LISTE DER NOCH ZU ERLEDIGENDEN PUNKTE

DENNIS WOLF

FLEXIBLE EVENT SUBSCRIPTION IN BUSINESS PROCESSES

FLEXIBLE EVENT SUBSCRIPTION IN BUSINESS PROCESSES

DENNIS WOLF

< Any Subtitle? >

August 2017 – version 1

Dennis Wolf: *Flexible Event Subscription*
in Business Processes, < Any Subtitle? >, © August 2017

ABSTRACT

Business process models have become an essential tool in organizing, documenting and executing company work flows 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 modeling 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 event subscription time in process models and derives new requirements for process modeling 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 behavior 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

Im heutigen Unternehmensumfeld sind Geschäftsprozessmodelle ein anerkanntes Werkzeug um Arbeitsabläufe zu organisieren, zu dokumentieren und automatisiert auszuführen. Complex Event Processing (CEP) kann dabei verwendet werden um die Flexiblität und Effizienz der Prozesse zu verbessern. Die Kommunikation mit CEP Plattformen folgt dabei dem publish-subscribe Muster, allerdings wird es von den wichtigsten Prozessmodellierungssprachen bisher nicht explizit beachtet.

Am Beispiel von BPMN 2.0 wird in dieser Arbeit zuerst der Bedarf nach einer flexiblen Nutzung von Event subscription erläutert, wovon konkrete Anforderungen an Modellierungsstandards abgeleitet werden. Es wird im Anschluss untersucht, inwiefern diese Anforderungen in BPMN untersützt sind, woraufhin zusätzliche Mängel ausgearbeitet werden.

Auf Basis dieser erweiterten Anforderungen wird anschließend eine Erweiterung zum BPMN Meta Model präsentiert, welche die Modellierung der subscribe-Operationen in Geschäftsprozessen ermöglicht. Dabei ist besonderer Wert auf den Zeitpunkt des Abonnierens gelegt, um den erarbeiteten Anforderungen gerecht zu werden. Zur Evaluierung des Konzepts wird zuletzt eine Referenzimplementierung unter Nutzung von Camunda und Unicorn vorgestellt.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Do I need a 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

EdBPC Event-driven Business Process Control

EOS Event Occurrence Scenario

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

1 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 (BPM) has been chosen as one of the tools to help stay competitive. Especially large enterprises, but also small to medium businesses, formalize their work flows in business process models to allow their automatized execution and management, archiving and documentation.

With Business Process Technology in constant progression, the opportunities that the field has to offer are ever growing. Since recent years, many efforts have been dedicated towards bringing together business processes and CEP. 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. Event Engines are designed to process thousands of events each second while supporting the integration of multiple event types and sources to derive new information. By having conglomerated information available almost in real time, companies can react quickly on complex situations within their organization or anywhere on the world. The connection of events and business processes has developed into an own discipline, Event-driven Business Process Management, which is performed from two perspectives. During execution, business processes can take the role of an event producer and publish information about their status, errors or incidents. Processed through complex event processing, this information can be utilized for business activity monitoring which aims at providing timeliness and effectiveness of operational business processes. In a second field of research, processes consume events in order to influence and control the projected work flow. By that means, events heavily increase the capabilities and flexibility in process flows. They can enable intra-organizational communication between processes or business departments, but also allow to respond to external situations within seconds and milliseconds.

The Business Process Model and Notation (BPMN), the most prominent industry standard for representing business processes, natively supports the use of events in numerous fashions. Event-related elements are a main building block of the modeling language and can, for instance, be used for instantiating processes, communicating between process participants or to support decisions. Thereby, event-driven process control combines the advantages of Business Process Management and Complex Event Processing.

1.1 MOTIVATION

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. The 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. As event processing has become an essential part of BPM, there is a need for the available modeling frameworks to consider these interaction patterns.

This work investigates the aspect at the example of the BPMN, where *Intermediate Events* enable event-based process control between the start and the end of a process. However, the BPMN specification does not specifically consider the publish-subscribe work flow and consequently provides only limited capabilities to represent event subscription and un-subscription operations in business process models. From the execution semantics of intermediate events, the common interpretation is that the subscription to an event source implicitly happens when an event is enabled, the un-subscription on termination of the element [**Pufahl2007**, 32]. Given that the subscription to an event must take place strictly before consumption, the time to listen to an event is limited to only a segment of the process execution. Naturally, in event processing, a large number of participants can be involved to cause a complex event. This high level of distribution and the inclusion of external sources implies the inability to control the time of event occurrence.

As defined in the BPMN specification, an intermediate catch event will wait for the associated event to occur. In case the event does not occur or only after a significant amount of time, this behavior leads to a delay or even a complete halt of the process execution. While some process designs require just these semantics, in others they cause an unnecessary reduction of efficiency and reliability. This behavior is further illustrated by the help of two example scenarios in Section 3.1. An earlier subscription to an event source would increase the timespan in which events can be received and thus reduce the probability of missing events. This motivates the investigation of mechanisms to more flexibly incorporate subscribe operations in business processes.

1.2 CONTRIBUTION & STRUCTURE

Aiming towards the extension of the BPMN modeling standard for the flexible use of event subscription, this thesis provides three main contributions: (1) From the analysis of motivating scenarios and the current state of research, an initial set of requirements (Section 3.3) is derived in the course of Chapter 3 and evaluated against current BPM

solutions in Chapter 4. The requirements define the functionality necessary for representing event subscription mechanisms in business processes models. They are supplemented by a list of shortcomings of BPMN (Section 4.3), which yields from the assessment of its meta model against the specified requirements.

(2) Based on the requirements and identified shortcomings of current solutions, Chapter 5 presents an extension to the notation and meta model of the BPMN including an XML-specification and a description of the expected execution semantics. The extension enables the incorporation of subscription-related information in the process model, allowing to flexibly chose the time of event subscription (Section 5.1). The concept hides the necessary underlying infrastructure tasks from the model by requiring automatic subscription handling within the process engine. Section 5.2 states the changes necessary to the behavior of the process engine and the event processing module.

(3) Finally, Chapter 6 evaluates the presented concept through a reference implementation at the example of the Camunda business process engine and the Unicorn event processing platform. The resulting BPM system is capable of handling the extended BPMN process models, automatically handling event subscription, un-subscription and event buffering.

2

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, 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 [43] 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. [43, p. 7]

The life-cycle of a business process can be described by four 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 life-cycle starts over by entering the design phase once again. The *process un-deployment* is performed if necessary, so that no new instances of the old process can be started [43, p. 11 ff.]. A similar life-cycle is described by Dumas in [16].

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.

2.1.1 *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

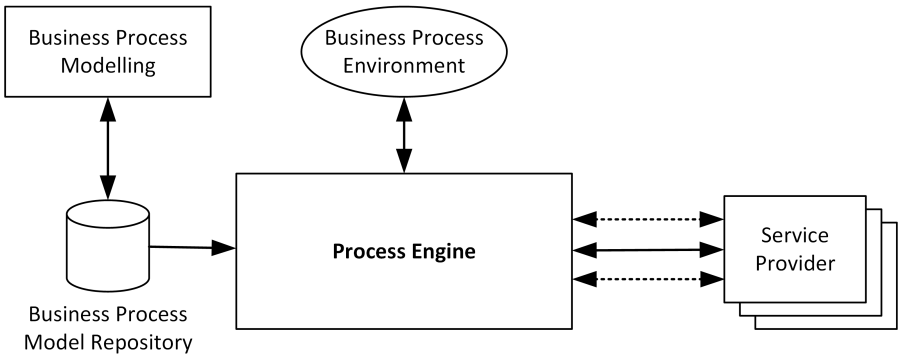


Figure 1: Business process management systems architecture model (see [43], p. 120)

*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, growing number of process engines is available on the market, including solutions from IT giants like SAP, IBM and Oracle. In this work, *Camunda BPM* [19] 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 6 will provide further details about the engine architecture and extension mechanisms.

2.1.2 *Business Process Model and Notation (BPMN)*

A generic meta model of business process models is proposed in [43]. According to the model, all processes are composed of nodes and edges with each node representing either an activity model, an event model or a gateway model. The complete definition is provided in *Definition 3*.

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. [43], 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.
- $type : N_G \to C$ assigns to each gateway model a control flow construct.

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. [28] organize some of the more popular languages as follows: A subset of them are focused on the control flow of business processes, for instance BPMN [37], Yet Another Workflow Language and Petri Nets; some focus on object-orientation, like the Unified Modeling Language (UML) activity diagrams and use case diagrams; some are data-flow oriented, e. g. the Structured Analysis and Design Technique.

Among these, the Business Process Model and Notation (BPMN) has developed into a widely-adopted industry standard, also becoming ISO-standard in 2013 [23]. The specification is developed by the Object Management Group [36] and now available in version 2.0 (January 2011) after being first released in January 2008. Whenever referring to BPMN in this work, version 2.0 of the standard is meant. BPMN can be understood as an extension to the abstract business process meta model 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 [43] points out. As emphasized by Muehlen, Recker, and Indulska [35], 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.

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 variations. 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

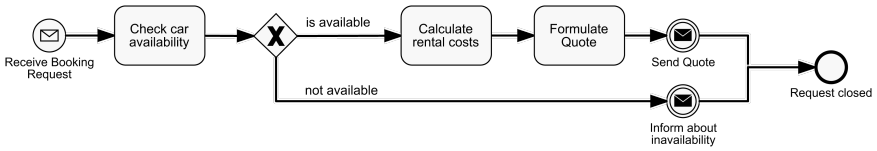


Figure 2: Simple BPMN model of issuing a quote for car rental

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 tasks 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.

The example illustrates the basic use of activities, events and gateways in BPMN. However, for each type of element, activities, gateways and events, BPMN offers a variety of different kinds. The Task elements shown in the example (e. g. *Check car availability*) are the most basic type of activities, representing *a unit of work*. The nature of a task can be further specified using task types, all activities can be additionally annotated with activity markers. Each of these modifications describes the activity semantics in a more detailed fashion. Apart from the utilized XOR-gateway (the execution-flow will proceed in exactly one of the outgoing branches), there is for example the parallel gateway, activating all outgoing branches. Moreover, the Event-based Gateway, which can be followed by events or tasks, and continues along the branch of the first element that completes. Last but not least, message events are used to represent the action of sending information in the form of a message to a certain recipient. Other available event types are, for instance, timer, signal or error events. Events can occur as *Start Events* (e. g. *Receive Booking Request*), where they trigger the instantiation of a process. Furthermore as intermediate events (e. g. *Send Quote*) or as end events, being fired when the process terminates. They can be *catching*, i. e. receiving a trigger, or *throwing*.

