the explicit subscription task, are used for a single BPMN message, the earlier subscription of the two will be executed, the second subscription will have no effect. That means for example if the *subscriptionTime* is set to *event reached* and an explicit subscription task is inserted before the event element, then the subscription will be executed at the time the explicit subscription task is active. If *subscriptionTime* is set to *Process Deployment*, then the subscription will happen at that time and the explicit subscription task will remain without effect. In case neither of the two is used, the system falls back to the BPMN default and executes the subscription when the event element is reached.

5.1.3 *Using Process Variables in Event Queries*

this could be added in the requirements and referenced

As shown in example Z, it can be the case that the values of process variables shall be dynamically used in an event query. Therefore, the name of the process variable should be part of the event query. At the time of subscription, the mentioned variable is dynamically replaced by its current value. The exact notation for including process variables in event queries can vary depending on the applied query language as it may not interfere with any existing notation schemes. For the use with the Esper query language, the following is suggested: The exact name of the variable has to be surrounded by curly brackets and preceded by a # character: *#{VARIABLENAME}*. This notation is inspired by the usage of substitution parameters in SQL queries that are embedded in Esper. They take the form *${expression}*.

reference esper docs 5.13.1. Joining SQL Query Results

define the esper version somewhere

In the example, the process uses the latest GPS position for a certain truck. The truck is identified by its unique ID which is part of the query: *SELECT lat, lng from GPSUPDATE where truckid = #{truckid}* .

missingref: dependent example

The use of dynamic process variable values introduces an additional complexity: Depending on the time of event subscription, the value of the process variable might not yet be available.

What does this mean for the process designer? A model that can take a state in that a subscription shall be issued, though the data is unavailable, is invalid. When will an error occur?

5.1.4 *Advanced Buffer Parameters*

Inspired by Mandal et al., a number of advanced buffer attributes are available through an extension attribute *BufferPolicies*.

add ref to sankalitas paper; do I elaborate on the changes made in comparison to the paper? + the reasons?

LIFETIMEPOLICY

The first attribute is the *LifetimePolicy*, which allows to specify after which timespan elements in the buffer should be deleted. Timespans shall be defined using ISO timespan format.

ref

The default value is *infinite*. Example A has been implemented by setting the *subscriptionTime* to *Process Deployment*, which means that there can be an infinite time difference between the action of subscribing to the event source and the reaching of the event element in one of the instances. In case events are not published in a longer time, for example due to technical fault at the event producer, the buffer will contain older events that might not be relevant anymore. Using the *LifetimePolicy*, the process designer can express, that events should be deleted from the buffer after a certain period of time and thus avoid outdated information. The buffer is maintained automatically by the system. That of course comes at the price that the process has to remain in waiting state until a new event message arrives.

CONSUMPTION BEHAVIOR

So far, the event buffers can be used isolated from each other. There is no interference between buffer instances and events are not removed from the buffer after retrieval. While for most use-cases this behavior is sufficient, more detailed control over the buffer can be desirable when a given message shall be used multiple times. Not always is it wanted, that events remain in the buffer after retrieval. An additional parameter *ConsumptionPolicy* is introduced which can take the values *Consume*, *Reuse*(default) and *Bounded Reuse(n)*. While *Reuse* denotes the behavior that is already known, *Bounded Reuse(n)* will allow an element to be retrieved exactly *n* times. *n* has to be replaced by an integer value greater 0. The option *Consume* will remove an element from the buffer immediately after it has been retrieved for the first time, it is therefor equivalent to *Bounded Reuse(1)*.

- given the option to consume from the buffer, it will now make a difference if the same buffer is accessed multiple times. - there are two scenarios to access the same buffer: (1) multiple times in the same instance, (2) multiple times because of parallel instances, (3) multiple times because of a shared buffer across processes - before proceding, we need to be clear about the buffer scope: it depends on the time of subscription. (1) after instantiation: buffer only instance-wide; (2) on pr depl: buffer reused across all instances; (3) system start: buffer reused across all processes - two additional buffer policies: *SizePolicy*, *OrderPolicy*, default values

write text for given keypoints

- what if messages are defined in different ways? => I want to reuse the buffer between processes, but the message definition (especially cep query) are not the same. how will the system behave?

