DENNIS WOLF

INVESTIGATING EVENT SUBSCRIPTION MECHANISMS IN BPMN

INVESTIGATING EVENT SUBSCRIPTION MECHANISMS IN BPMN

DENNIS WOLF

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

< Any Subtitle? >

August 2017 – version 1

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 distributed 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...

v

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

API   Application Programming Interface

BPM   Business Process Management

BPMN   Business Process Model and Notation

CEP   Complex Event Processing

EdBPM Event-driven Business Process Management

EPL   Event Processing Language

JVM   Java Virtual Machine

HTTP   Hypertext Transfer Protocol

PEP   Process Engine Plugin

REST   Respresentational State Transfer

SQL   Structured Query Language

UML   Unified Modeling Language

XML   Extensible Markup Language

x

INTRODUCTION

Given the increasing competition on the global market place, companies are seeking to improve their products while reducing costs. In many areas, Business Process Management has been chosen as one of the tools to help stay competitive. Especially large enterprises, but also small to medium businesses formalize their workflows in business process models to allow archiving, documentation and automatized management and execution.

With Business Process Technology in constant progression, the opportunities that the field has to offer are ever growing. Since the recent years, many efforts have been dedicated towards bringing together business processes and Complex Event Processing. By the help of Event Processing Systems, companies are trying to get a hold of the exponentially growing amounts of data that occur in today's IT environment. Incorporated in process executions, events heavily increase their capabilities and flexibility. They can be utilized for intraorganizational communication between processes or business departments, but also allow to respond to external situations within seconds or milliseconds. The Business Process Model and Notation (BPMN), an industry standard for representing business processes both visually and textually, natively supports the use of events in a plethora of ways. Events are considered a main building block of a feature-rich process modeling language and can, for instance, be used for instantiating processes, communicating between process participants or to support decisions.

An interaction with a Complex Event Processing platform generally follows the publish-subscribe paradigm: The event consumer contacts the platform and issues a subscription to a specific subset of available events. An event producer, for example a Vehicle providing its current GPS location, publishes information to the event processing platform, which is then forwarded to every consumer that had subscribed. Intermediate Events are a basic way to implement event-based communication in BPMN and facilitate the reception of external message events. Nevertheless, the BPMN specification does not specifically consider the publish-subscribe workflow and provides limited capabilities when it comes to incorporating event subscription and un-subscription operations in business process models. This work investigates the consequences of this lack of specification and provides a design and implementation to overcome the identified shortcomings.

1.1   MOTIVATION

why? - cant live without complex event processing in business processes, increasing demand for using events in processes - pub/sub is a fundamental part of using events in BPs - still it is not considered in bpmn - there is research on including the subscription query in the model, but the subscription time is not further defined - the bpmn specification says "..", that leaves us with a very limited listening time

- but distributed setup -> hard to control - we need subscription before occurrence time - issues will occur when mis-used, delay or blockage, significant time and financial loss -> brief, textual example - to ensure the efficient use of events in processes, a more flexible use of subscription is necessary

> the problem will be further illustrated in motivating examples

1.2   CONTRIBUTION

Working towards a more flexible use of event subscription in business processes...

(0) reviewing the problem from different perspectives, deriving requirements (1) assessment of the capabilities of standard bpmn (2) Proposition of a BPMN extension for flexible event subscription, its advantages (3) derived requirements to process engines and CEP platforms (4) A reference implementation using Camunda and UNICORN

1.3   STRUCTURE

write

BACKGROUND ON EVENT-DRIVEN BUSINESS PROCESS MANAGEMENT

2.1 BUSINESS PROCESS MANAGEMENT

With its origins dating back to the process orientation trend of the 1990s, Business Process Management (BPM) has meanwhile become a mainstream tool to support organizations. It had been noted, that company workflows can essentially be broken down into activities that are executed in a coordinated manner by one or more parties. A certain group of activities thereby form a process which is executed within an organization. More precisely, Weske [30] defines a single *business process* as follows:

DEFINITION 1 (BUSINESS PROCESS): A *business process* consists of a set of activities that are performed in coordination to realize a business goal. Each business process is enacted by a single organization, but it may interact with processes performed by other organizations.

The term *Business process management* describes the techniques available to develop and support processes throughout their life-cycle. It is grounded in the use of explicit process representations which ultimately allow the exchange, analysis and reproduction of the workflows. This process specification is referred to as the *business process model*, composed mainly of activities and the rules that are necessary to coordinate their execution. When a process is performed according to its model, the single execution is called *process instance*. Based on a process model, the number of possible instances is theoretically unbounded.

DEFINITION 2 (BUSINESS PROCESS MODEL): A *business process model* consists of a set of activity models and execution constraints between them. A *business process instance* represents a concrete case in the operational business of a company, consisting of activity instances. [30, p. 7]

The lifecycle of a business process can be described in four cyclic phases in that numerous stakeholders interact and contribute depending on their specialization. Process development starts with a *Design & Analysis* phase which yields a refined and validated business process model. In the following *Configuration* phase, it is necessary to prepare the process implementation, select the means and an environment to run the process in. The action of making the process runnable

in the execution environment is called *process deployment*. After these preparations the process can be enacted in daily business while its current state is monitored and system maintenance is performed if necessary (*Enactment* phase). A single process execution begins with the *process instantiation*, when the process has succeeded or is aborted, we say the process is *terminated*. During the enactment, system and stakeholders can start collecting performance indicators and process execution logs to allow evaluating the quality of the process specification. If that *Evaluation* step reveals deficiencies, the lifecycle starts over by re-entering the design phase. The *process un-deployment* is performed if necessary, so that no new instances of the process can be started. [30, p. 11 ff.]

add reference to [8], mention that Weske says pretty much the same

2.1.1 *Business Process Meta Model and Activity Lifecycle*

Attempting to...

DEFINITION 3 (BUSINESS PROCESS META MODEL): Let C be a set of control flow constructs. $P = (N, E, type)$ is a *process model* if it consists of a set N of nodes, and a set E of edges. [30], p. 91

- $N = N_A \cup N_E \cup N_G$, where N_A is a set of activity models, N_E is a set of event models and N_G is a set of gateway models. These sets are mutually disjoint.

- E is a set of directed edges between nodes, such that $E \subseteq N \times N$, representing control flow.

- $type : N_G \rightarrow C$ assigns to each gateway model a control flow construct.

Much like the process instance lifecycle, each activity follows a cycle - introduce activity lifecycle/control flow

Fehlende Abbildung   activity lifecycle

While traditionally activities are executed manually by company staff following the written process specifications, computer systems are used today to drive the execution and enforcement of business processes and organizational rules. The generic software systems utilized for that purpose are introduced in the following section.
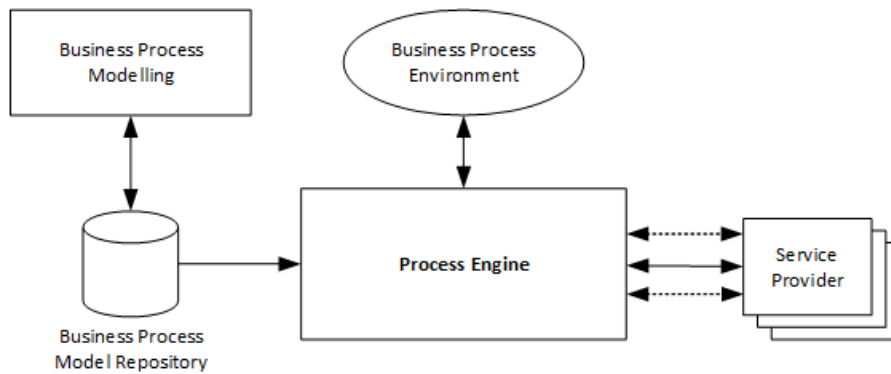
Figure 1: Business process management systems architecture model (see [30], p. 120)

2.1.2 *Business Process Management Systems*

The implementation of business processes has developed from a manual execution guided by business rules to a fully automatized execution in a specialized IT environment. One of the main reasons for BPM's growing popularity is that in today's fast-paced economy, a large part of the business activity is either supported by computers or even carried out autonomously by them. The specialized software systems that are utilized to support the enactment of business processes are referred to as *business process management systems*.

PROCESS MANAGEMENT ARCHITECTURE   A typical IT infrastructure for driving business processes is illustrated in Figure 1. Five principal building blocks are considered which will be explained in the following.

With reference to the business process lifecycle, the visualized scenario commences with the *Business Process Modeling*. As a result of the *Design & Analysis* phase, new process models are created and refined to be stored in the *Business Process Model Repository*. The relation between the two elements includes writing new models to the repository as well as reading models for review and further modification. Given that the desired model is approved for enactment, the process gets deployed to the *Process Engine* as part of the configuration step. The process engine is the heart of the execution environment. It performs the execution of the processes from deployment until un-deployment, while the enactment and instantiation is controlled by the *Business Process Environment*. An indefinite number of *Service Provider*s realize application services to support the process execution. A service provider can be a software module but also a knowledge worker performing a particular process step.

THE CAMUNDA BUSINESS PROCESS ENGINE   A large and ever growing number of process engines is available on the market, in-

cluding solutions from IT giants like SAP, IBM and Oracle. In this work, *Camunda BPM* [12] has been chosen to illustrate implementations. As of August 2017, the software product is available in version 7.7.0 and comes in a commercial, regularly updated version and in a free, community-driven solution that is updated with every major release. Camunda is popular among the research community as the source code is openly available, the product is mature, but actively developed and offers comprehensive support for BPMN 2.0. It is designed to be extensible and easily modifiable to adapt to custom requirements. *Camunda BPM* comprises a modeling tool, the Camunda process engine core and a number of browser-based user-interfaces to control process enactment and monitor execution state. Chapter 7 will provide further details about the engine architecture and extension mechanisms.