Multiple participants taking part in a process can be expressed using *pools*, which can be sub-divided into *swimlanes*. Interactions between different pools take place using message flows, which are depicted by a dashed arrow. An example of such a *Collaboration Diagram* is shown in Figure 3. It describes the previously discussed car rental process from the side of the customer, including the information exchange between rental company and customer. When the airline issues the booking confirmation of the flight, the customer has to book a rental car and a hotel. The hotel booking is modeled using

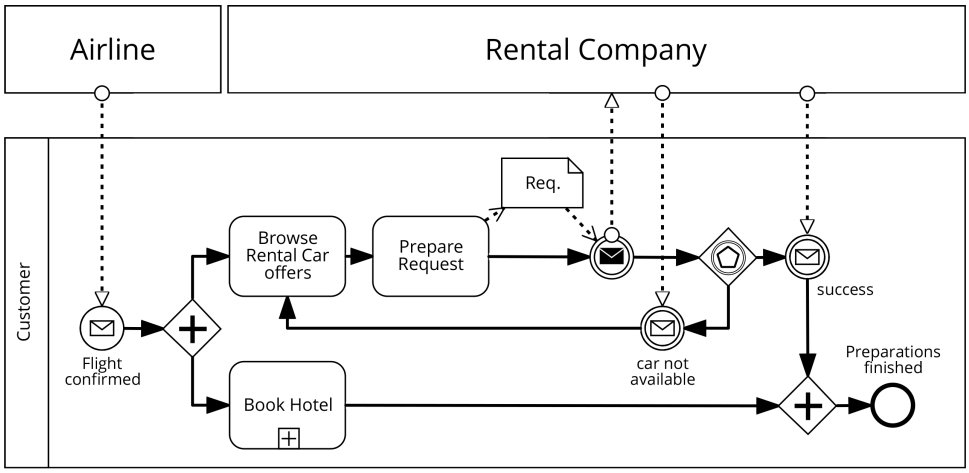


Figure 3: Rental car booking process with multiple participants

a collapsed sub-process to reduce the complexity of the model. After issuing the rental car request, an event-based gateway shows that the process is ready to consume any one of the two event messages. If the booking was successful, the control flow proceeds to the inclusive join gateway and the process can terminate once the hotel booking is also completed. If the car rental company informs about unavailability, the customer starts looking for an alternative car option. A data object is used to model the *Request* artifact as output of the *Prepare Request* task and as input to the message throw event.

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) [18]. More specifically, Etzion and Niblett define an event as *an occurrence within a particular system or domain; it is something that has happened, or is contemplated as having happened in that domain*. It is understood that events are of a certain *event type*, defining the attributes that each event instance of the event type is composed of. An attribute is described by a unique attribute name and a data type.

DEFINITION 4 (EVENT): An event is a tuple $e = (et, eventtime, c)$, whereas

- et is the event type
- $eventtime$ is the time the event happened

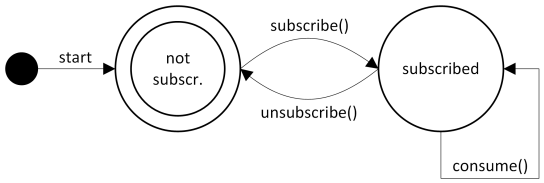


Figure 4: Publish-Subscribe workflow expressed in a state diagram

- c is the content of the event consisting of a set of key-value-pairs according to the content description cd of the corresponding event type et.

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(Figure 4). The consumption and un-subscription can only happen after the subscription. Once an un-subscription has been issued, no consumption can follow. [40]

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 [18]. 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.

STREAM PROCESSING To be able to cope with potentially large amounts of data, Complex Event Processing Platforms work on the basis of *stream processing*. In a traditional relational database, information is stored for an indefinite amount of time. When a user queries

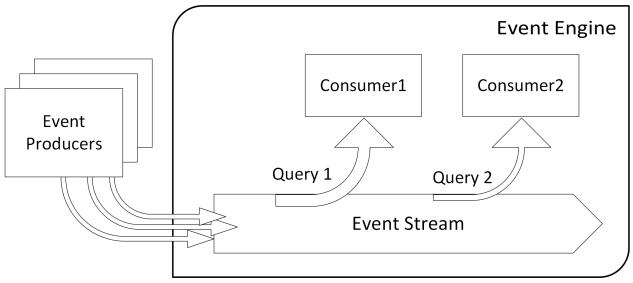


Figure 5: Stream processing concept applied in a CEP Engine

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. If an event matches a query, a notification is sent to the subscribed consumers. Figure 5 illustrates the concept. 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. [27]

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) [22], utilized in Esper-based event processing engines like the one employed in the presented reference implementation, Chapter 6. A simple event query in Esper looks as follows:

Listing 1: Sample Query in Esper EPL

```
SELECT occurrencetime, delay, delayreason
FROM eurotunnel.win:time(2 hour)
WHERE delay > 30
```

2.3 EVENT-DRIVEN BUSINESS PROCESS CONTROL

The disciplines of Complex Event Processing and Business Process Management are connected in Event-driven Business Process Management (EdBPM), which discusses the uses of events to enhance business processes [18]. There are two usage scenarios for events in BPM. In Business Activity Monitoring and analysis, events are published by the process engine and processed to obtain analytical information, for instance about process status, efficiency or errors. This thesis considers events from the second perspective, Event-driven Business Process Control (EdBPC). The field focuses on the business process as an event consumer. Within business processes, event-based communication can for example be used to exchange information between participants or to react to external events. While the BPMN supports a large variety of different events, we are going to investigate message events specifically. Whenever talking of an event-occurrence, it can be assumed that the term *event* implicitly refers to a message-event in the BPMN context.

EdBPC using external event sources is facilitated by connecting the business process engine to an event engine. As explained in Section 2.2, interactions with a CEP platform follow the publish-subscribe paradigm. In the BPM scenario, this means that a subscription to events must be present, so that events can be received by the process engine and correlated to a specific message element within a process instance [3]. The correlation has to be performed by the process engine on the basis of context information, for example a subscription identifier incorporated inside every CEP message. An interaction between process engine and event engine is illustrated in Figure 6 [32, , p. 13] at the example of the event engine Unicorn. Note that a different event engine might implement the work-flow slightly differently, but still follow the same general concept. In the given case, the process engine first issues a HTTP POST call containing the subscription query a notification-path. The path contains the address of the desired notification recipient in case a matching event occurs. The event engine answers that call with a unique identifier associated to that single subscription. As soon as an event occurs that matches the query, the event engine sends the query output along with the subscription identifier to the process engine which can correlate the event back to the associated instance. In the next chapter, we will discuss when exactly the event subscription is issued by the process engine.

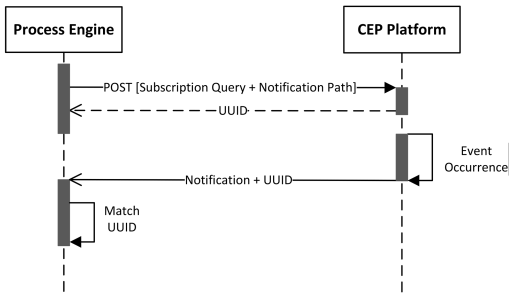


Figure 6: Even-subscription work-flow between process engine and event engine.

3 PROBLEM STATEMENT

Event subscription is an essential part of event-driven architectures and must consequently be considered when using events in business processes. While the use of external events is an active topic in research, most approaches only briefly discuss the subscription mechanism and use the BPMN semantics for orientation (e. g. [38] and [3]). Though event subscription is not directly addressed in the BPMN specification, the descriptions on intermediate events state:

"For Intermediate Events, the handling consists of waiting for the Event to occur. Waiting starts when the Intermediate Event is reached. Once the Event occurs, it is consumed." [37], p. 440

The usual interpretation of this excerpt is that the subscription to the event takes place as soon as the event gets enabled, the un-subscription when the event terminates. As pointed out by Mandal et. al. [32], these subscription semantics significantly limit the flexibility of using events in business processes and can cause undesired behavior and fault. Due to the strict temporal order between event subscription, occurrence and un-subscription as required by the publish-subscribe paradigm, any events that occur before the enabling of the event element will be ignored by the process execution. That reduces the timespan for events to occur to a potentially small part of the total execution time and means that crucial events might be missed which can delay or block a process execution unnecessarily.

In the following section, the problem is further illustrated at the example of two motivating scenarios. The observed situations then lead to the definition of *Event Occurrence Scenarios* and the derivation of a set of requirements that must be fulfilled by a mechanism for flexible event subscription in business processes.

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. It is illustrated, why the time of event subscription is of great importance which motivates to study the mechanics and implications of event subscription in business processes.

DELAY OF A LOGISTICS PROCESS The first example (Figure 7) is taken from the logistics domain and shows a truck transport that has to cross the English Channel. The truck driver receives the transport

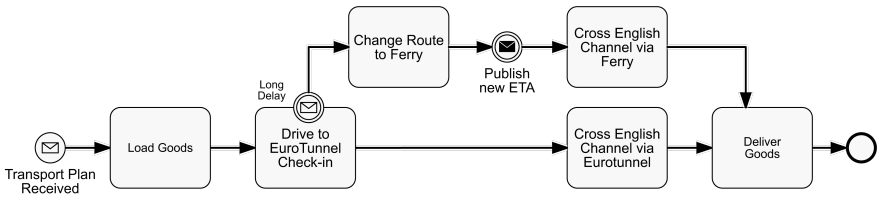


Figure 7: BPMN Model of a Logistics Process using events for route optimization (Example 1.1)

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 crosses the channel on the train and then delivers 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 multiple 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.

As interpreted from the BPMN specification, the subscription to the boundary event is issued as soon as the related activity is enabled. 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 that day and the train runs 3 hours behind schedule. The last information on the RSS feed was published at 2:35 pm. At that time the truck driver is still in the process of loading goods, finishing the activity at 2:40 pm. Following the process definition, the driver now departs towards the Eurotunnel check-in. The system publishes updated information at 3:15 pm, operations are still 2:30 h 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 8 is an extension of the the transport process. In logistics, it is common that a delivery cannot be accepted at an arbitrary time. In-

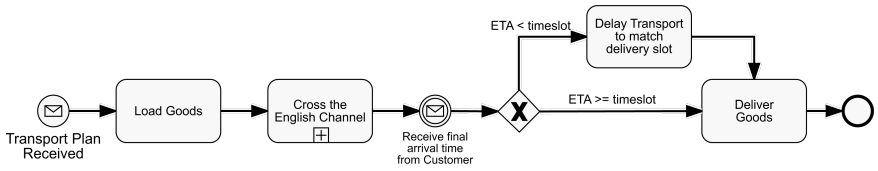


Figure 8: Transport via English Channel that is timed to a delivery slot (Example 1.2)

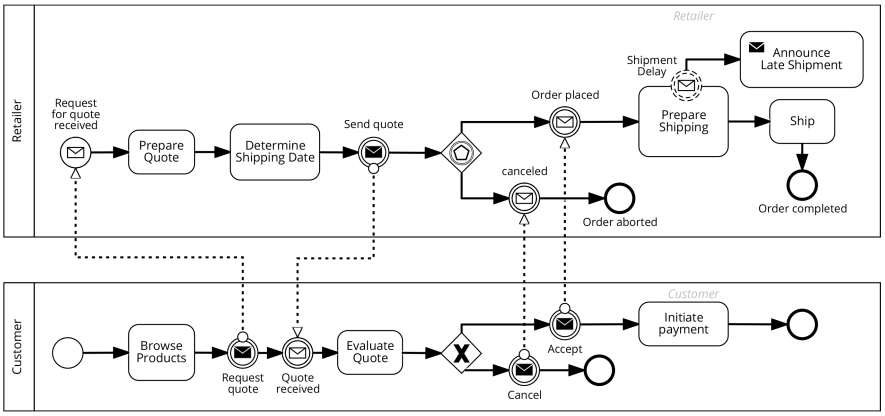


Figure 9: Model of a retail order management process (Example 2)

stead, 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 the drop-off point immediately. 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 enabled, in this case 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, i. e. 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.