5.2 DESIGN DECICIONS

The target functionality of the BPMN extension was clearly defined by the identified requirements. To implement that functionality, there were a number of options to consider and design decisions to make. This chapter provides background information on the decisions that have influenced the presented concept for flexible event subscription.

Can I put together an alternative solution? Table of events for a certain event source. The buffer is already there for me to pick from when designing the process.

THE TIME OF SUBSCRIPTION IS A QUESTION OF PROCESS DESIGN

-> process designer

Is there an additional event engineer?

- this is why the bpmn extension is presented first - hide complexity from the designer

THE ACTUAL BUFFER IS MOSTLY HIDDEN FROM THE USER - Buffers are implicitly defined through the BPMN model - keep the look and feel of the message catch event - minimal changes to existing models, backwards compatibility

AVOID ADDITIONAL USER INTERFACES - the user will only use the bpmn model - we don't want any other element because of complexity - the process should be self-contained, contain all necessary information for subscription and buffering. -> single point of contact

BUFFERS ARE CLOSELY LINKED TO PROCESS MODELS Messages are only buffered as soon as they are explicitly required by a model - we don't just buffer n messages because we might need them in the future. That would be a fuzzy, incalculable performance overhead. - instead we keep as little as possible in the buffer

THE BPMN EXTENSION IS BASED ON THE MESSAGE ELEMENT - if I want to talk about related work that goes another way - why are the policies a parameter of the message and not the catch event element?

REUSE EXISTING TECHNOLOGY -> we assume that a cep is present and that basic features of event queries can be used - if not present, then only very basic buffer functionality is available - but we dont want to start designing another event processing layer with duplicated functionality

6 AUTOMATIC SUBSCRIPTION HANDLING

After defining the functionality provided to the user of flexible event subscription, this chapter describes the changes necessary to the software infrastructure that is used for event-driven business process management. The concept requires that all subscription and event handling is executed by the system itself, without further interaction by the user. All necessary information for that purpose is provided by the BPMN model.

As described in REF, an event-driven process management setup primarily consists of ...

missingref

Changes are necessary to both, the Event Processing Module and the Process Engine. This chapter attempts to keep the change descriptions general so they can be applied to any common process engine and event processing platform. The first section describes the necessary extension to the event processing to support early subscription and event buffering. The following Section 6.2 specifies the changes necessary to the behavior of the process engine as the connecting element between the BPMN model and the event processing platform.

6.1 BUFFERED EVENT PROCESSING

When reduced to the basics, a standard event processing platform works as follows: The user subscribes to events providing an event query and a notification-path. The platform responds with a unique identifier for that subscription. Whenever an event occurs that matches the provided query, the platform issues a notification to the notification-path. Subscriptions can be deleted through their unique identifier. These two operations, *subscribe* and *unsubscribe*, make the fundamental API of a CEP platform.

ref

EVALUATION OF COMMON CEP PLATFORMS - looking at the new bpmn extension reveals that a different behavior is required: notifications need to be kept in a buffer until they are requested by an entity - three event platforms were studied to check if the required functionality can be implemented natively: wso2, esper and ... - wso2: ? - esper: - ? : ? - a window will achieve something similar, but not exactly the same - moreover, it is desired that the same full-featured event queries as before can be used, no restrictions. - as functionality wasn't available in any of the three, it was decided to specify an extended api.

Maybe: we consider these two operations as given, because common event processing platforms have these in common, but there is not common event buffering concept yet. even though e.g. esper has something which goes in that direction: output clauses

AN API FOR BUFFERED EVENT PROCESSING The novel BPMN extension for flexible event subscription allows to issue a subscription for buffering well before the events ought to be delivered via the notification-path. The introduction of an event buffer as a separate entity between the varying list of notification-recipients and the event query makes an extension of the API necessary. Firstly, the *subscribe* operation has to be divided into two steps:

A. *registerQuery(queryString[, bufferPolicies]): queryId*
The call registers an event query in the CEP platform and instantiates a buffer. Matching events will be held in the buffer according to the specified policies. It returns a unique identifier to that new query registration and hence for the connected buffer. That identifier must be used to modify the query later.

bufferPolicies is an optional parameter which is provided as an object with four possible fields: *LifetimePolicy*, *ConsumptionPolicy*, *SizePolicy*, *OrderPolicy*. Refer to Section 5.1.4 for a detailed specification of the semantics of the parameters. If *bufferPolicies* is not or only partly specified, the system should fall back to the default values.