2.1.3 *Business Process Model and Notation*

Given the general semantics of business processes, a specific modeling notation has to be selected to express an informal process description in a formal, interchangeable way. Different languages and notations have become available over the years, each serving different specializations. Kossak et al. [19] organize some of the more popular languages as follows: A subset of them are focused on the control flow of business processes, for instance Business Process Model and Notation (BPMN) [26], Yet Another Workflow Language and Petri Nets; some focus on object-orientation, like Unified Modeling Language (UML) activity diagrams and use case diagrams; some are dataflow oriented, e. g. the Structured Analysis and Design Technique.

references to the other languages

Among these, the BPMN has developed into a widely-adopted industry standard, also becoming ISO-standard in 2013 [15]. The standard is developed by the Object Management Group [25] and now available in version 2.0 (January 2011) after being first released in January 2008. BPMN can be understood as an extension to the abstract business process meta model (Section 2.1.1) adding a comprehensive catalog of visual representations and semantic constructs on top of a meta model. Furthermore, one of the most important features of its latest version is the a standardized interchange format provided through an XML specification, as [30] points out. As emphasized by Muehlen, Recker, and Indulska [24], the increased expressiveness of modern languages like BPMN comes at the cost of an increased complexity. An aspect that, apparently, did not stop it from gaining popularity.
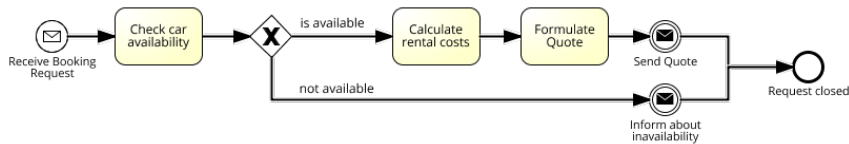
Figure 2: Simple BPMN model of issuing a quote for car rental

ESSENTIAL ELEMENTS OF A BPMN MODEL   Following the abstract business process meta model, the core elements in any BPMN model are flow elements (nodes) and connecting objects (edges). Flow elements can be either *Events*, *Activities* or *Gateways*, each of them coming in different versions. This section will introduce a subset of the elements available through the BPMN specification to build the foundation to comprehend the thoughts presented in this work.

Figure 2 shows how a booking request might be handled in a car rental business. Circular elements represent events, diamond-shaped elements are gateways. Activities are visualized by rectangles with rounded corners. The given process gets instantiated whenever a booking request request is received from a customer, shown as a Message Start event. As a first step, the employee assigned to handle the request must check if the desired car is available. To that follows an exclusive OR-Gateway, distinguishing the further process flow depending on the availability of the car. If the car is available, the quote must be created in two sequential activities to be then sent out by the *Send Message Event*. If the car is not available, the customer is informed about the closing of his request. In either case, the process ends with an *End Event* after the customer was informed about the result of his request.

- introduce the other essential elements, types of events

PROCESS CHOREOGRAPHIES   - talk about how several parties can interact => choreography - Message - choreographies

Fehlende
Abbildung   Sample BPMN choreoraphy model
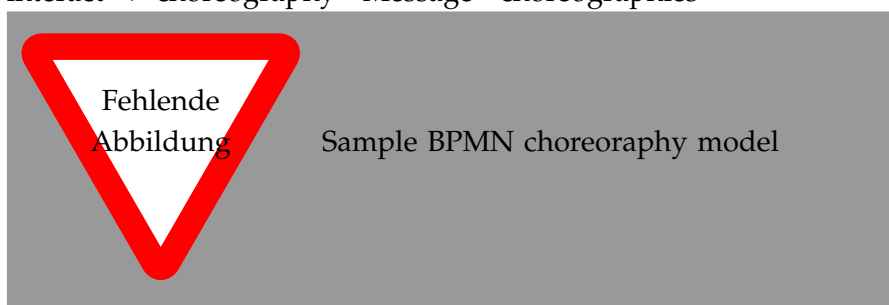
2.2 COMPLEX EVENT PROCESSING

The IT world is facing an exponential increase in the amount of produced data. A significant part of this data are pieces of information about real-life occurrences, such as a current sensor value, an interaction on a website or the location of a vehicle on the road. We call this kind of strongly time-related information an *Event* and the according computer science field Complex Event Processing (CEP) [10]. - events are of a certain type

More precisely, Events are defined as follows...

from http://www.softnet.tuc.gr/ minos/Papers/rtstreams15.pdf - Event tuples are defined by [11] as tuples e =< s, t >, where e represents the event of interest, s refers to a list of attributes and t is a list of timestamps, the first being the start of the event and the last the end of it

Four major components take part in an event processing network: (a) An *event producer* provides information to the system, (b) an *event agent* processes the occurring information, so that it can be delivered to the *event consumer* (c). The components are linked through *event channels* (d). Typically, a so-called *CEP Engine* (also: CEP platform) is at the heart of the system, taking the role of an event agent. Modern CEP platforms are trimmed to maximum efficiency, being able to process hundreds of thousands of events every minute. Their main purpose is to accept incoming events from event producers, filter and match them according to selection criteria and, finally, derive a new event occurrence to be sent to the registered event consumers.

THE PUBLISH/SUBSCRIBE PRINCIPLE   In event-based architectures, communication takes place according to the *Publish/Subscribe Principle*. The concept essentially demands that an event processing middleware publishes events to processes only after they have issued a subscription for these events. Consequently, there is a strict temporal order between the actions subscribe, consume and un-subscribe. The consumption and un-subscription can only happen after the subscription. Once an un-subscription has been issued, no consumption can follow. [29]

One of the main advantages of this principle is, that the involved parties are *referentially decoupled*. They do not need to explicitly refer to each other, an aspect that is also acknowledged in [10]. Their decoupled nature facilitates the management and development of event processing networks. Event producers and consumers might change frequently. Whenever a new event source is available it can be connected to the CEP platform without considering all future consumers. Consumers can subscribe and un-subscribe without influencing the operations on the consumer side.

Fehlende Abbildung   pub/sub
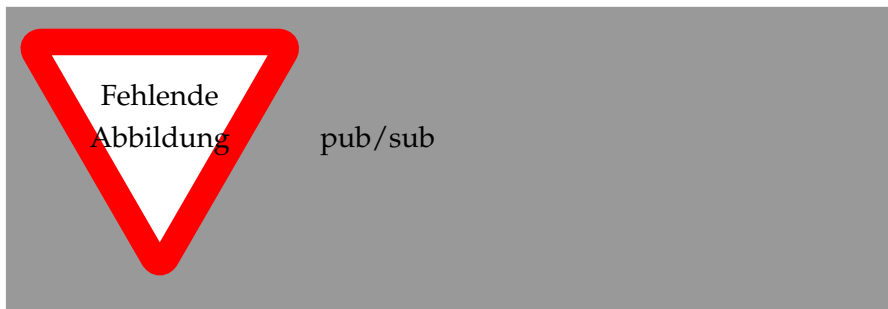
STREAM PROCESSING   To be able to cope with potentially large amounts of data, Complex Event Processing Platforms work on the basis of *stream processing*.

'relational' db

In a traditional database, information is stored for an indefinite amount of time. When a user queries the data store, the system processes the tabular data and calculates the requested result. The advantage of this approach is that the user can access historic data at any time, as long as it is not explicitly deleted from the database. In many occasions, the amount of available data significantly surpasses the storage capacities and that concept can no longer be followed.

Stream processing addresses the mentioned challenge by largely reducing the amount of data that is persisted in the system. Instead, it is the goal to keep only those pieces of information, that are necessary to process a result for currently registered queries. Incoming data objects, or events in the case of a CEP platform, enter the event stream and are immediately evaluated against all existing query expressions. If the information is not required to process any of the queries, it is deleted instantly. As a consequence, a certain event can not be part of a query result, if that query has been registered after the occurrence of the event. In case aggregated information is demanded by the query, the stream processor will internally store aggregated information, but not keep every information that led to the aggregated value. [18]

Fehlende Abbildung   stream processing
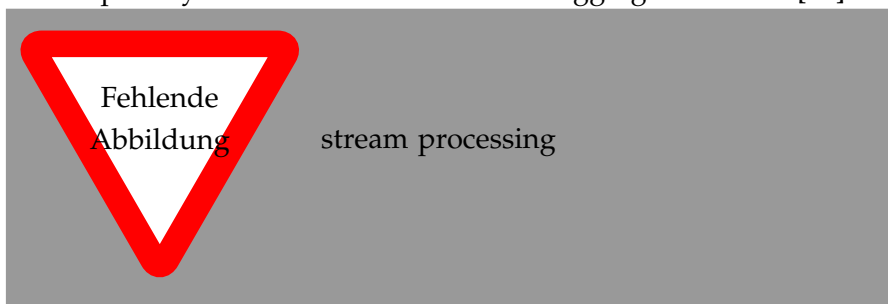
EVENT QUERY LANGUAGES   The subscription to an event in a CEP platform is primarily defined by an *Event Query*. Many modern event query languages are inspired by SQL, but cannot be entirely compliant due to the different underlying data processing concept. When

formulating event queries, it is essential to consider the stream processing principle.

For the illustration of the concepts, this thesis relies on the Esper Event Processing Language (EPL) [14], utilized in Esper event processing engines like the one employed in Chapter 7.