UP-TO-DATE SHIPPING INFORMATION FOR AN ORDER A similar situation can be observed in the order-management process depicted in Figure 9. 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 retailer 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 logistics process in Example 1. At first we 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 8), 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 terminated. The Esper query to obtain delay information about the product with the id 123 looks as follows:

Listing 2: Esper EPL Query to obtain delay information for a product

```
SELECT newArrivalDate, delayAmount, delayReason, productId
FROM shippingdelay
WHERE productId = 123
```

Conforming to the definition of the process, the system will listen to shipment delays once the activity *Prepare Shipping* is enabled and therefor much later than possible. Any events that occur after the completion of *Prepare Quote* but before *Prepare Shipping*, cannot be considered in the process execution and the customer will not be informed about a possible late shipment. That will, for instance, concern events

that occur while the customer makes the decision about accepting or canceling the order.

The two presented examples have illustrated the complexity of using events in business processes, especially when all possible event occurrence times are taken into consideration. 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 is unrelated to the execution of that very process. Motivated by this complexity and the possible implications, it is the goal of this work to evaluate the capabilities of BPMN and to present a concept for the flexible handling of event subscription in business processes.

3.2 EVENT OCCURRENCE SCENARIOS AND TIME OF SUBSCRIPTION

Given the motivating examples, a generic set of *Event Occurrence Scenarios* is defined in this section. Each of the scenarios represents a real-world situation and process implementations need to be capable of handling them to avoid negative effects.

The dominant variable to consider is the event occurrence time. According to the BPMN specification, it is possible to catch an event if it occurs after the event element is enabled. As shown before, it is often impossible to control occurrence time and events do occur outside of the listening time intervals. An Event Occurrence Scenario (EOS) describes the time of event occurrence in relation to a specific step in process or engine execution. The life-cycle of a process within a process engine is explained in Section 2.1 and taken as reference. It is assumed that the process and event engine are configured and running. An event is considered to always occur before or after a life-cycle step or in between two consecutive steps. Given the life cycle steps *process deployment*, *process instantiation* and *event enabling*, the following occurrence scenarios are distinguished in this work:

EOS_1 Occurrence while the BPMN event element is enabled

EOS_2 The event does not occur

EOS_3 Occ. between process instantiation and the enabling of the BPMN event

EOS_4 Occ. between process deployment and process instantiation

EOS_5 Occ. before process deployment

An overview of the EOSs and the related life-cycle steps is provided in Figure 10, which uses a time-line to illustrate the possible event

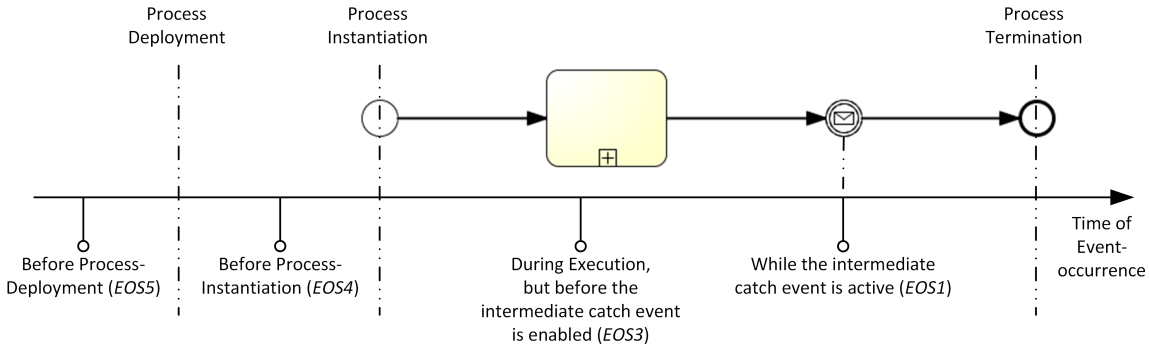


Figure 10: Event Occurrence Scenarios
Possible times of event occurrence in relation to a process execution life-cycle. *EOS2* (event does not occur) not illustrated.

occurrence times. The model deliberately excludes event occurrences after termination of the event element, because that would presume that there is no more interest in the event as the execution flow continued on another branch. Otherwise, the process will remain in waiting state until the event occurs.

As introduced in Section 2.2, there is a strict temporal order between event subscription, consumption and un-subscription. To catch an event that occurs as described in any of the EOSs, the related subscription operation has to be executed before the start of the time interval associated with the EOS. On that basis, the latest possible event subscription time can be derived from each event occurrence scenario and is as follows: *before process deployment* (EOS_5), *during process deployment* (EOS_4), *at process instantiation* (EOS_3) and when the *BPMN event element is enabled* (EOS_1).

It is important to highlight that the subscription to an event source can depend on additional context information or process data, which can be a significant limitation to the possible subscription time. That situation is illustrated in *Example 2* (Figure 9), where the subscription to the shipment delay information cannot be executed before the *Prepare Quote* activity is completed. More generally put, a *subscription dependency* is defined as follows:

SUBSCRIPTION DEPENDENCY The subscription operation is *dependent*, if any of its parameters, for instance the event query, contains a variable value that has to be determined at the time of subscription. The value might reference a piece of information from the process instance context, an engine-wide piece of data or external information. In case of a dependence, the subscription cannot be issued before the associated information is available.

If a subscription dependency resolves before the process execution reaches the event element, it should still be possible to flexibly chose

the subscription time in the process. For that reason, in addition to the four subscription-time options presented earlier, an additional option *at any time during process execution* must be available.

3.3 REQUIREMENTS DEFINITION

The previous sections have exemplified how the BPMN execution semantics and notation capabilities limit users when using events in business processes. Now, these deficiencies are formalized into an initial set of requirements additional to the features offered by BPMN. They build the foundation for enhancing BPM solutions towards flexible event subscription management (Chapter 5). Before that, the formal requirements will be used to evaluate the capabilities of current Process Management Solutions (Chapter 4), which results in an additional set of shortcomings *S1* to *S3* of current solutions presented in Section 4.3.

R1: SUBSCRIPTION FUNDAMENTALS

R1.1 SUBSCRIPTION TO AN EVENT SOURCE: To enable the reception of events from a publish/subscribe event source, the subscription operation must be part of the execution flow. In accordance with the obligatory temporal order of operations, the subscribe operation must happen before an event can be received and/or the un-subscription takes place.

R1.2 UN-SUBSCRIPTION: As soon as events from the source must not be received anymore, an un-subscription has to take place.

R1.3 AVAILABILITY OF INFORMATION: For each event that is used in a business process, all information necessary to execute an event subscription in the given execution environment must be available. That information generally includes the event query, the address of an event processing platform and auxiliary information to establish the communication, for instance authentication data.

R1.4 VARIABLES IN EVENT QUERIES To adapt an event subscription to execution- or time-specific information, variables can be utilized as part of the query string. At the time of event subscription, these variables must be replaced their current values before the subscription is executed.

R2: EVENT SUBSCRIPTION TIME

R2.1 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.

R2.2 FLEXIBILITY: The time of subscription can be influenced independently from the place the event element takes in the model. Timing options are made available to catch events according to any of the event occurrence scenarios EOS1, EOS3, EOS4 and EOS5. Consequently, the options must include but are not limited to the subscription before process deployment, during pr. deployment, at process instantiation, subscription at an arbitrary but explicit time during process execution or when the event element is enabled. Thereby, also events that occur after the time of subscription, but before the event element is enabled shall be available to be consumed. (cf. [32])

R2.3 AWARENESS OF SUBSCRIPTION DEPENDENCIES: The additional flexibility provided through *R2.2* is limited by the use of variables in event queries (*R1.3*) and the implicit dependencies on context information. The execution environment must only allow a subscription time after the resolving of all dependencies.

Dependencies can exist within a certain process instance, for example when an information from a local DataObject is referenced in a query. In more complex scenarios, the subscription might depend on data from other process instances or even other processes which must first be obtained.

R3: EVENT BUFFERING

R3.1 BUFFERING PRINCIPLE: As introduced by *R2.2*, an indefinite time can pass between the time of subscription and the consumption of the event. While the subscription operation generally assures that the event is received, it is necessary to temporarily store it until the consuming element is reached.

This temporary storage is referred to as an *Event Buffer*. The storage must be designed to fulfill the semantics demanded by *R2.2* which means that any event that occurs after the time of event subscription must be available to consume.

R3.2 BUFFER SCOPE: The flexible event subscription time (*R2.2*) allows an event subscription after process instantiation, but also before process instantiation or deployment. It can be inferred that an event buffer operates either in the scope of a process instance, a process definition, or in the scope of the complete execution environment. To implement all event subscription times stated in *R2.2*, the execution environment must support event buffering in each of the three scopes. The provided concept must make clear to the user, which scope is referenced at any point in time.

R3.3 BUFFER POLICIES: The behavior of a buffer in any occurring situation must be specific. The parameters necessary to specify

the behavior are referred to as *Buffer Policies*, following the notion chosen by Mandal et. al. [32].

R3.3.1 LIFESPAN POLICY: Given that the time between the event subscription and its consumption may be indefinitely long, process designs can require to limit the maximum time an event is held in the buffer.

R3.3.2 CONSUMPTION POLICY: Requirement *R3.2* implies that buffers can be shared among process instances of the same process and other processes. In this scenario, it needs to be defined if an information is consumed from the buffer upon retrieval, or if the information remains in the buffer for other participants to access.

R3.3.3 SIZE POLICY: Provided that multiple events can occur between the time of subscription and the enabling of the event element, it must be specified for each buffer, how many events should be stored for later consumption.

R3.3.4 RETRIEVAL ORDER POLICY: If multiple events are stored in the buffer (*R3.3.3*), it must be defined, in which order the events are provided for consumption.

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 additional requirements *R1* to *R3* for process management solutions have been presented. In this section, we take a closer look at the capabilities of current solutions with regards to the event occurrence scenarios (see Section 3.2) to get a better understanding of the issues that arise when working with event subscription in business processes.