B. *requestEvents(queryId, notificationPath)*
Initiates the delivery of notifications for a given queryId to a notification recipient. The recipient is specified through the *notificationPath*, the full address of the entity that is supposed to receive the message. Notifications are delivered asynchronously as soon as they are available. If the buffer is not empty, a message will be sent right after the *requestEvents* call. A similar operation, *addNotificationRecipient*, is available in existing CEP platforms. The difference is in the delivery of the first buffered message: *requestEvents* sends out the message from the buffer, *addNotificationRecipient* will send out notifications only for future query output.

be clearer about notificationPath. Specify in background and reference. also about addNotificationRecipient

add table with buffer policies, their possible values and the default val improve explanations

A similar situation holds for the un-subscribe operation: Traditionally, a subscription is canceled through a unique identifier that is obtained as a result of the subscribe operation. After cancellation, no more notifications are delivered, the query is removed from the system. Given flexible event subscription, this operation must be split into two parts as well:

c. *unsubscribe(queryId, notificationPath)*
Removes a notification-recipient for a given query-id. Note that the buffer and query instance remain intact, so that other recipients can still subscribe.

d. *removeQuery(queryId)*
Completely deletes the query and its buffer, so that no notifications are sent out any longer.

All four methods are required to execute a subscription on process deployment. The query must be registered using *registerQuery* before the process instance, i. e. notification-path, is available. For each process instance, events can be requested individually using *requestEvents* and thereafter, the notification-recipient can be removed with *unsubscribe*. The query and its buffer will remain active even after any single instance has terminated. When the process gets un-deployed, the query can be deleted using *remove query*. Section 6.2 describes the steps in detail.

show the steps with sample data from one of the examples

- to achieve the default behavior, each two steps have to be executed just after the other.

Fehlende Abbildung

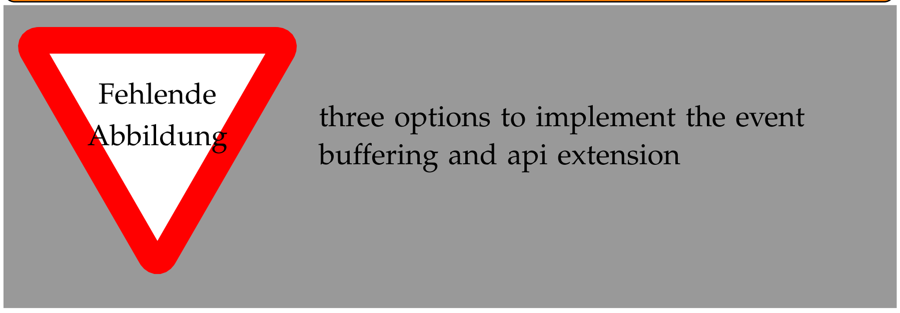
maybe a uml sequence diagram. Or other diagram format?

maybe provide a swagger definition for this?

what kind of API should this be? REST? Java? none specifically, but the reader might want clarification when reading this.

different options to implement the api extension That extension can either be implemented by adopting the CEP platform itself, by implementing a separate middleware between process engine and CEP platform, or by implementing a buffering module as part of the process engine. Which of the three options suits best has to be evaluated for the given use-case and existing infrastructure. In some cases it might not be possible to adapt the code of process engine or event processing platform, which leaves a separate middleware as the only choice.

Generally, extending the event processing platform is advisable? Or put up brief pros and cons of each of the options



three options to implement the event buffering and api extension

A reference implementation for an extended complex event processing platform is presented in Section 7.1 at the example of the Esper-based CEP platform *Unicorn*. It also explains, why extending the event platform was the preferred choice in the given scenario.