> a little more on epl, what can it do? windows, joins, filters

> event patterns

Listing 1: Sample Query in Esper EPL

```
SELECT time, delay, delayreason
FROM eurotunnel
WHERE delay > 30
```

2.3   EVENT-DRIVEN BUSINESS PROCESS MANAGEMENT

- see opheretzion p.17 and where are we now - maybe also reference baumgrass paper / GET project | Towards a Methodology for the Engineering of Event-Driven Process Applications (BG2016) - +

- there are 2 use cases for events in bp: (1) for process monitoring (=> pe is event producer) > what's it about this thesis focusses on (2) as it investigates subscription mechanisms during the execution of a process

(2) for driving processes (=> pe is event consumer) > Event-Driven Process Control (event-driven business process management: where are we now)

- interplay BPT to CEP platforms, a subscription must be issued - no standard yet available to do subscription in bpmn - it must be assumed that the subscription is either already active or explicitly modeled in BPMN, e.g. using a service task - OR given the BPMN spec it is generally assumed that the subscription is executed as soon as en event is enabled - further analysis of this topic is provided in ...

- Correlating events to process instances

- exercise through one example

PROBLEM STATEMENT

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

illustrate the complexity through 2 sample scenarios that are similar but different

3.1 MOTIVATING EXAMPLES

To allow a better understanding of the issue, event-driven use-cases from two different domains are presented in the following. The cases are revealed through their standard BPMN representation, which can cause problems in certain real-life situations. It is illustrated, why the time of event subscription is of great importance and motivate to study the mechanics and implications of event subscription in business processes.

DELAY OF A LOGISTICS PROCESS   The first example(Figure 3) is taken from the logistics domain and shows a truck transport that has to cross the English Channel. The truck driver receives the transport plan for his next tour from France to the UK. By default, the company crosses the Channel using the Eurotunnel, an underground train connection between London and Paris.

After loading the goods at the factory, the truck will head towards the check-in location of the Eurotunnel. If everything runs on schedule, the truck will cross the channel on the train and then deliver the goods in Great Britain. Alternatively, the process considers a route using the ferry from Calais(FR) to Dover(UK). The Eurotunnel administration publishes delay information approximately every 30 minutes through an RSS feed on their website. While it mostly operates on schedule, delays ranging from 15 minutes to several hours occur regularly. It can happen that new information is not published for multi-
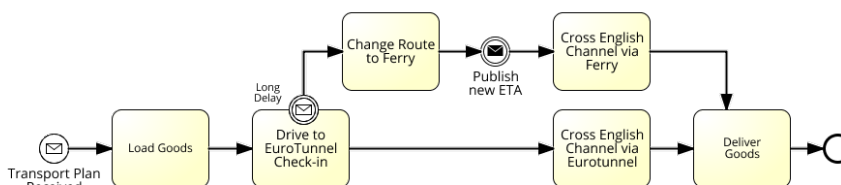

Figure 3: BPMN Model of a Logistics Process using events for route optimization (Example 1.1)
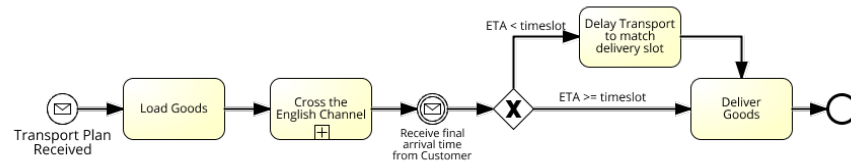
Figure 4: Transport via English Channel that is timed to a delivery slot (Example 1.2)

ple hours. Significant delay events (delay > 30 minutes) are received through a boundary catching message event attached to the activity *Drive to Eurotunnel Check-In*. The boundary event is interrupting, hence the activity is canceled if a delay occurs. The transport continues towards the ferry terminal and crosses the English Channel over sea. After crossing the channel, the goods are delivered to the recipient.

> show a cep query for that scenario to make it more precise

According to the BPMN specification, the listening for the boundary event starts as soon as the related activity is activated.

> is that exactly correct?

Given that events arrive every 30 minutes, there can be a gap of up to half an hour, before the first information becomes available. In the worst cases, when data isn't published for several hours, this gap will be even bigger. Let's consider a very busy weekday. A technical fault occurred in the tunnel earlier and the train runs 3 hours behind schedule. The last information on the RSS feed was published at 2:35pm. At that time the truck driver is still in the process of loading goods, finishing the activity at 2:40pm. Following the process definition, the driver now departs towards the Eurotunnel check-in. The system publishes updated information at 3:15pm: operations are still 2:30h behind schedule. The message gets received through the process and the truck driver takes the alternative route to the ferry, but only after heading to the Eurotunnel for 35 minutes. The late change of plans causes an unnecessary delay to the shipment.

Figure 4 is an extension of the the transport process. In logistics, it is common that a delivery cannot be accepted at arbitrary time. Instead, the receiving party assigns delivery windows to the transport company. The transport must arrive during the given time window, otherwise the delivery cannot be completed. After crossing the English Channel, the process model shows the catching of a message event containing the desired final arrival time at the factory. There is an agreement with the factory, that the delivery slots will be approved 2 hours before the expected arrival. If the current ETA of the transport is greater or equal to the arrival time, the driver will head to
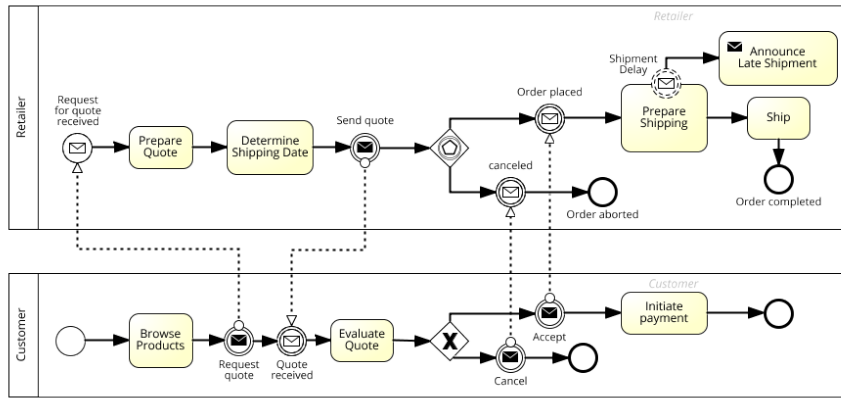
Figure 5: Model of a retail order management process (Example 2)

the drop-off point straight away. If the transport is ahead of schedule, the driver will have to delay the delivery to match the time window.

The presented process model illustrates another complexity of using events in processes. Again, the listening to the announcement of a delivery window will start when the event element is reached, in this case only after crossing the English Channel. Until an event has been received, the process will not continue. Much worse: if the receiving party sends out the arrival time information too early, while the truck is crossing the channel, the event is missed. If it is not issued again, the process cannot receive a message and will get stuck indefinitely waiting to catch the event.

Neither of the two presented catch events allow for an efficient and reliable execution of the process. They can cause unnecessary delays and even blocking of the process execution. The remainder of this work will further analyze the capabilities of BPMN to express event-usage scenarios and propose solutions to the mentioned problems.

> Consider more possible event occurrence times to prepare for the next chapter

UP-TO-DATE SHIPPING INFORMATION FOR AN ORDER   A similar situation can be observed in the Order Management process presented in Figure 5. It describes the interaction between customer and seller in a traditional distance retail scenario: After browsing the product catalog, the customer requests a quote for the articles he or she is willing to buy. The retailer makes an offer including an approximation of the expected shipment date and sends it to the customer. That quote is then either accepted or not and the payment is issued if necessary. Once the retailer is informed about the placement of the order, the products are packed and shipped as soon as possible. For articles that are not currently in stock, the retailer must await the shipment from the factory. If any of the factory-shipments is delayed, the re-

tailer cannot ship in time and will announce a delayed shipment date to the customer. This situation is modeled through a non-interrupting boundary event attached to the *Prepare Shipping* activity, which triggers the sending of the updated shipment date to the customer.

The process shows a number of similarities, but also differences in terms of event-use when compared to the previous process example. At first we want to look at the three intermediate catch events, *Quote received*, *Order canceled* and *Order placed*. In each of the cases, the event to be caught is the direct response to a message that was sent right before. While the process will also enter a waiting state until the response arrives, that waiting is not to be interpreted as an unnecessary delay to the process execution. Other than in Example 1.2(Figure 4), there is nothing useful to do before the response is received. It is furthermore worth noting that the response messages cannot be missed, because the message catch event immediately follows the message send event.

A different situation holds for the boundary event *Shipment Delay*. While the subscription to a Eurotunnel event can be issued at any time, it does not depend on any process data, the shipment delay has to be observed for each product that is part of the order. A subscription can therefor not be executed before the activity *Prepare Quote* has finished executing

> show a cep query for that scenario to make it more precise

By process definition, the system will listen to shipment delays once the activity *Prepare Shipping* has started executing, much later than possible. Any events that occur between the two activities, that means, for example, when the customer makes the decision about accepting or canceling the order, cannot be considered in the process execution and the customer will not be informed about a possible late shipment.

> write about service interaction patterns somewhere

The two presented examples have illustrated the complexity of using events in business process, especially when the possible event occurrence times are considered in detail. Differences have been pointed out as to how exactly the event is placed in the process, if it waits for a direct response to an earlier request or if the event occurrence unrelated to the execution of that very process. Motivated by this complexity, it is the goal of this work to identify the capabilities of BPMN to handle event subscription in business processes.