The assessment will be carried out on the basis of the BPMN and Camunda, a state-of-the art and widely adopted business process engine (Section 2.1.1). Both tools shall be used *as they are*, that means without utilizing their comprehensive extension mechanisms. The main goal is to identify and illustrate the shortcomings of the current process technology stack by attempting to implement the requirements identified previously. We therefor first investigate options to handle the EOSs on the BPMN model level (Section 4.1) and then proceed to implementing early event subscription and event buffering in a set of Camunda processes (Section 4.2).

Finally, the findings are discussed in Section 4.3 which leads to the formulation of three shortcomings, *S1* to *S3*. They will be referenced in Chapter 5 in addition to *R1-R3* to develop a more refined subscription handling model.

4.1 BPMN MODELS IN PRESENCE OF THE EVENT OCCURRENCE SCENARIOS

According the BPMN specification, the listening for an event starts only when the event element is enabled. The start of the listening is interpreted as the time of event subscription. The motivating examples illustrate that process executions can be delayed or even blocked because of a late time of event subscription (Section 3.1).

In this section, I first describe for each EOS how a BPMN intermediate event behaves in presence of the given scenario. I then investigate if it is possible to create a BPMN model that is free from unnecessary delay in these situations, thereby evaluating requirement *R2*. The investigations are carried out on the basis of the abstract process model shown in Figure 11. It uses an intermediate message catch event which follows an arbitrary process flow, represented through a collapsed sub-process. The process terminates after the event is received. It is assumed, that the event can occur independently from

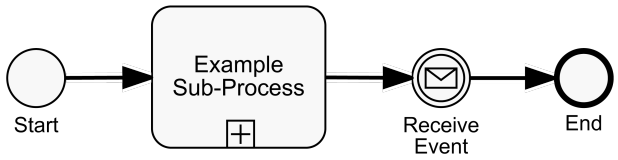


Figure 11: Generic process terminating after an intermediate message catch event

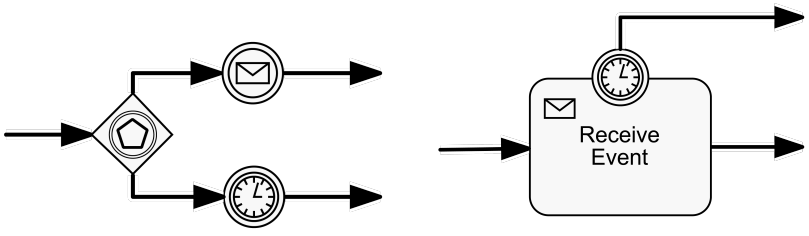
the preceding subprocess and therefor in accordance with any of the event occurrence scenarios. That enables us to investigate each EOS using this simple model. The event behavior is comparable with the *Eurotunnel Delay Event* considered in *Example 1.1* . Depending on the discussed EOS, the motivating examples will be used for further illustration of the situations.

EOS1: The event occurs while the catch element is enabled

The first scenario represents the case that is assumed by the BPMN 2.0 specification: The event occurs after the event element has been enabled and before it has terminated. It will be consumed by the catch event and the process flow can proceed without unnecessary delay. In this scenario, the use of a simple intermediate catch event is sufficient.

Notably, there always is a certain delay when consuming an event through an intermediate catch event. The BPMN specification itself states that *the handling consists of waiting for the Event to occur*. However, that delay can theoretically be infinitely small. We speak of an *unnecessary* delay in the context of flexible event subscription if an earlier subscription time would have meant a reduced waiting time for the event.

EOS2: The event does not occur



(a) Parallel timer event (b) Boundary timer event

Figure 12: Receipt of an intermediate message event, either interrupted by a boundary event (a) or parallel timer event (b).

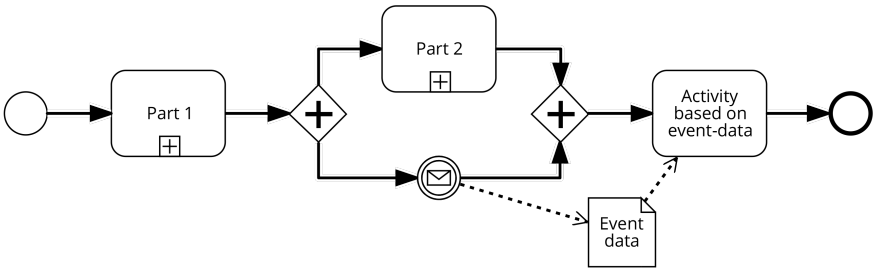


Figure 13: Event Element in parallel process flow

In certain situations an event might not occur at all. Given a basic event implementation like in Figure 11, 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. In *Example 1.2*, the process relies on the *final arrival time* information from the customer. If the event does not occur, for example due to a mistake in a factory work-flow, the process execution waits for the event indefinitely. To improve the process design, the logistics company decides that after waiting for the event for 30 minutes, an arbitrary delivery window shall be assumed. The waiting for the event shall be terminated so that the goods can be delivered immediately.

Figure 12 shows how this behavior can be implemented in two ways: (a) By the help of an event-based gateway which puts a timer event in parallel to the intermediate catch event: If the timer event occurs before the message event is consumed, the message event is terminated immediately and the process flow proceeds from the timer event. An alternative solution with equivalent semantics is depicted in (b), using a receive task and an interrupting boundary timer event. Once the timer event fires, the receive activity is canceled and the process continues along the outgoing flow of the time event. Any of the two model improvements will make sure that a process does not run into a lock if the expected event does not occur.

EOS3: 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 available for consumption. The execution will come to a halt at the event element as if the event did not happen at all.

To avoid a lock in this scenario, the intermediate catch event can be placed in parallel to the rest of the process flow using a parallel gateway. This is illustrated in Figure 13. 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 the event subscription at a specific point during process execution, part of the process must execute before reaching the Parallel Gateway. As modeled in Figure 13, the event may occur at any time during the execution of the collapsed sub-process *Part 2*.

EOS4 and EOS5: 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 (*EOS4*) and an occurrence time before the deployment of the process in the Process Engine (*EOS5*).

To create a process model that allows to catch an event before the process instance exists, three new elements are introduced in addition to the existing process referred to as *target-* or *original process*. The elements are: (1) An additional *Auxiliary Buffering Process* (or *buffering process*) that can catch an incoming event independently from the target process; (2) an *Event Buffer*, a temporary data-store that keeps event data until it is ready for consumption; (3) an *Auxiliary Event Delivery Process* (or *delivery process*), that retrieves events from the buffer and makes them available to the target process. The introduction of a temporary data store for events addresses *R3*. By introducing an additional process for listening to the event, the subscription time to the *Event Engine* can be before instantiation or even deployment of the original process and hence realizes *R2.2*. Figure 14 shows the interaction of the original process, the two auxiliary processes and the data-store.

THE EXECUTION FLOW To start listening for an event, the auxiliary buffering process has to be instantiated through a message start event containing the information necessary for the event subscription. Considering that the related process instance is not yet running, it is assumed that participants from the *IT-Department* start the buffering any time before process instantiation. It is also their task to stop the buffering when it not required anymore, for example when the process gets un-deployed.

The buffering process subscribes to the event engine through a throwing event. It then waits for an event to occur in a receive activity *Receive external event* and outputs the received event data to a DataObject. That object is then written to a persistent, global datastore *Event Buffer* by the service task *Write to buffer*. The design is able to handle multiple event occurrences, as the execution flow proceeds back to the receiving activity after writing to the buffer. The

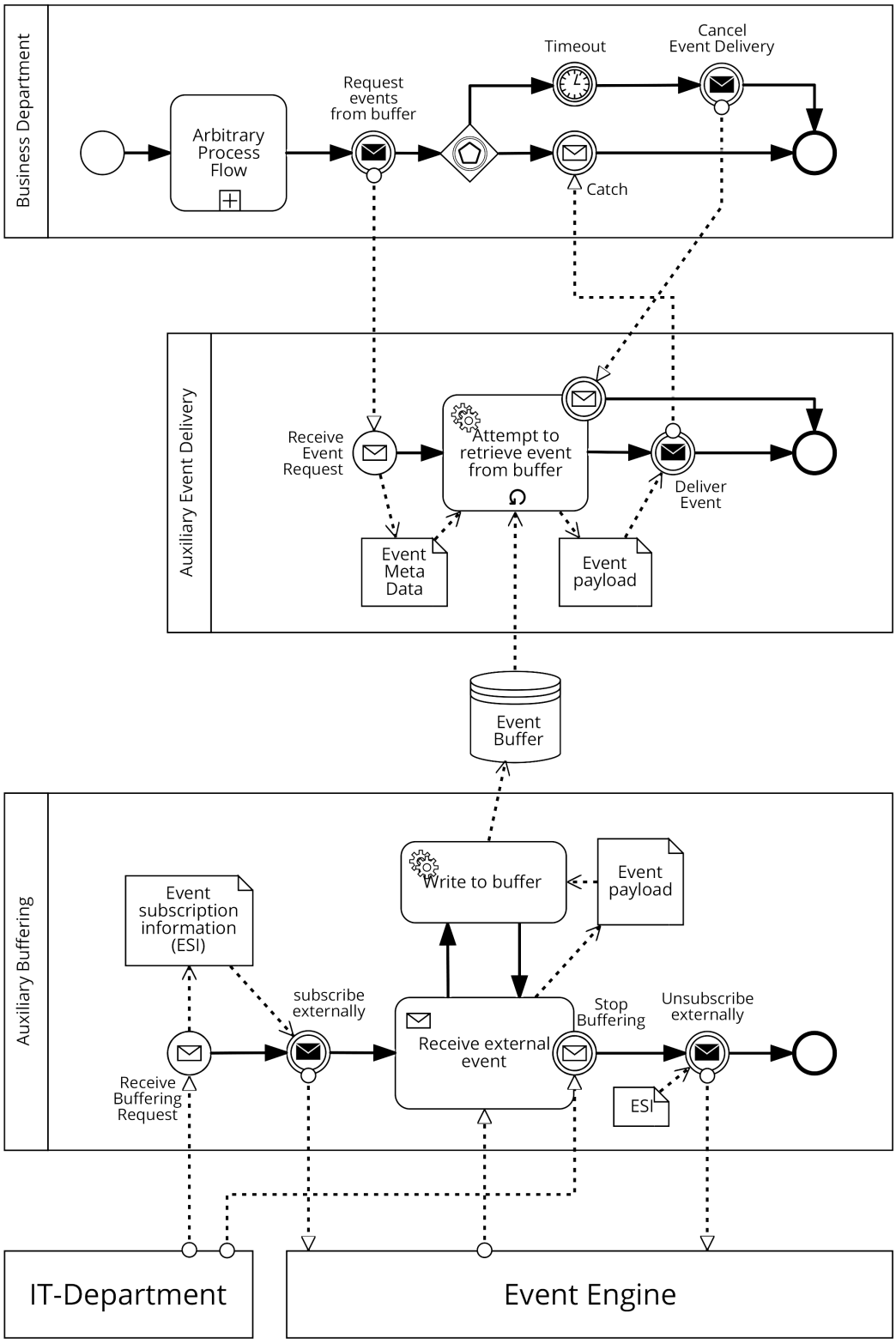


Figure 14: Event Buffering through an auxiliary buffering process

implementation of the service task decides about the exact semantics of the buffering and hence about requirement *R3.3*. For the sake of simplicity, it is assumed that the buffer can hold a maximum of 1 element (*Size Policy*) and the service task overwrites that element whenever a new one is available, specifying the retrieval order (R3.3.4). An implementation of the *Lifespan Policy* is not provided. The consumption behavior (*R3.3.2*) is defined by the service task of the auxiliary delivery process: After the retrieval of an event from the buffer it remains available.