PERFORMANCE CONSIDERATIONS

todo

- performance improvements through shared windows - due to its performance optimizations, an extension of the cep itself would make perfect sense - given the extended api, it is now possible to implement early event subscription from the process engine

6.2 EXTENDED PROCESS ENGINE BEHAVIOR

It is the task of the Business Process Engine to interpret and execute process models and connect to an event processing platform in event-driven setups. From the three relevant parts, two have already been defined, the BPMN extension and the buffered event processing module. Out of the box, a process engine like Camunda will ignore any proprietary BPMN extensions and the subscription to an event source must be especially implemented. An example for such an implementation is provided in Section 4.2. One goal of this work is to automatize the handling of event subscriptions solely based on the information available through the extended BPMN model. Additional process elements should not be required. This section will clarify, which operations need to be executed by the process engine to enable the automatic subscription handling. Section 7.2 demonstrates the implementation of automatic subscription handling at the example of Camunda.

PARSING ADDITIONAL INFORMATION FROM THE BPMN MODEL It is required that the process engine is able to read the additional information from the BPMN extension (see Section 5.1) so that it is available during process deployment and execution. This affects the BPMN message element, which can contain the additional attributes *eventQuery*, *subscriptionTime* and *bufferPolicies*. Secondly, the *Explicit Subscription Task* has to be processed. It contains a reference to a Message entity within the same model. The process engine might have to be adopted to read all relevant data from the extended model.

MANAGING SUBSCRIPTION AND UN-SUBSCRIPTION As defined in the BPMN extension for flexible event subscription, the action of subscribing to an event source can happen at different times during process deployment and execution. The options and the implicit timing of subscription and un-subscription are specified in Section 5.1.2. The process engine must communicate with the process engine using the four calls *registerQuery*, *requestEvents*, *unsubscribe* and *deleteQuery*, that were presented in the previous chapter. For each possible subscription time, the following briefly enumerates which operations must be executed when.

IN EVERY CASE: The return-value of *registerQuery*, a unique identifier of that query, must be stored for the related BPMN *Message*. The id is later necessary to execute the other three API methods. When an event element is reached, a call to *requestEvents* must be issued. When the execution of that event element is finished, call *unsubscribe*.

SUBSCR. ON PROCESS DEPLOYMENT: When a process gets deployed, the process engine must check if subscription information is in the model. For every *Message* element that is set as *subscribe on pr. deployment*, a call to *registerQuery* must be issued as part of the deployment process. A call to *deleteQuery* is executed when the process gets un-deployed for the same messages.

SUBSCR. ON PROCESS INSTANTIATION: When a process gets instantiated, *registerQuery* must be executed for each *Message* that is set to *subscribe on pr. instantiation*. *deleteQuery* can be called when the last reachable event element for a *Message* has finished executing or no connected event can be reached anymore. The deletion happens at the latest when the process instance terminates.

SUBSCR. THROUGH EXPLICIT SUBSCRIPTION TASK: If the control flow reaches a subscription task, the process engine executes *registerQuery* for the referenced *Message*. The execution of *deleteQuery* follows the same rules as in the preceding case.

SUBSCR. WHEN THE EVENT ELEMENT IS REACHED: Once the event element is reached, *registerQuery* must be executed for any *Message* that is not covered by one of the prior cases. *deleteQuery* must be called when the event element is finished.

be more precise about the time the calls should be executed (if possible). "reached"? "completed"? use the right bpmn words

Do I want to write about extended validity checks? soundness? The question would be: What happens if the model erroneous w.r.t. to the bpmn extension?

- handling subscription dependencies: - when are process variables replaced by their actual values - The use of dynamic process variable values introduces an additional complexity: Depending on the time of event subscription, the value of the process variable might not yet be available. - reference BPMN data elements: process INSTANCE variable -> the variable value might only be available during instance execution -> can we find an exact definition of this in the spec? - see BPMN2 spec pp.211+ : Process and Activity can have DataInput and DataOutput. DataInput can have an 'optional' attribute - during execution the variable data might or might not be available. Related Work: Francesca? -> too complex, we need a simplification for this. - what happens if the data is not available?

Time of un-subscription also must be clarified in bpmnx

7 REFERENCE IMPLEMENTATION

7.1 EXTENDING THE EVENT PROCESSING PLATFORM UNICORN

7.2 EVENT SUBSCRIPTION HANDLING IN CAMUNDA

8 RELATED WORK

9 CONCLUSIONS

complete xml example of a BPMN model using the extensions

DECLARATION

Put your declaration here.

Potsdam, August 2017

Dennis Wolf

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/