> this chapter must be closer related to event subscription. Don't make it too general. subscription is the thing
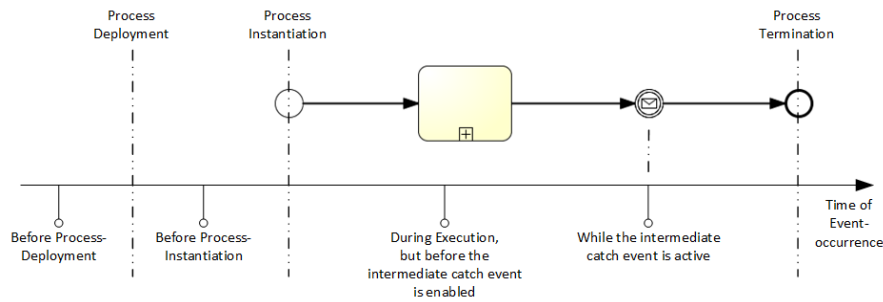
Figure 6: Possible event occurrence times in relation to a process execution
          life cycle

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.

EVENT SUBSCRIPTION TIME   The most important variable to con-
sider is the time of event occurrence.

> reformulate

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 out-
side 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 6 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.

> How about system-deployment/process engine start and process
> undeployment? show in illustration, but say in text that we sim-
> plify this for now. after undeployment is essentially before de-
> ployment 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 instan-
tiation* 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.

> Explain that, to catch events at these times, the subscription has to happen before. Therefor we derive event subscription times

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 significant 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

> formulate a concise Problem Statement

3.3 REQUIREMENTS DEFINITION

> Maybe move to the end of 4. and add non-functional requirements as well?

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 (Chapter 4) and to develop a new concept to handling event subscription in business processes (Chapter 5).

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.

> target only intermediate events. (limitation, although it also makes the most sense.)

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?

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 7](#) 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.
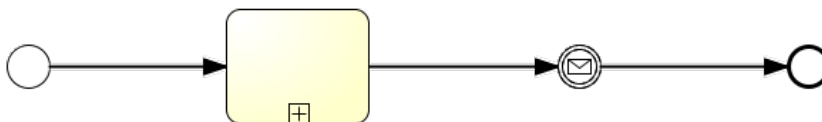
Figure 7: Abstract Process using an Intermediate Catch Event

19
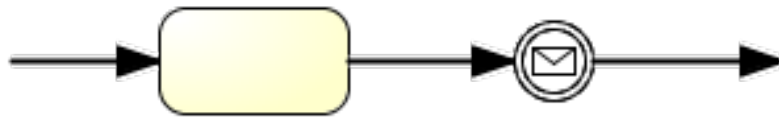
Figure 8: Standard Intermediate Catch Event
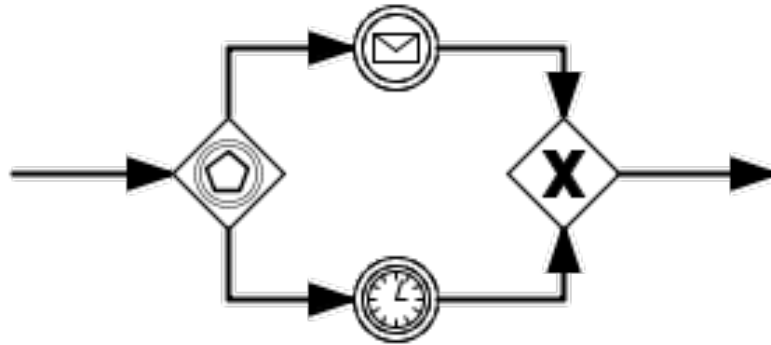
Figure 9: Intermediate Event with a parallel Timer Event

SCENARIO 01: 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 02: THE EVENT DOES NOT OCCUR   [alternatively: a receive task with bounda

In certain situations an event might not occur at all. Given a basic event implementation like in Figure 7, 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 9 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.
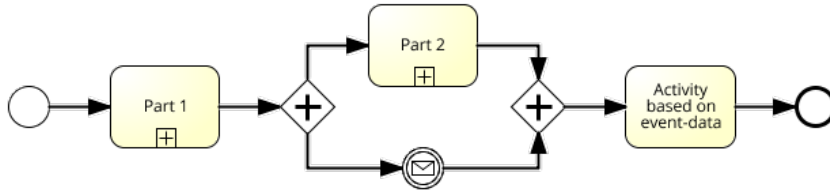
Figure 10: Event Element in parallel process flow

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

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

SCENARIO 03: 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 10. 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 10, the event may occur at any time during the execution of the collapsed sub-process *Part 2*.

SCENARIOS 04 AND 05: 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 (O_4) and an occurrence time before the deployment of the process in the Process Engine (O_5).

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 11 reveals the interaction of the *Orig-*
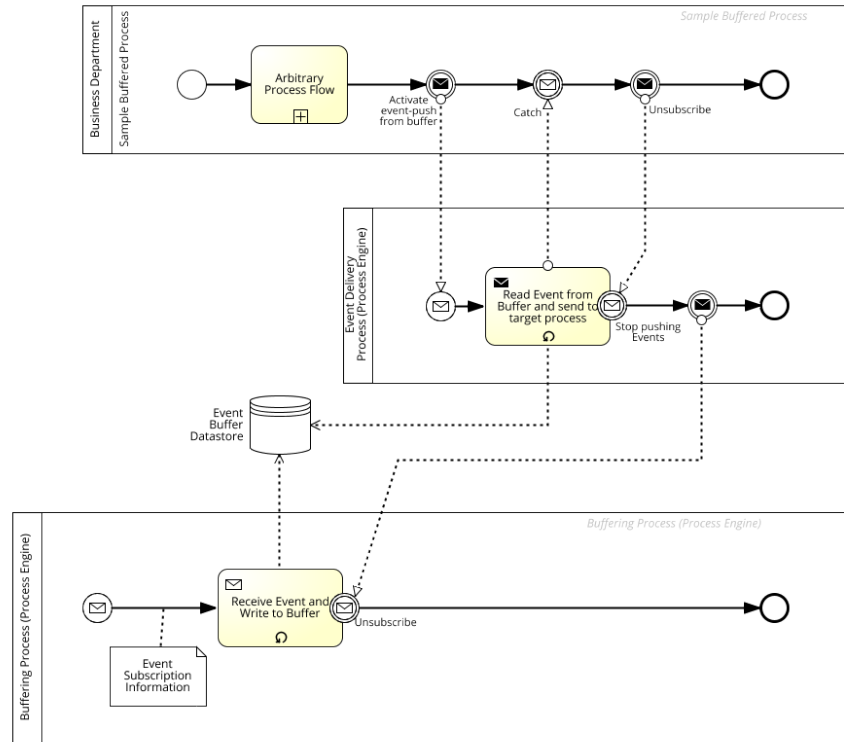
Figure 11: Event Buffering through an auxiliary Buffering Process

inal Process, the two auxiliary processes and the data-store. To start listening for an event, the *Auxiliary Buffering Process* has to be instantiated through 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 11, 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 12 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 13) 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
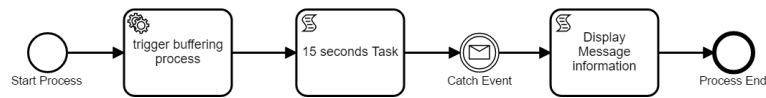
Figure 12: Generic Example Process in Camunda for Occurrence Scenario *O3*
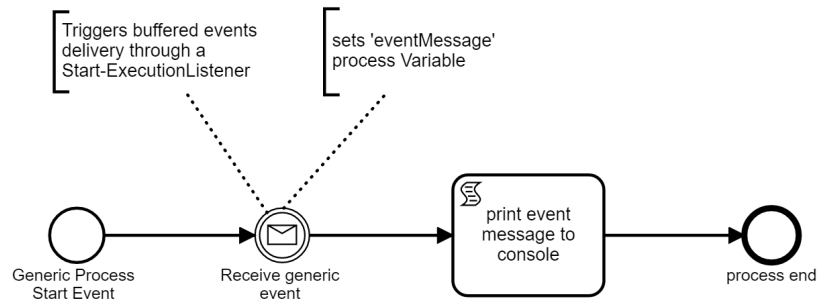
Figure 13: Generic Example Process in Camunda for Occurrence Scenarios *O4* and *O5*

been chosen for buffering the event data because it's freely available, quick to set up, offers standardized access via SQL queries and Java 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 14 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 *Unsub-*
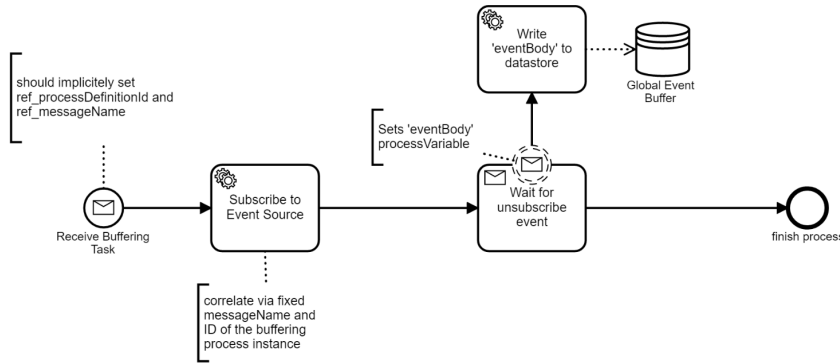
Figure 14: Auxiliary Buffering Process in the Camunda Modeler

scribe 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 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 15) 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 instan-
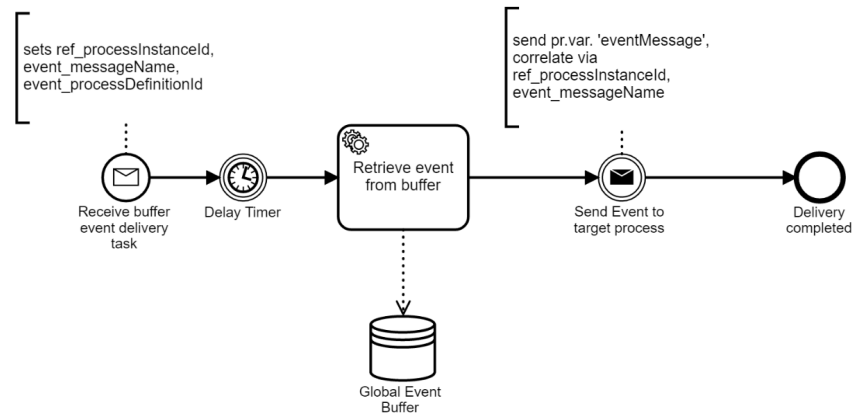
Figure 15: Auxiliary Event Delivery Process in Camunda Modeler

tiation 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 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? Or make one of them Requirement 5

MISSING 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.