An implementation of *R1* can be observed in the auxiliary buffering process, which models the subscribe and unsubscribe operation using an intermediate message throw event. The event subscription information is represented as a data object and used as input to the events, meaning that its contents are sent to the according API method of the event engine. It is assumed that the subscription information can be used for both operations, subscribe and unsubscribe.

After the buffering process is running, the original process can be instantiated. It executes the *Arbitrary Process Flow*, then a send event instantiates the *Auxiliary Event Delivery Process*, which tries to read from the event buffer and delivers the event to the original process if there is one available. The central looping activity will retry reading from the buffer until data becomes available. In the provided example, the original process will alternatively stop waiting for the message reception as soon as the time event fires. In that case, the delivery process is stopped by an additional message throw event and both processes stop. Note that the delivery process is designed to retrieve and deliver a single message from the buffer.

CONCLUSIONS Thanks to the complex interaction of the three processes, the original process can consume the desired event, even though the event occurs before process instantiation. Consequently, *EOS4* can be handled by the model. Moreover, the *Auxiliary Buffering Process* is not bound to a specific event, it works generically with any event information that is passed 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 *EOS5*. The buffering process can alternatively be started using an explicit message send event during process execution, similar to the *Explicit Subscription Task* introduced in [32]. By that means, the event subscription on process instantiation or at any time during process execution is also supported, which results in the fulfillment of requirement *R2.2*.

It remains to be highlighted that this approach will work for *EOS4* and *EOS5* only if the buffering process is instantiated before the associated event occurs. As there are no other systems in place to automatically instantiate the process, it must be assumed that the process is manually started for each event and each process model that requires

an early event subscription. Furthermore, the number of processes running in the execution environment increases significantly. An instance of the buffering process is required for each event element that ought to make use of an event subscription time before deployment. For each instance of the original process and each buffered event, an instance of the auxiliary event delivery process must be started.

More precisely: Given a process P1 with a number n_{bea} of event elements that subscribe after instantiation, and a number n_{beb} of elements that subscribe before instantiation of P1. Furthermore, the buffering process BP and the delivery process DP, then the number of instances $instances(P)$ with P being an arbitrary process is as follows:

- $instances(BP) = n_{beb} + instances(P) \times n_{bea}$
- $instances(DP) = instances(P) \times (n_{beb} + n_{bea})$, assuming that exactly 1 instance of each event is currently in waiting state.

That puts additional load on the process engine which can endanger the reliability of business-critical processes and makes the management of processes and their instances more complex. The downsides of the presented approaches are discussed in more detail in Section 4.3.

4.2 FLEXIBLE EVENT SUBSCRIPTION IN STANDARD CAMUNDA

The previous chapter has shown that it is possible to create BPMN models to match each of the Event Occurrence Scenarios, though for the scenarios *EOS4* and *EOS5* the solution becomes increasingly complex. In the next step I investigate the capabilities of Camunda as an example for a state-of-the-art business process engine. Camunda shall be used without any code customization, that means as offered through the deployment package. The solution presented in Figure 4.1 (*EOS4* and *EOS5*) has proven capable enough to handle all event occurrence scenarios, it is therefor the goal to implement a prototypical solution in Camunda in strong accordance with the BPMN model shown in Figure 14.

Two generic sample processes have been modeled for evaluate the functionality of the system. Figure 15 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 *EOS3*). 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 *EOS4* and *EOS5* (Figure 16) 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

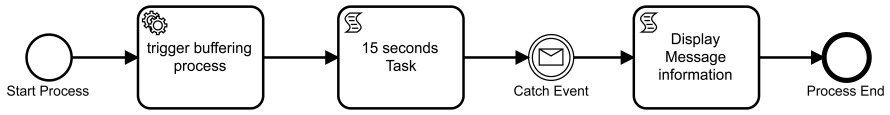


Figure 15: Generic Example Process in Camunda for *EOS3*

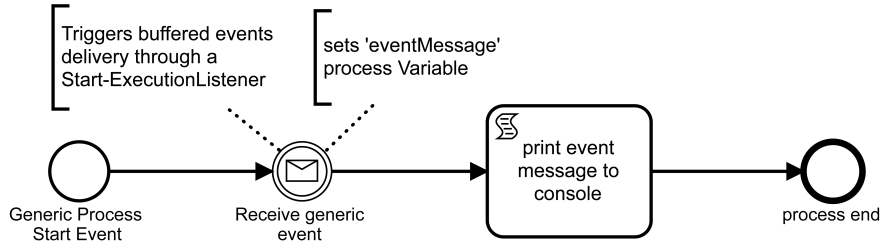


Figure 16: Generic Example Process in Camunda for the Event Occurrence Scenarios *EOS4* and *EOS5*

and last the process end event. Both figures show screen captures from 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*). A local MySQL database has been chosen for persisting the event data because it is freely available, easy to set up, offers standardized access via SQL queries and Java connectors. As complex event processing functionality itself is not required to demonstrate the use-case, the role of the *Event Engine* is taken by a basic web-service which was specifically implemented in Python. It exposes a *subscribe* method via HTTP and can be used to trigger event messages associated to registered subscriptions.

Figure 17 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 event engine. That call registers the event query in the platform, providing the process instance identifier and the message name. Based on this information, Camunda can correlate received events back to a specific process instance and message element. After the subscription, the process reaches the receiving activity *Wait for unsubscribe event* that will terminate the process as soon

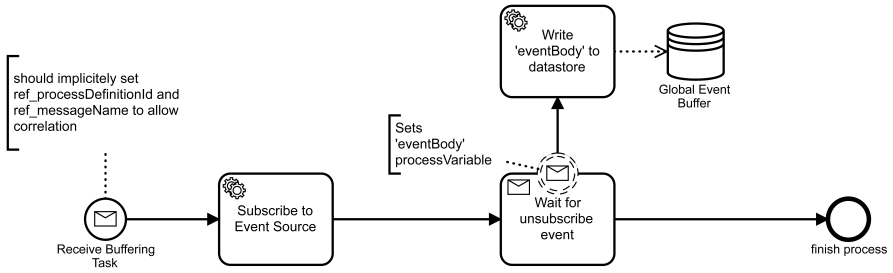


Figure 17: Auxiliary Buffering Process in the Camunda Modeler

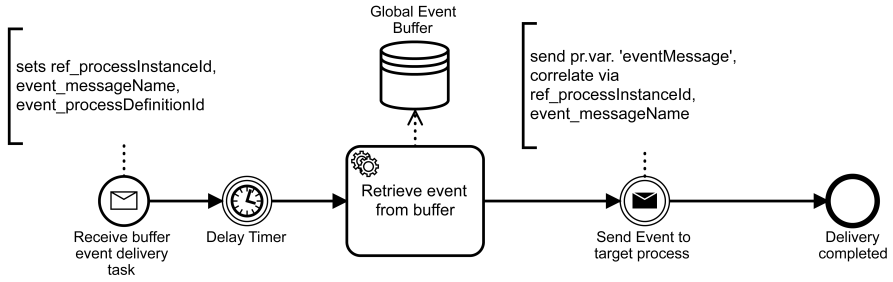


Figure 18: Auxiliary Event Delivery Process in Camunda Modeler

as the *Unsubscribe* event has been received. As long as this activity is active, events can be received through the attached non-interrupting boundary event. Incoming events have a field *eventBody*, which contains the event information and becomes available through a process variable with the same name. 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 (*Global Event Buffer*).

AUXILIARY EVENT DELIVERY PROCESS The delivery process (see Figure 18) 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 timer event *Delay Timer* has been inserted to make sure that the requesting process is already listening for an event, when the delivery process sends the message. The outgoing execution flow is treated 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 To initiate the subscription at the event source, the Auxiliary Event Buffering process has to be

started. For scenario *EOS3*, 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 *EOS4* and *EOS5*, the subscription and thus the instantiation of the Buffering Process must happen before the instantiation of the Target Process. As there is no such mechanism in the standard Camunda Process Engine, the Buffering Process must be started manually, 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 necessary communication for the termination of the processes and the unsubscribe-operation have not been implemented in this prototypical implementation, as they would not add particular value to the evaluation.

The code of the Camunda process application is provided as part of the deliverables of this thesis.

CONCLUSION The described implementation of the BPMN buffering concept presented in Figure 4.1 serves as a brief evaluation of the model. As illustrated by the help of the two examples, the resulting set of process applications allows to issue an event subscription flexibly according to the event occurrence scenarios. While a thorough analysis was not the target of this section, it still provides an introduction to the capabilities of the Camunda process engine, which will take an important role again in 6.

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 OVERVIEW & DISCUSSION

It was the goal of this chapter to evaluate the capabilities of current BPM solutions against the requirements presented in Section 3.3. The evaluation was performed at the example of the BPMN without allowing the use of the in-built extension mechanism. Furthermore, the developed BPMN solution for buffered event handling (Figure 4.1) was implemented prototypically in Camunda to evaluate the applicability of the concept to a business process engine.

Thanks to the flexibility of the BPMN standard, all of the requirements could be expressed in a model, but the solutions significantly vary in complexity. Table 1 summarizes the results, categorizing the findings into three classes: *(+)* means that the requirement is fulfilled natively and without the necessity for additional elements int the model; *(o)* signifies that there are acceptable ways to express the desired behavior, but additional elements are necessary for that purpose which increases model complexity; if the aspect is marked as *(-)*, the solution requires a drastic increase of complexity, which might not be acceptable. The categorization is based on the findings of the previous sections and a brief explanation is provided in the last column.

As revealed in the tabular overview, a means to express the required behavior could be found for each of the aspects. However, only two of them are considered to be natively supported by the standard while most of the concepts reveal certain shortcomings when it comes to modeling flexible event subscription using BPMN. In the following, the identified shortcomings are discussed one after the other. The discussion is arranged into three parts which are specifically marked as *S1* to *S3*. The shortcomings should be specifically addressed in addition to the requirements *R1* to *R3* when developing solutions to flexible event subscription.

Shortcoming S1: The necessary additional process elements increase model complexity and hence reduce comprehensibility.

The topic of model complexity has been investigated for example by Mendling et. al. [33, 34] and Vanderfeesten et. al. [42]. In their work on process modeling guidelines [34], Mendling et. al. state that *without a proper understanding of the business process [...] they are doomed to fail)*. One of the main reasons for decreased comprehensibility lies in the amount of elements and routing paths.