ADDITIONAL MODEL COMPLEXITY

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?

ADDED 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? or is there an additional event engineer?

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.

The extensibility mechanism allows to extend standard BPMN elements with additional attributes, while maintaining a valid BPMN core. Extensions are specified through an external definitions file and can be included into a BPMN process model by reference. To allow the flexible use of event subscription in BPMN models, a number of additional attributes must be added to BPMN process models.
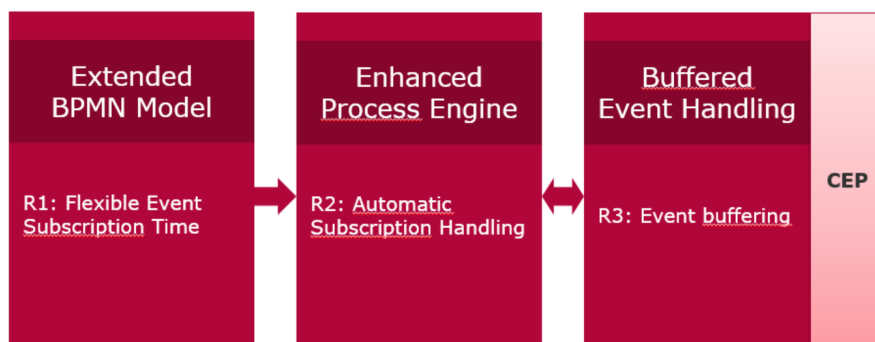

Figure 16: The concept for flexible event subscription involves three modules: A BPMN extension, enhanced process engine behavior and buffered event handling.

The presented extension is designed for use with intermediate catch events, boundary events and receive tasks. Start events and End events are not considered because flexible subscription time is not relevant in these cases. Each of the three elements references a BPMN *Message*, the common denominator for communication within and across business processes. Semantically, the *Message* type is most suited to be extended with subscription information.

mention that others have extended the events instead, but this proposal goes another way

Fehlende Abbildung

A UML diagram of Message, Message-Flow, MessageEventDefinition, Catch Event, Boundary Event, ReceiveTask

explain that the Message is a common element in BPMN models. It is the main generic type used for communication, collaboration. The proposed extension to the Message type is only to be used by Message Receive situations in Intermediate Catch Event, the Boundary Catch Event and the Receive Task.

By specification, *tMessage* comprises an attribute *name*, the name of the message, and *itemRef*, the reference to a BPMN *ItemDefinition*. Additionally, it inherits all attributes from the BPMN *RootElement*. In the following, the required additional attributes will be explained one after the other. A complete list is available in Table 1. The goal is to retain a stand-alone model that contains all information necessary to execute the subscription to the event source.

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.

ref background

It is assumed that only one CEP platform is in use, whose access information is configured centrally for the current process execution environment. Consequently, there is no need to specify these two parameters on message level. 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.

ATTRIBUTE NAME	VALUE OPTIONS (<u>DEFAULT</u>)	OPTIONAL
eventQuery	any string	n
subscriptionTime	process-deployment, process-instantiation, <u>event-reached</u>	y
bufferPolicies	Complex Type (see below)	y
BUFFERPOLICIES		
LifetimePolicy	string in ISO time-span format OR '<u>infinite</u>'	y
ConsumptionPolicy	<u>Reuse</u>, Bounded-Reuse(n), Consume	y
SizePolicy	int (< 1 for infinite), <u>1</u>	y
OrderPolicy	<u>FIFO</u>, LIFO	y

Table 1: Available attributes in the BPMN extension for flexible event subscription

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; explain what they do (different)

Given this first 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 an event-driven process. Two different tools are to be offered to support all subscription times demanded by *R1.1*: Firstly, the subscription can happen in the background . 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.

INTRODUCING EVENT BUFFERS   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. Section 5.1.4 introduces a set of advanced buffer policies to adapt this behavior further.

> 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.

INTERPLAY OF EVENT QUERIES AND BUFFERS   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

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 can take one of the following three values: *Process Deployment*, *Process Instantiation*, *Event reached*. The last option is the default option, coming closest to the BPMN specification. 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.

> 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* of *tMessage*, an extension to the BPMN ServiceTask is proposed.

> it follows the ideas presented in ... but through referencing the message element

> latest concept (xsd) states that it's an extension of tTask

Fehlende Abbildung

Example 2 using an explicit subscription task

The extended task is used to execute the subscription explicitly as part of the process flow. 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, because the necessary information is not yet available. Instead, an early subscription can be implemented using the extended service 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.

> 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

> only if subscriptionTime not 'on depl'

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

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, Weidlich, and Weske [23], a number of advanced buffer attributes are available through an extension attribute *buffer-Policies*.

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

LIFE-TIME OF BUFFERED EVENTS
    The *LifetimePolicy* allows to specify after which timespan elements in the buffer should be deleted. Timespans shall be defined using ISO timespan format.

> ref external

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.

> ref example

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

> for the reader it would not be clear, what a 'buffer instance' is

    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
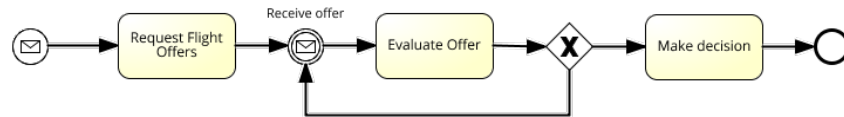
Figure 17: Flight Booking process using a consuming buffer
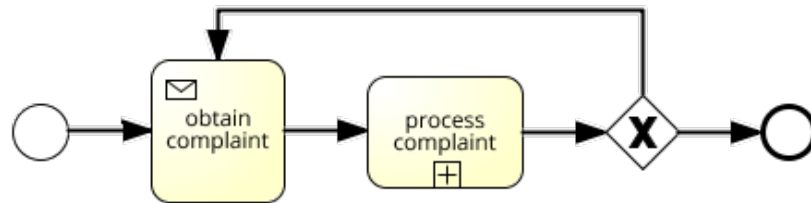
Figure 18: Shared consuming buffer in Complaints Handling

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

BUFFER SIZE AND ORDER POLICY

  - 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.

Talk about 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 - 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 com-
plexity - 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 dupli-
cated 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: ws02, esper and ... - ws02: ? - 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 functional-

39

ity 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.

> note that splitting this into four actions is not a new concept. actually thats how event processing platforms normally operate, only without creation of the buffer and without delivering the buffered events

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

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. *subscribe(queryId, notificationPath): subscriptionId*
   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

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(subscriptionId)*
   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 must be available 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, a subscription can be issued individually using *subscribe* 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 *removeQuery*. 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?
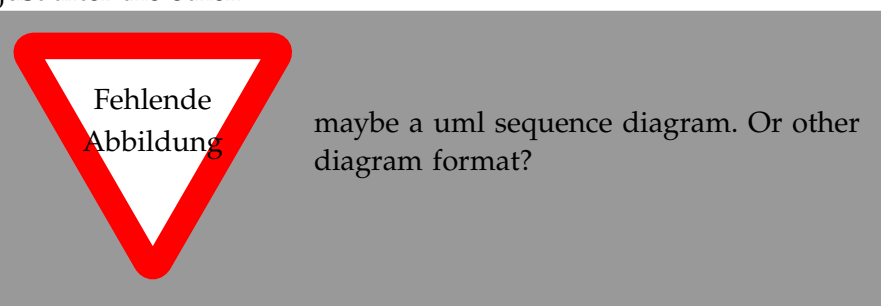
maybe provide a swagger definition for this?

ARCHITECTURAL OPTIONS TO IMPLEMENT THE EXTENDED API
That extension can either be implemented by adopting the CEP plat-
form 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 middle-
ware as the only choice.

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

Fehlende
Abbildung   three options to implement the event
                        buffering and api extension
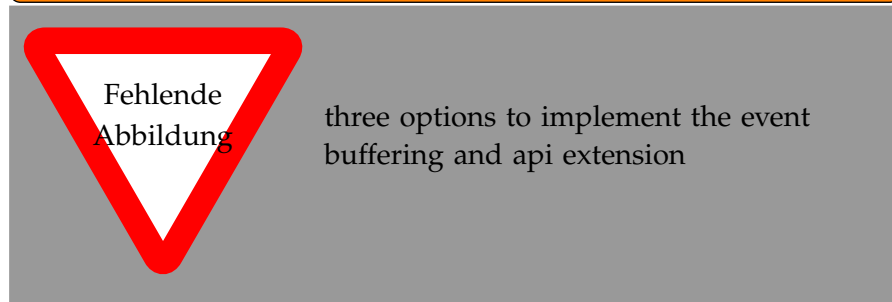
A reference implementation for an extended complex event pro-
cessing platform is presented in Section 7.1 at the example of the
Esper-based CEP platform *Unicorn*. It also explains, why extending
the event platform was the preferred choice in the given scenario.

the communication overhead must be considered: if many events
are sent to the buffer, but they are rarely issued to a process in-
stance, then it will make sense to place the buffer closer to the
event engine w.r.t. the the involved comm overhead. Though it
must be noted, that if events occur rarely, but many process in-
stances consume them, than there will be an additional overhead
to send events from the buffer to the instance. in that scenario, it
might be advantageous to implement the event buffering close to
the process engine

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

write

6.2   EXTENDED PROCESS ENGINE BEHAVIOR

It is the task of the Business Process Engine to interpret and exe-
cute 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

> did i cover all requirements?

REFERENCE IMPLEMENTATION

Chapter 5 has presented a concept that involves the model layer, the process engine and the event engine. It enables flexible event subscription to overcome the issues revealed in ... By explicitly chosing the time of event subscription and bridging the gap between event reception and consumption by the process instance it is possible to ...

> write from keypoints

the model extension is formalized in Section 5.1, hence BPMN models extended by all information necessary for flexible event subscription can be created.

to evaluate our results and provide a reference implementation, we enhance the business process engine Camunda and the event processing platform UNICORN following the procedures described in Section 6.2 and Section 6.1 respectively. Camunda is extended by providing a Process Engine Plugin, Unicorn by adapting the source code. The resulting illustration of an event-driven architecture is evaluated by implementing the examples shown in Section 3.1.

7.1 EXTENDING THE EVENT PROCESSING PLATFORM UNICORN

UNICORN is an event engine developed for academic purposes at the Business Process Technology chair of Hasso-Plattner-Institute, Potsdam. [13] It focuses on event processing for BPM, ... Based on Esper, that means, what is esper what unicorn adds to esper: graphical and rest user-interface to ..., event persistence store, bp execution and correlation, recently also an event generator/replayer ==> it translated user interactions into the according Esper method calls

- currently unicorn uses version 5.3.0 of esper, therefor all future considerations are made based on the documentation of that version

> write from keypoints

It has been decided to extend the event engine in this scenario, for several reasons. Firstly, the performance and operation of the process engine shall not be jeopardized. Event Processing features potentially require a lot of performance to handle a large number of requests in a short amount of time. By implementing the event Buffering module loosely coupled to the Process Engine, we ensure that its performance is not influenced. Implementing a separate middleware was avoided in the course of this work, to keep the architecture easy and concise to describe. Unicorn is an academic prototype in constant develop-
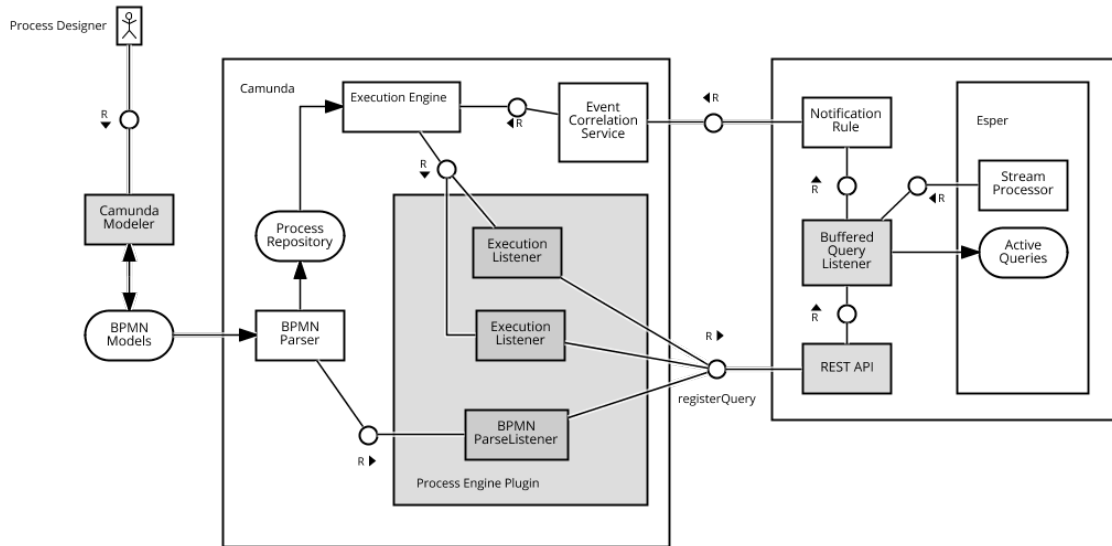
Figure 19: Architecture for flexible event subscription in Camunda and Unicorn

ment and therefor predestined to be directly extended for this kind of use-case. Moreover, the chosen architecture promises performance advantages thanks to running the event buffering in the same JVM as Esper.

7.1.1   *Event Buffering*

To allow the delayed delivery of events, buffering functionality is added to Unicorn. Whereas, originally, events that match a certain query in esper are directly forwarded to the notification-recipients, they will instead be held in a buffer until requested by recipient. If a recipient is already subscribed, than the behavior will be equivalent to the original scenario, because the event will be delivered instantly.

ENGINE-SPECIFIC IMPLEMENTATION OPTIONS   When investigating the options for implementing event buffers in UNICORN, solutions that are specific to the event processing platform were considered. Notably, the concept of a *Window* in event processing languages is very much comparable with a buffer as described in this work. Windows hold a variable number of events in memory according to specified window properties. That way, windows can act like size- or time-oriented buffers and allow access to older events whenever a new event occurs. *Named Windows* in Esper allow to create global data windows that can be modified and read from multiple statements.
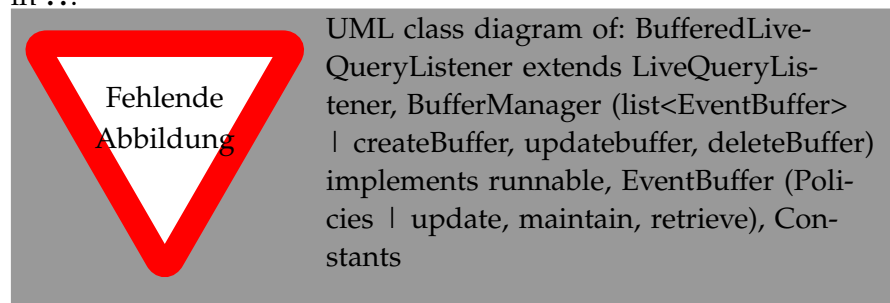
Similar functionality is provided by *Tables*, which follow a relational approach by primary key. [1]

If a query window is connected with the Esper-specific *Output Clause*, it is possible to delay query output with timing constraints or trigger it on the change of a global variable. [2] When putting together these features, an event buffer could be implemented as follows: The call *registerQuery* instantiates the window. If the *Consume*-policy is used, we use a named window that we share among queries and delete from after receiving the event. At *subscribe*, a notification-recipient is added to a query and output is triggered by causing a variable change in Esper. *Unsubscribe* and *removeQuery* remove a notification-recipient and de-register the query respectively.

A major drawback of this approach is, that the event queries have to be adopted to use the specific features which makes additional knowledge necessary when designing the processes. Alternatively, the original queries submitted by the process engine can be transformed before registering them in Unicorn, for example by encapsulating them in a sub-query. That way they make use of the mentioned features, but certain expressions will not be possible anymore due to limitations of the Esper EPL with regards to sub-queries.

Apart from this approach, it is worth noting that Unicorn offers a persistent storage of historic events, which can theoretically be used to implement the buffering functionality. However it would be necessary to abuse the relational approach to implement a persistent stream buffer on top of the SQL database, which is not desirable.

THE GENERIC BUFFERING SOLUTION The investigation of engine-specific solutions did not bring up an optimal solution to implement an event buffer with existing mechanisms. Furthermore, this reference implementation should not be entirely limited on one specific event engine. For that reason a new generic buffering module has been implemented within Unicorn. A UML class diagram is provided in **??**.

Fehlende Abbildung

UML class diagram of: BufferedLiveQueryListener extends LiveQueryListener, BufferManager (list<EventBuffer> | createBuffer, updatebuffer, deleteBuffer) implements runnable, EventBuffer (Policies | update, maintain, retrieve), Constants

---

1 see *Chapter 6. EPL Reference: Named Windows And Tables*, http://www.espertech.com/esper/release-5.3.0/esper-reference/html/nwtable.html

2 see *Chapter 5. EPL Reference: Clauses*, http://www.espertech.com/esper/release-5.3.0/esper-reference/html/epl_clauses.html

The module is built around the *EventBuffer*-class. Objects of the class are managed by the *BufferManager* and represent a single buffer entity, holding events of one query. Esper provides the query output as an object of type *EventBean*, which is stored in a list in *EventBuffer*. The buffer behaves according to the value of its policies, which influence the length of the list, the order in that items are retrieved from the buffer and the consumption behavior. The lifetime-policy, which requires that events are deleted from the buffer after a certain time, is ensured by a maintenance-thread that runs from the BufferManager class and iterates over all EventBuffer objects in a specified time interval. The default value is 5 seconds.

Unicorn uses the *LiveQueryListener* to react on new event-occurrences. It implements the Esper *UpdateListener*-interface and is registered in Esper as callback-function for a new query. Whenever an event matches that query, the object of *LiveQueryListener* is notified. Originally, the QueryListener notifies all notification-recipients that are known for that query and then drops the event. For the event buffering, a new class *BufferedLiveQueryListener* is created which extends the behavior of the standard query listener. Instead of only notifying all recipients, it also updates the associated buffer instance with the latest query output. The provided implementation serves for demonstration and reference purposes. It is not optimized for performance, for example in the case of overlapping data between buffers. Considering an event subscription issued on process instantiation, multiple *EventBuffer* instances will essentially store the same data, if multiple instances of the process are started at the same time or with minimal difference.

The presented buffering module supports the temporary storage of query results as demanded by **??**. After query creation, matching events can be stored in a buffer to be retrieved when necessary. All buffer policies described in Section 5.1.4 have been implemented. The following section explains how the REST API of Unicorn is extended to make use of the buffering module.