S1.1: Additional process elements are required within the additional process.

Most of the presented approaches require additional process elements or even entire processes to implement the behavior. To cope with the event occurrence scenarios *EOS2* and *EOS3*, the introduction of addi-

REQUIREMENT	EV.	EXPLANATION
R1: Subscription Fundamentals		
R1.1 and R1.2 (Un-) Subscription to/from the Event Source	o	requires additional model elements, e. g. throw/catch event, send/receive task, service task
R1.3 Availability of Information	o	possible by using Annotation or DataObject
R1.4 Variables in Event Queries	o	supported through custom implementation in service task
R2: Event Subscription Time		
R2.1 Explicitness	o	not explicitly stated by BPMN, but possible to derive
R2.2 Flexibility		
> EOS1	+	natively supported
> EOS2, EOS3	o	added complexity through parallel process flow
> EOS4, EOS5	-	requires auxiliary processes, significant increase of model elements
R2.3 Awareness of Subscription Dependencies	o	if subscription is modeled explicitly, a DataInput can be used
R3: Event Buffering		
R3.1 Buffering Principle	+	Temporary storage can be expressed through DataObjects
R3.2 Buffer Scope	o	DataObjects for instance-scope, DataStore otherwise. Interaction with datastore adds complexity.
R3.3 Buffer Policies	o	by textual description or the implementation-specifics of a service task; pure graphical representation will become extremely complex.

Table 1: Overview of the assessment of BPMN against the requirements

tional gateways and therefor routing paths is necessary for each event element that is supposed to use flexible event subscription. Due to that dependency on the number of buffered events in the process, the added complexity will increase linearly with that number.

S1.2: Two more participants must be added to implement the subscription before instantiation.

Another dimension becomes apparent in the BPMN model for buffered event handling that is presented at the end of Figure 4.1. While it allows the subscription to events even before process deployment, it requires the execution of an additional set of auxiliary processes for that purpose. Consequently, the increase of process elements is accompanied by an increase of participants. A simple model involving only one event will blow up into a complex process model with three pools, being less concentrated on the actual business case.

The increased complexity will not only make the resulting model more difficult to understand, but also complicate the design phase by requiring a basic understanding of the publish/subscribe model, the usage semantics of the auxiliary processes and the patterns revealed to control subscription time within a process instance.

Shortcoming S2: Event buffering and subscription are infrastructure tasks and therefor not suited to be implemented through a business process model.

Arguably, the buffering of data and the subscription and un-subscription from events are not business tasks and should hence not be entirely modeled in a business process.

S2.1: The buffering and subscription activities pollute the model with non-business-related content.

Though necessary to allow the successful execution of the process, the effort to enable flexible event subscription stands in no relation to the business value it represents. The models essentially get polluted with non-business-related process segments. It can be observed from the findings of this chapter, that it is not handy to model the required functionality in the BPMN. After all, two essential activities within the auxiliary tasks are still service tasks. Although the notation allows these kind of code-based tasks, their detailed behavior is still hidden within the underlying implementation.

S2.2: Process designers likely lack the skills to cope with event buffering and subscription.

Moreover, as noted in *S1*, the design of a process gets more complicated due to the increased number of elements. Another issue to consider in that matter is that the process designer who has to make

use of the auxiliary processes usually does not have a comprehensive knowledge in information technology (see [43], p. 16). An additional stakeholder will have to be included in the design process, resulting in more complex workflows.

Once a given process has been deployed in a process engine, it has to be monitored and maintained. Given that even the simplest process involves the execution of two additional instances for buffering, any maintenance and search for errors will be significantly more complicated. Just like in the design phase, the maintenance will require knowledge of event subscription and buffering, which is probably not available by default.

Shortcoming S3: Implementing event buffering through process models is likely to cause performance issues.

S3.1: The execution of additional buffering processes affects process engine reliability.

Because of the buffering and delivery jobs, two additional processes are potentially running in parallel to any given process instance. For each Event Element used in a process, the engine has to execute an instance of the buffering process and, eventually, an instance of the buffer delivery process. For example in a process using two event elements, one subscribing at deployment time and one subscribing after event instantiation, **the maximum number of process instances for a single execution increases from 1 to 5**. An exact formula to determine the number of instances is provided in 4.1.

It is not only the increase in the number of processes, but also the kind of work the the processes are executing: Each instance of the auxiliary buffering process constantly listens to an event source, potentially receiving events and thus creating load in sub-second scale. Instead of only listening to an event source when the event element is reached, the process engine has to listen, receive, process and write to the buffer all the time. Even if no instance of the original process is about to use the events. That increase of running process instances puts additional load on the process engine, which might prevent business critical processes from executing without delays. Taking into consideration that a process engine can only handle a limited number of process instances and operations, there is an increased possibility of complete engine failure.

S3.2: The execution of event buffering through a business process is limited by process engine performance.

Apart from the reduced reliability and the possible delay to the execution of the processes, there is a second performance-related perspective to the issue. Given the large amount and high frequency in

that events can occur in reality, optimal performance is required for an event-buffering module. Depending on the event query and frequency of event occurrence, the engine has to be ready to receive events at a very fast pace. When executing these operations from a BPMN model inside a process engine, the speed in that steps can be executed is only as fast as the process engine itself. While complex event processing platforms are optimized to handle many events in a short amount of time, a process engine is not built for that purpose and might fail to deliver the required performance. An aspect that cannot be influenced without customizing the process engine code.

5 FLEXIBLE EVENT SUBSCRIPTION IN BPMN

Given the requirements *R1-R3* (see Section 3.3, p. 21 ff.) and the shortcomings *S1-S3* (see Section 4.3, p. 35 ff.) identified in the previous chapters, this section presents an extension to the BPMN event handling model that allows the flexible use of event subscription mechanisms in business processes.

At first, 5.1 describes an extension to the Business Process Model and Notation (BPMN), which aims at providing a convenient solution for modeling events that must be obtained through the publish-subscribe interaction pattern. Afterwards, Section 5.2 clarifies the changes necessary to the event handling platform and the process engine to implement the implicit subscription management required by the BPMN extension. The results are summed up in Section 5.3, also providing a tabular overview (Table 3) of how the presented concept addresses the requirements and the identified shortcomings.

5.1 BPMN EXTENSION

The extensibility mechanism of the BPMN allows to extend standard process elements with additional attributes, while maintaining conformity with the BPMN core [37, p. 44]. The BPMN extension developed in the present work is revealed in this section. First, an overview is provided (Section 5.1.1), then a more detailed documentation of the functionality and usage is provided.

5.1.1 *Overview*

This section introduces a novel BPMN extension for incorporating event subscription semantics in business process models. The extension develops around the addition of subscription-related attributes to the BPMN *Message* type. Based on the additional information, subscription and un-subscription for message events is handled automatically by an especially adapted process engine. The added attributes are *EventQuery*, *SubscriptionTime* and *BufferPolicies*: *EventQuery* contains an arbitrary query string; *SubscriptionTime* defines when the subscription should happen; *BufferPolicies* is a complex type which influences the behavior of the related event buffers. Only *EventQuery* is a mandatory parameter, all others will fall back to default values if not provided. An overview of all possible fields, values and their defaults is provided in Table 2.

ATTRIBUTE NAME	VALUE OPTIONS (DEFAULT)	REQ.
EventQuery	any query string	y
SubscriptionTime	process-deployment, process-instantiation, manual, element reached	n
BufferPolicies	Complex Type (see below)	n
BUFFERPOLICIES		
LifespanPolicy	string in ISO time-span format OR 'infinite'	n
ConsumptionPolicy	reuse, bounded-reuse(n), consume	n
SizePolicy	positive integer, 1, 0 means infinite	n
OrderPolicy	FIFO, LIFO	n

Table 2: Available attributes of <subscriptionDefinition> in the BPMN extension for flexible event subscription

The presented extension is to be used in connection with intermediate catch events, boundary events and receive tasks through their direct or indirect association with the message element. Additionally, the *Event Subscription Task*, an extension of the *Task* is introduced. It adds a single additional attribute *messageId* to *Task* and can thereby be used to manually trigger the subscription for the associated message element as part of the execution flow of a process. By the use of the *SubscriptionTime* variable, the event subscription can be issued before the process flow reaches the event element and listens for the message. Any messages that arrive before the event element is active are temporarily stored and will be delivered to the catch event or receive task once it is active. These storages as referred to as *Event Buffers*. By default, they store the latest message they receive, though the behavior can be adjusted using the *BufferPolicies*.

5.1.2 *Documentation*

Through the BPMN extension, subscription-related information can be added to the Message element. While other authors have chosen to add subscription information directly to the catch event [**beyer2016unicorn**, 3], the Message element has been chosen for several reasons: It is a good match semantically, subscription meta information about an event message has a close relation to the Message element itself; By extending the Message element, a single extension applies to the intermediate event, the boundary event and the receive task at the same time (the extension is specifically designed for the use with exactly these three elements); If a single Message shall be used in a process

multiple times, the subscription information must only be provided once.

By specification, the *Message* element comprises an attribute *name*, the name of the message, and *itemRef*, the reference to a BPMN *ItemDefinition* specifying the Message structure. Additionally, it inherits all attributes from the BPMN *RootElement* (see [37] p. 93). In the following, the required additional attributes will be explained one after the other. A complete list is available in Table 2. The goal is to retain a standalone model that contains all information necessary to execute the subscription to the event source. All subscription information is incorporated in a model element *subscriptionDefinition*, which is added to the *extensionDefinitions* of the Message element. The formal XSD-Schema is available in the appendix, Listing 5.

Adding basic subscription information

For a subscribe operation, we consider the event query, the platform address and optionally authorization information of the CEP platform. This extension assumes that only one event engine is in use, so that access information can be configured in a central configuration store for the current process execution environment and not redundantly for each message. 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.

Given this first fundamental part of the BPMN extension, it is possible to execute the subscription, thus addressing requirement R1. However, the time of subscription cannot be influenced.

The time of event subscription modeled in BPMN

This section specifically addresses the requirement *R2*, aiming to provide a flexible event subscription time to be selected for each BPMN message when designing an event-driven process. Two different means 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 a task of the process design phase, to elaborate the correct time of subscription necessary for the 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 Section 5.1.3. As this means that the subscribe and unsubscribe operations themselves do not have to be explicitly modeled, shortcoming *S2* is addressed.

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 values: *process-deployment*, *process-instantiation*, *manual*, *element-reached*. The last option is the default option, intepreting the standard BPMN semantics. Note that a *subscriptionTime* set to *element-reached* will remain without effect if an explicit subscription task for the same event was executed before the event is reached.

In motivating Example 1.1 (Section 3.1), it is necessary to issue the subscription to the Eurotunnel event as early as possible to make sure that data is available and the process execution is not delayed. Using the BPMN extension, the use case can be implemented by defining the event query and the subscription time in the BPMN model as depicted in Listing 3.