7.1.2 *REST API Extension*

Unicorn offers a webservice that allows users to interact with the event processing engine via the Hypertext Transfer Protocol (HTTP). The restful API comprises the most basic functionality, query registration, query deletion and obtaining query strings by the subscription identifier. An interaction generally works as follows: The user executes a POST to *<platform>/EventQuery/REST*, providing an event query and information about the desired notification-recipient (*notification path*). Unicorn registers the query in Esper and returns a unique identifier to the user. Whenever an event matches the query, the platform sends a notification to the specified recipient. a DELETE call to

<platform>/EventQuery/REST/{eventQueryUuid} triggers the removal of the query.

In accordance with the API requirements introduced in Section 6.1, additional functionality has been added to the Unicorn Webservice. The methods were added under a new path */BufferedEventQuery* to make sure that the existing features remain. They are as follows:

REGISTER QUERY POST to /BufferedEventQuery, returns queryId
Payload: JSON (eventQuery[, bufferPolicies]) with bufferPolicies:(lifetime, consumption, size, order)

Description: The provided eventQuery is registered in Esper using a BufferedLiveQueryListener. The BufferManager is used to instantiate a new EventBuffer object. The payload JSON object bufferPolicies is optional, and will be passed to the EventBuffer if available. Otherwise, the system falls back to the default values (Table 1). A unique identifier of that query and associated buffer is returned.

SUBSCRIBE POST to /BufferedEventQuery/{queryId}
returns subscriptionId
Payload: JSON (notificationPath) with notificationPath:(notificationAddress, processInstanceId, messageName)

Description: An new subscription is added to the selected query, so that a notification is issued based on the current buffer content and whenever an event matches the query. The notificationpath is specified including the id of the target process instance and the message name. This enables Camunda to automatically correlate the issued notification to the right process execution and message.

UNSUBSCRIBE DELETE to /BufferedEventQuery/{queryId}/{subscriptionId}
Description: Remove the specified subscription from the list of subscriptions of the selected query.

REMOVE QUERY DELETE to /BufferedEventQuery/{queryId}
Description: Remove the query and the associated buffer from the system.

> ??? Swagger definition of the implemented rest API

Using instances of *NotificationRule*, Unicorn sends notifications to the recipients (see Figure 19). In our scenario, the messages must be sent to Camunda, more specifically to a specific process instance within the engine. The *Event Correlation Service* within Camunda is responsible for relating incoming message events to process instances. One way to enforce the correlation is by inserting the process instance

identifier and the message name inside the message. For that reason, message name and process instance id must be provided when adding a subscription in the event engine. Unicorn has been adopted to send notifications to Camunda including the required correlation information. A sample notification for a eurotunnel delay event is shown in Listing 2.

> ref/footnote camunda documentation

Listing 2: Example of a JSON notification sent by UNICORN

```
{
  "messageName":"eurotunnelDelay",
  "processInstanceId":"274a876f-aed7-4a1a-916b-e85a0c2416f7",
  "processVariables": {
    "eurotunnelDelay":{"value":"60", "type":"Integer"}
  }
}
```

> can i reference another part of the thesis? could be mentioned in buffered event handling

After implementing the necessary extensions to the event engine, it is the task of the process engine to connect the extended process model to the buffered event handling API. The adaptations to the process engine are presented in the following section.

7.2   EVENT SUBSCRIPTION HANDLING IN CAMUNDA

through the bpmn extension for flexible event subscription, the information necessary to register event queries in a cep platorm is available in the bpmn model. **??** outlines how the business process engine must be adapted to execute the operations for subscription handling. Camunda is an open-source business process engine with support for the latest version of the BPMN. Further information is provided in **??**

> write from keypoints

?? depicts the architecure of Camunda, highlighting the modified parts in gray. for our purposes we consider the core components execution engine, model repository, correlation serviceand the modeler the tasks of them are...

7.2.1   *Employing ExecutionListeners in a Process Engine Plugin*

the platform is designed to offer customizability (source) a core concept to execute own code during process execution are Execution Listeners. they can be inserted straight into a bpmn model using the camunda modeler as demonstrated in **??** one of the main goals of introducing the bpmn extension is to get rid of the necessity to explicitly
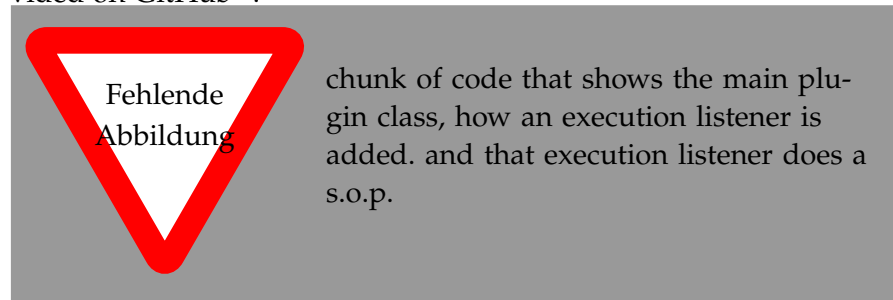
model the subscription in the process (unless an explicit subscription task shall be used). instead subscriptions shall be managed automatically by the process engine, solely based on the modeled attributes.

To achieve this kind of behavior, Camunda offers the concept of a Process Engine Plugin (PEP) to register global execution listeners. The plugin is a seperate software module that implements the ... interface. (ref) it is activated by adding an entry ... in the process engine config. within the implementation of the plugin, it is possible to intercept the process engine execution at predefined points. the engine itself implements an event-oriented architecture: significant milestones during engine executions fire an event that execution listeners can react on. Some of the available events are...

[23] and [27] have chosen to directly adapt the source code of camunda. more precisely, they propose to modify the Behavior class to execute additional code when a BPMN element starts executing. In this work, we implement a Process Engine Plugin as it allows a clearer, more understandable approach to adopting the execution behavior. especially when it is only necessary to execute additional operations and not modify or delete existing code. the PEP furthermore facilitates re-usability across environments and different versions of Camunda.

..ref illustrates how a pep can be used to execute a piece of code on process instantiation.

A helpful example for using a BPMN Parse Listener is also provided on GitHub [3].

Fehlende Abbildung

chunk of code that shows the main plugin class, how an execution listener is added. and that execution listener does a s.o.p.

from Section 5.1.1

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

7.2.2 *Managing event Subscriptions at Runtime*

The Process Engine Plugin enables us to execute custom java code during process deployment, instantiation and execution. By this means, event subscription and un-subscription can be implemented.

---

3 see   GitHub: *camunda-bpm-examples/process-engine-plugin/bpmn-parse-listener*, https://github.com/camunda/camunda-bpm-examples/tree/master/process-engine-plugin/bpmn-parse-listener

Before event subscription and un-subscription itself can be addressed, it must be ensured that the subscription information is read from the bpmn model. when a process gets deployed in Camunda, the BPMN-file is read and stored in an internal Java-Object representation. Custom BPMN extension elements can be accessesd using an in-built generic XML model API. It allows to read and modify any element in the XML model read from file. The API-call to read the CEP query from the BPMN model is shown in ..ref

Fehlende Abbildung   xml model api call to read the cep query and s.o.p. it
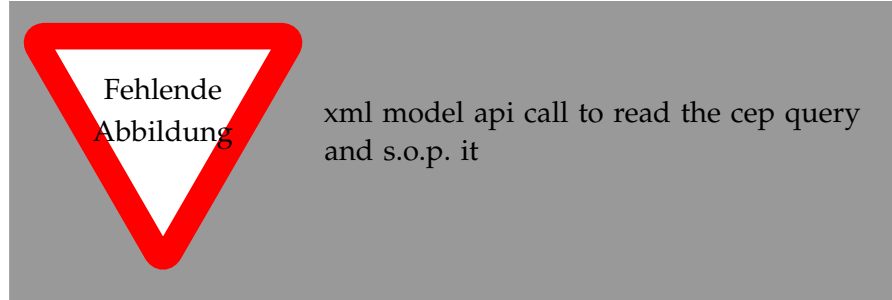
- event correlation > SubscriptionManager: memorize the subscriptionIDs for a process instance

RELATED WORK

The field of Event-driven Business Process Management (EdBPM) has developed from connecting CEP and BPM in an attempt to increase the quality and performance of business processes. EdBPM must be viewed from different perspectives. In event-driven business process monitoring and business activity monitoring, the process engine is solely an event producer that publishes information to an event engine. A considerable amount of research was undertaken to make use of CEP to detect process violations and execution performance. **??** BAM, process mining... [16] In [20], the authors did a comprehensive survey of research efforts in the area event-driven business process management and conclude that... > they point out that most research is in this area

...

This work has investigated how a flexible event subscription handling can be incorporated in BPMN models. It is therefor contributing towards a facilitation

roughly: event-driven business process management (anything that correlates bpm and cep)

Research in the area of event-driven business process management has been undertaken for several years now.

[27] makes use of events to control process executions and considers event subscription as an essential step in the process. However, the subscription time cannot be chosen flexibly.

[23] has been the starting point for this thesis, not only acknowledging the discrepancy between the event occurrence time in the real world and the event subscription time as interpreted from the BPMN standard, but also incorporating event subscription in the BPMN model. Inspired by their work, this thesis revisits the topic from scratch however:

[6]: investigations on the connection between events and process models, leading to [3]: extension of bpmn for receiving events, process variables in the query

similar to chapter 4: [2]: includes the streams and stream processing right into the process model same: [4]

(1)

> business activity monitoring

(2) - edpc: baumgraß, re-eval decisions - eda [7]: Process Instantiation, talks a lot about subscription, but no flexible subscr time, nobuffer, only for instantiation

[1]: Integrating Complex Events for Collaborating and Dynamically Changing Business Processes; talk about un/subscription, extend ws-bpel with subscription, reporting, patterns, expression language, ..; scope, time of subscription and event buffering not discussed also: [17]

[21]: acknowledge the lack of usability in CEP and address it by applying bpmn as graphical support for the definition of cep patterns [11]: generate queries from new graphical notation, anbieterunabhängige Modellierung von EdBPM

- use case implementations [9]: event-driven manufacturing process;

- correlating events to processes

- maybe: persisting events in cep platforms | or delayed delivery of events > [28]: Event data warehousing for complex event processing and [5]:Event-Driven services: Integrating production, logistics and transportation also [22]: historic data in pub/sub > but if i want to have that historic information, than I would have to issue a subscription some time before. I integrate everything in the process model

- event buffering for the application in a BPM environment may not be confused with internal buffering techniques used within event engines to perform load shedding, even out short term load peaks.

CONCLUSIONS

9.1 DISCUSSION

- discussion: > it remains the problem, that epl knowledge is not available in process design, a problem that has been addressed in ... > while the event buffering can now be controlled from the process model level, events must still be acquired by the event engine before (set up event types and make sure that the events are pushed to the engine) > the concept tries to bridge the gap between event persistence/historic events and real-time event processing by arguing that event occurrences are kept for a limited time and can therefor still be treated as events. A contra argumentation might be that an event becomes a simple piece of information as soon as it is not instantaneously consumed. hence it should not be modeled as an event but simply as a data object, using a data store > many design decisions have been begründet with an improved usability though no empirical basis was available. Reasoning was only based on literature and chats with a small group of fellow researchers

   - it can be argued that at process design time you dont care about the subscription time itself, but about the maximum age you can accept for your events. this would connect the lifetimepolicy and the subscription time into one value. but make the processing more complex and more fuzzy

9.2 FUTURE WORK

- this work attempts to provide a standard for handling event subscription in bpm architectures. given the necessity and the repeated attempts to address to topic, it would be a reasonable next step to discuss available solutions and start providing a foundation that is accepted and reused across the bpm community. No matter if based on the solutions of this work or not. > it must be evaluated if that's the way event buffering shall be used. > prove the value of flexible event subscription in an industry case study - future work: event data from other events could be used as historic data to allow the access even before process deployment

complete xml example of a BPMN model using the extensions

[1] Rainer von Ammon, Thomas Ertlmaier, Opher Etzion, Alexander Kofman, and Thomas Paulus. "Integrating complex events for collaborating and dynamically changing business processes." In: *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*. Springer. 2010, pp. 370–384.

[2] Stefan Appel, Pascal Kleber, Sebastian Frischbier, Tobias Freudenreich, and Alejandro Buchmann. "Modeling and execution of event stream processing in business processes." In: *Information Systems* 46 (2014), pp. 140–156.

[3] Anne Baumgraß, Mirela Botezatu, Claudio Di Ciccio, Remco Dijkman, Paul Grefen, Marcin Hewelt, Jan Mendling, Andreas Meyer, Shaya Pourmirza, and Hagen Völzer. "Towards a Methodology for the Engineering of Event-Driven Process Applications." In: *Business Process Management Workshops: BPM 2015, 13th International Workshops, Innsbruck, Austria, August 31 – September 3, 2015, Revised Papers*. Springer International Publishing, 2016, pp. 501–514.

[4] Biorn Biornstad, Cesare Pautasso, and Gustavo Alonso. "Control the flow: How to safely compose streaming services into business processes." In: *Services Computing, 2006. SCC'06. IEEE International Conference on*. IEEE. 2006, pp. 206–213.

[5] A Buchmann, H-Chr Pfohl, Stefan Appel, Tobias Freudenreich, Sebastian Frischbier, Ilia Petrov, and Christian Zuber. "Event-Driven services: Integrating production, logistics and transportation." In: *International Conference on Service-Oriented Computing*. Springer. 2010, pp. 237–241.

[6] Cristina Cabanillas, Anne Baumgrass, Jan Mendling, Patricia Rogetzer, and Bruno Bellovoda. "Towards the Enhancement of Business Process Monitoring for Complex Logistics Chains." In: *Business Process Management Workshops: BPM 2013 International Workshops, Beijing, China, August 26, 2013, Revised Papers*. Springer International Publishing, 2014, pp. 305–317.

[7] Gero Decker and Jan Mendling. "Instantiation semantics for process models." In: *BPM*. Vol. 5240. Springer. 2008, pp. 164–179.

[8] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management*. Springer, 2013, pp. I–XXVII, 1–399. ISBN: 978-3-642-33142-8.

[9]   Antonio Estruch and José Heredia Álvaro. "Event-driven man-ufacturing process management approach." In: *Business Process Management* (2012), pp. 120–133.

[10]   Opher Etzion and Peter Niblett. *Event Processing in Action.* Man-ning Publications Company, 2010, pp. I–XXIV, 1–360. ISBN: 978-1-935182-21-4.

[11]   Stefan Gabriel and Christian Janiesch. "Konzeptionelle Modell-lierung ausführbarer Event Processing Networks für das Event-driven Business Process Management." In: *Modellierung 2016. Lecture Note in Informatics*. Vol. 254. 2016, pp. 173–180.

[12]   Camunda Services GmbH. *Camunda BPM platform - overview.* 2017. URL: https://camunda.com/bpm/features/.

[13]   Nico Herzberg, Andreas Meyer, and Mathias Weske. "An event processing platform for business process management." In: *En-terprise Distributed Object Computing Conference (EDOC), 2013 17th IEEE International*. IEEE. 2013, pp. 107–116.

[14]   EsperTech Inc. *EsperTech - Esper*. 2017. URL: http://www.espertech.com/esper/.

[15]   *Information technology — Object Management Group Business Pro-cess Model and Notation*. Standard. International Organization for Standardization, July 2013.

[16]   Christian Janiesch, Martin Matzner, and Oliver Müller. "Be-yond process monitoring: a proof-of-concept of event-driven business activity management." In: *Business Process Management Journal* 18.4 (2012), pp. 625–643.

[17]   Matjaz B Juric. "WSDL and BPEL extensions for Event Driven Architecture." In: *Information and Software Technology* 52.10 (2010), pp. 1023–1043.

[18]   Martin Kleppmann. *Making Sense of Stream Processing*. O'Reilly Media, Inc., 2016.

[19]   Felix Kossak, Christa Illibauer, Verena Geist, Jan Kubovy, Chris-tine Natschläger, Thomas Ziebermayr, Theodorich Kopetzky, Bernhard Freudenthaler, and Klaus-Dieter Schewe. *A Rigorous Semantics for BPMN 2.0 Process Diagrams*. 1st. Springer, 2014.

[20]   Julian Krumeich, Benjamin Weis, Dirk Werth, and Peter Loos. "Event-Driven Business Process Management: where are we now?: A comprehensive synthesis and analysis of literature." In: *Busi-ness Proc. Manag. Journal* 20 (2014), pp. 615–633.

[21]   Steffen Kunz, Tobias Fickinger, Johannes Prescher, and Klaus Spengler. "Managing Complex Event Processes with Business Process Modeling Notation." In: *Business Process Modeling Notation: Second International Workshop, BPMN 2010, Potsdam, Germany, October 13-14, 2010. Proceedings*. Springer Berlin Heidelberg, 2010, pp. 78–90.

[22]   Guoli Li, Alex Cheung, Sh Hou, Songlin Hu, Vinod Muthusamy, Reza Sherafat, Alex Wun, H-A Jacobsen, and Serge Manovski. "Historic data access in publish/subscribe." In: *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*. ACM. 2007, pp. 80–84.

[23]   Sankalita Mandal, Matthias Weidlich, and Mathias Weske. "Events in Business Process Implementation: Early Subscription and Event Buffering." In: *BPM 2017 Forum, accepted for publication* (2017).

[24]   Michael zur Muehlen, Jan Recker, and Marta Indulska. "Sometimes Less is More: Are Process Modeling Languages Overly Complex?" In: *Proceedings of the 2007 Eleventh International IEEE EDOC Conference Workshop*. EDOCW '07. IEEE Computer Society, 2007, pp. 197–204.

[25]   Inc. Object Management Group. *OMG | Object Management Group*. 2017. URL: http://www.omg.org/.

[26]   OMG. *Business Process Model and Notation (BPMN), Version 2.0*. Object Management Group, Jan. 2011. URL: http://www.omg.org/spec/BPMN/2.0.

[27]   Luise Pufahl, Sankalita Mandal, Kimon Batoulis, and Mathias Weske. "Re-evaluation of Decisions Based on Events." In: *Enterprise, Business-Process and Information Systems Modeling: 18th International Conference, BPMDS 2017, 22nd International Conference, EMMSAD 2017, Held at CAiSE 2017, Essen, Germany, June 12-13, 2017, Proceedings*. Springer International Publishing, 2017, pp. 68–84.

[28]   Heinz Roth, Josef Schiefer, Hannes Obweger, and Szabolcs Rozsnyai. "Event data warehousing for complex event processing." In: *Research Challenges in Information Science (RCIS), 2010 Fourth International Conference on*. IEEE. 2010, pp. 203–212.

[29]   A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. 2nd. Prentice-Hall Inc, 2007.

[30]   Mathias Weske. *Business Process Management - Concepts, Languages, Architectures*. 2nd. Hasso-Plattner-Institute (HPI), University of Potsdam, Potsdam: Springer, 2012.

DECLARATION

Put your declaration here.

Potsdam, August 2017

<div style="text-align: right;">
_____

Dennis Wolf
</div>