Listing 3: XML representation of an extended BPMN Message element

```
<bpmn:message id="sample-message" name="SampleMessage">
  <bpmn:extensionElements>
    <flexsub:subscriptionDefinition>
      <flexsub:eventQuery>SELECT delay, occurrenceTime,
          delayReason FROM eurotunnel</flexsub:eventQuery>
      <flexsub:subscriptionTime>process-deployment</flexsub:
          subscriptionTime>
    </flexsub:subscriptionDefinition>
  </bpmn:extensionElements>
</bpmn:message>
```

THE EVENT SUBSCRIPTION TASK

As an alternative to specifying the subscription time using the extension field *SubscriptionTime*, an extension to the BPMN *Task* type is proposed. 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 5.1.2, 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.

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.

5.1.2.1 *Using Process Variables in Event Queries*

As shown in motivating example 2 (Section 3.1), it can be the case that the current values of process data must be dynamically used in an event query. Therefore, the name of the dataObject 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 may vary depending on the employed query language so that it does not interfere with any existing notation schemes. For the use with the Esper EPL, 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 EPL. They take a similar form, though using a $ sign[1].

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. Variables are replaced by their runtime values at the time of event subscription and an execution error is invoked if the required information is not available, causing the process instance to be stopped.

5.1.2.2 *Advanced Buffer Parameters*

This section of the documentation addresses requirement *R3*.

LIFESPAN OF BUFFERED EVENTS

The *LifespanPolicy* allows to specify after which timespan elements in the buffer should be deleted. Timespans are defined using the ISO8601 notation for time intervals[2]. The default value is *infinite*.

Motivating Example 1.1 is implemented by setting the *SubscriptionTime* to *process-deployment*, which means that there can be an infinite time difference between the action of subscribing to the event source and the reaching of the event element in one of the instances. In case

1 see *5.13.1. Joining SQL Query Results*, http://www.espertech.com/esper/release-5.3.0/esper-reference/html/epl_clauses.html, accessed 2017-08-13

2 see *Date and time format - ISO 8601*, https://www.iso.org/iso-8601-date-and-time-format.html, accessed 2017-08-13

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 *LifespanPolicy*, the process designer can express, that events should be deleted from the buffer after a certain period of time and thus avoid outdated information. The buffer is maintained automatically by the system. That of course comes at the price that the process has to remain in waiting state until a new event message arrives.

CONSUMPTION BEHAVIOR So far, the event buffers can be used isolated from each other. There is no interference between buffer instances and events are not removed from the buffer after retrieval. While for most use-cases this behavior is sufficient, more detailed control over the buffer can be desirable when a given message is accessed multiple times. More precisely, if an event element is activated multiple times or multiple elements reference the same *Message* element, then the consumption behavior must be clear. 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 possibility to *consume* from an event buffer, more complex scenarios can be designed. Let us consider the flight booking process shown in Figure 19: A request for flight booking offers is issued as a manual Event Subscription Task instantiating an Event Buffer. The referenced message with the identifier *FlightOffer* is modeled with a *manual SubscriptionTime* and the *ConsumptionPolicy* set to *consume*. The receipt of flight offers is modeled as an Intermediate Catch Event referencing the same message. After the receipt of the first offer, the offer is deleted from the buffer and the sequence flow proceeds to the activity *Evaluate Offer*. The buffer can receive flight offers during the evaluation of the first offer. When the evaluation is finished, either another offer is considered from the event buffer or decision-making is closed and the best offer is selected from all offers that where considered so far. Using the consuming buffer, the same buffer can be accessed multiple times in a single process without using the same event every time.

In that regard, the scope of a buffer is defined as follows: If the subscription time is *on-deployment*, a buffer ranges all instances of a given process. If it is set to *on-instantiation*, the buffer only ranges the one instance of the process. A buffer shared across multiple process definition is assumed when the subscription time is set to *manual*

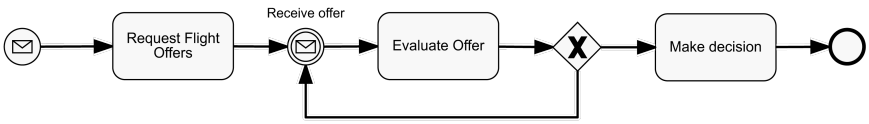


Figure 19: Simplified model of a flight Booking process using a consuming buffer

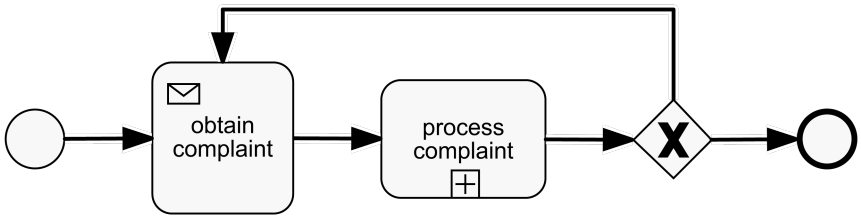


Figure 20: Shared consuming buffer in a simplified model of a complaints handling process

and the referenced message share the same message identifier and subscription information across all process definitions in scope.

A possible use case for a shared buffer is shown in Figure 20. Within an organization, multiple processes might exist to handle complaints. The referenced message elements share the same event query, manual subscription time and the *consume* policy across all these process definitions. Figure 20 shows how the events are retrieved from the shared buffer in *obtain complaint* and then processed. In the course of the gateway it is either decided to process another complaint using the given process or to end the process execution.

BUFFER SIZE AND ORDER POLICY Offering the option to consume from buffer instances that are reused within a process instance, across multiple instances of the same process or across multiple processes implies the need to additionally specify the number of elements that are stored in the buffer and the order of retrieval from the buffer.

The order policy can be either *FIFO* (first in first out) or *LIFO* (last in first out). Accordingly, the buffer either behaves like a queue (FIFO) or a stack (LIFO). The default value is *FIFO*. The size of the buffer is denoted as a positive integer value, whereas *0* means that the buffer is of infinite size, the default behavior. If the buffer contains its maximum number of elements, the element that would be retrieved from the buffer last (considering the order policy) is removed from the buffer.

5.1.3 *Execution Semantics*

After the previous section has described the functionality and usage of the BPMN extension, the following section textually defines the ad-

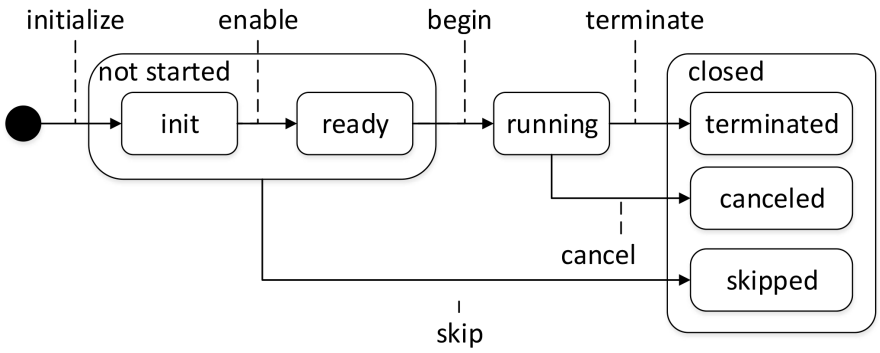


Figure 21: The activity instance life cycle

ditions necessary to the BPMN execution semantics to implement the subscription, the un-subscription and the interaction with the event buffer. According to the specification, the purpose of the execution semantics is *to describe a clear and precise understanding of the operation of the elements*. [37], p. 425 ff. While the execution semantics in BPMN must not be supported for a product to achieve *Modeling Conformance*, they must be implemented to achieve *Process Execution Conformance*.

LIFE CYCLE OF EVENT ELEMENTS The semantics are described in accordance with the activity instance life cycle [43, p. 84], which defines the states and transitions any activity instance traverses during process execution (Figure 21). When a process is instantiated, also activities are initialized and enter the init state. Once the execution activates the incoming flow of an activity, its state changes to *ready* and can enter the *running* state any time. If the activity execution completes without interruption, it traverses into *terminated*, if the execution is interrupted through a boundary event, it ends in the *canceled* state. If an activity never enters the *running* state because the process flow proceeded on a different branch, the activity is skipped.

Originally, this life cycle specifically relates to activity instances. To specify the execution semantics of the presented extension, for example the time of subscription and un-subscription, a similar model is required for describing the life cycle of event elements. For this purpose, the life cycle of an event is considered to consist of the same states and transitions as an activity. Event elements can be canceled when they appear after an event-based gateway and another event following the same gateway occurs earlier. If the trigger of the event fires, it enters *terminated*. Equivalent to activities, event elements can be skipped if the process execution choses a different path.

OPERATIONAL SEMANTICS FOR SUBSCRIPTION HANDLING The subscription behavior of the existing BPMN elements intermediate message catch event (IMCE), boundary message catch event (BMCE) and receive task (RT) can be defined using the *subscriptionDefinition*

extension inside the associated Message element. Furthermore, the subscription can be modeled as a flow element using the *Event Subscription Task*. Each of these elements already has execution semantics defined by the BPMN specification, which are extended with additional operations for flexible event handling. Due to the possibility to define the *SubscriptionTime* independently from the event element or receive task, adopting the semantics for these four elements is not sufficient to implement all possible event subscription times.

Because the execution semantics are reliant on the subscription time specified in the message, the enhanced semantics are not described per element, but for each possible subscription time. The required additions to the operational semantics of BPMN are as follows:

SUBSCRIPTIONTIME *element reached*: IMCE/BMCE/RT: The subscription to the event source is issued on the *begin* transition. The element remains in *running* state until an event is consumed. The un-subscription from the event source is performed either on the *terminate* or on the *cancel* transition.

IN EVERY OTHER CASE: IMCE/BMCE/RT: On the *begin* transition, events are requested from the Event Buffer associated to the element. The element remains in *running* state until the buffer delivers an event or the *cancel* transition takes place. When the *cancel* or the *terminate* transition is performed, the buffer is informed to stop delivering events.

SUBSCRIPTIONTIME *process-deployment*: The BPMN specification itself states that the *deployment of Business Processes [is] out of scope* [37, p. 22]. However, there must be a certain operational semantic implied by the start of the listening process to Message Start Events. Therefore: An Event Buffer for the Message element is initialized at the end of the process deployment process. At the same time when the listening process to a Message Start Event begins. The Event Buffer is destroyed when the process gets un-deployed, the same time a Message Start Event stops listening.

SUBSCRIPTIONTIME *process-instantiation*: The Event Buffer for the Message element is created when the Start Event that initializes the process instances creates a token on its outgoing Sequence Flows, that means after the start event has occurred. The Event Buffer is destroyed when the process instance terminates, more precisely, when all tokens of the process instance have reached an end node (a node without outgoing sequence flows).

SUBSCR. THROUGH EXPLICIT SUBSCRIPTION TASK: An Event buffer for the referenced Message is initialized when the Event Subscription Task performs the *begin* transition. The deletion of the buffer follows the same rules as in *Subscr. on process instantiation*

SUBSCRIPTIONTIME *manual*: The existence of an event buffer instance is assumed, hence no further operations are specified.

5.2 AUTOMATIC SUBSCRIPTION HANDLING

After defining the notation and functionality provided to the user through the BPMN extension for flexible event subscription, this chapter describes the changes necessary to the software infrastructure to implement the execution semantics. The concept requires that all subscription operations and event handling is executed by the system itself, solely based on the extended BPMN model. 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 so that a delayed delivery of events is supported. The following Section 5.2.2 specifies the changes necessary to the behavior of the process engine as the connecting element between the BPMN model and the event processing platform.

5.2.1 *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 (see Section 2.2).

AN API FOR BUFFERED EVENT PROCESSING The novel BPMN extension for flexible event subscription allows to issue a subscription to an event source well before the events ought to be delivered to the process instance. The introduction of an event buffer as a separate entity between the notification-recipients and the event query makes an adaptation of the API necessary. In the following, we propose an API for buffered event processing which provides the functionality necessary to implement the execution semantics specified in the BPMN extension. Note that the proposal is not restricted to a certain kind of technology or protocol to implement the API.

Firstly, the *subscribe* operation has to be available in 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 5.1.2.2 for a detailed specification of the semantics and possible values of the parameters. If *bufferPolicies* is not or only partly specified, the system should fall back to the denoted 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. It adds a notification recipient to a query that is already registered in the platform. 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.

A similar situation holds for the un-subscribe operation: Given flexible event subscription, the operation must comprise of two steps:

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 before process deployment. The query must be registered using *registerQuery* before the process instance, and hence the 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*.

ARCHITECTURAL OPTIONS Three options are considered to implement the explained functionality (Figure 22). (a) It can either be

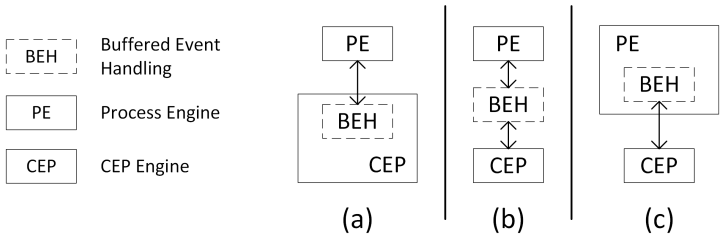


Figure 22: Architectural options to implement the event buffers

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

A reference implementation for an extended complex event processing platform is presented in Section 6.1 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.

5.2.2 *Extended Process Engine Behavior*

It is the task of the Business Process Engine to interpret and execute process models and connect to an event processing platform in event-driven setups. From the three relevant parts, two have already been defined, the BPMN extension and the buffered event processing module. Out of the box, a process engine like Camunda will ignore any proprietary BPMN extensions and the subscription to an event source must be especially implemented. An example for such an implementation is provided in Figure 4.1. 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 6.2 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 *Event Subscription Task* has to be processed. It contains a reference to a Message

entity within the same model element. 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 accordance with the defined execution semantics.

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. If the event query contains one or more variable values (see Section 5.1.2.1), all have to be replaced by their current values when *registerQuery* is performed. 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.

5.3 CONCLUSIONS

By the help of the presented BPMN extension it is possible to make use of subscription-based message reception while the time of subscription can be explicitly defined. The extension design aims to fulfill the requirements defined in Section 3.3 and to provide solutions to the shortcomings shown in Section 4.3. Table 3 contains an overview of how each aspect aspect is addressed by the extension.

ASPECT	EV.	EXPLANATION
R1: Subscription Fundamentals		
R1.1 and R1.2 (Un-) Subscription	+	operations are explicitly considered.
R1.3 Avail. of Information	+	in <subscriptionDefinition>
R1.4 Vars. in Queries	+	allowed as #varName
R2: Event Subscription Time		
R2.1 Explicitness	+	given for every choice of parameters
R2.2 Flexibility	+	by ext. attribute *subscriptionTime* and *EventSubscriptionTask*
R2.3 Awareness of Subscription Dependencies	o	recognized, but violation results in error
R3: Event Buffering		
R3.1 Buffering Principle	+	as *Event Buffers*
R3.2 Buffer Scope	+	instance, process or engine-wide
R3.3 Buffer Policies	+	controlled through ext. element *BufferPolicies*
Shortcomings		
S1 Incr. model compl.		
S1.1 Add. process elements	o	added extension elements but no additional nodes
S1.2 More participants	+	no add. participants in model
S2 Infrastructure tasks		
S2.1 Polluted models	+	minimal notation supported by default values
S2.2 Lack of IT-skills	o	subscr. & buffering still modeled, but less complex
S3 Performance Issues	+	buffering implemented in specialized module

Table 3: Overview of how the requirements and shortcomings are addressed by the BPMN extension for flexible event subscription

6 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 in BPMN to overcome the issues revealed in Chapter 3. The model extension is described in Section 5.1, and allows to create BPMN models that contain all information necessary to to issue event subscriptions for the employed event elements. To evaluate the results we enhance the business process engine Camunda and the event processing platform UNICORN following the procedures described in Section 5.2.2 and Section 5.2.1 respectively. Camunda is extended by providing a Process Engine Plugin (Section 6.2), Unicorn by adapting the source code (Section 6.1). Figure 23 shows an overview of the applied architecture with added and modified elements highlighted in gray.

6.1 EXTENDING THE EVENT PROCESSING PLATFORM UNICORN

UNICORN is an event engine developed for academic purposes at the Business Process Technology chair of the Hasso-Plattner-Institute, Potsdam [4, 20]. It has been developed in the course of the EU *GET project* [15], which investigates options to increase efficiency in logistics. Unicorn is based on the Esper [22], an open source component for complex event processing available in Java. Esper can be integrated into custom applications using its feature-rich Java API and will provide stream processing capabilities including an own Event Processing Language (EPL). Unicorn integrates Esper and adds a web-based user-interface, a REST API, a persistent data-store for events and a number of features specifically related to BPM. The latest version of Unicorn uses Esper 5.3.0, therefor all future considerations are made based on the documentation of that version.

Section 5.2.1 considers three architectural options to provide buffered event handling capabilities. 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 (see shortcoming *S3* in Section 4.3). 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-

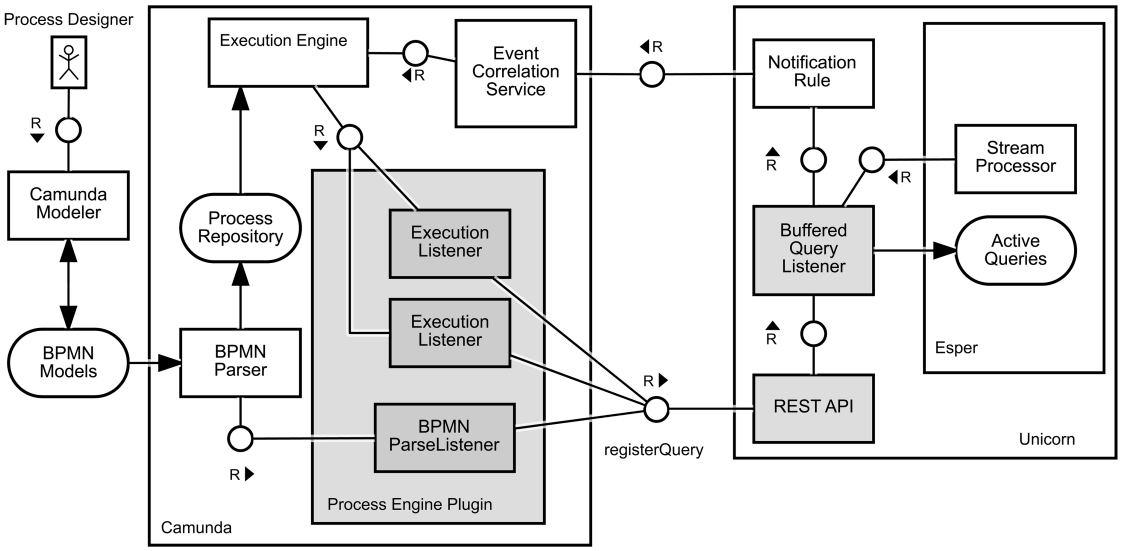


Figure 23: Architecture for flexible event subscription in Camunda and Unicorn

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.

6.1.1 *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.

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 BufferMan-

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

ager 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 5.1.2.2 have been implemented. The following section explains how the REST API of Unicorn is extended to make use of the buffering module.

6.1.2 *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 5.2.1, 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 2). 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.

Using instances of *NotificationRule*, Unicorn sends notifications to the recipients (see Figure 23). 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 4.

Listing 4: Example of a JSON notification sent by UNICORN

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

```
  }
}
```

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.

6.2 EVENT SUBSCRIPTION HANDLING IN CAMUNDA

Through the BPMN extension for flexible event subscription, the information necessary to register event queries in a CEP platform can be made available in process models. Section 5.2.2 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 and used to exemplify the modification process. Further information about Camunda is provided in Section 2.1.1.

Figure 23 depicts a simplified architecure of Camunda, highlighting modifications in gray. Our workflow for flexible event subscription mainly involves the core components, execution engine, model repository, correlation service and the Camunda modeler.

6.2.1 *Process Engine Plugin and ExecutionListeners*

Being a community-driven open-source project, the Camunda project offers numerous options for customizing and extending its behavior. A core concept to trigger the execution of custom code during a process execution are *ExecutionListeners*. They can be incorporated in BPMN models using the properties window of the Camunda Modeler as demonstrated in Chapter 4. In that case they have a textual representation in the model file, a specification can be found in the product documentation [3].

However, one of the main goals of this thesis is to provide an alternative solution to explicitly modeling the subscription, unless an explicit subscription task shall be used). Thus the additional necessity attach execution listeners to BPMN elements shall be avoided. Instead, subscriptions shall be managed automatically by the process engine solely based on the subscription definition provided in the extended message element. To achieve this kind of behavior, Camunda offers the concept of a Process Engine Plugin (PEP) to intercept significant engine operations and introduce custom code. By this means, execution listeners can be added programatically. The plugin is a separate software module that implements the interface *ProcessEngine-*

3 *Camunda BPMN Extension Elements*, https://docs.camunda.org/manual/7.7/reference/bpmn20/custom-extensions/extension-elements/#executionlistener

Plugin [4]. It is activated by adding a *plugin* entry in the process engine configuration.

[32] and [38] have chosen to directly adapt the source code of camunda. More precisely, they propose to modify the *Behavior* class of the Camunda core 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. That holds especially when it is only necessary to execute additional operations and not modify or delete existing code. Moreover, the PEP facilitates the re-usability across environments and different versions of Camunda.

6.2.2 *Managing event Subscriptions at Runtime*

The Process Engine Plugin enables us to execute custom JAVA code at predefined points during engine execution. An entry point to the modification of the execution behavior is provided through the implementation of the *ProcessEnginePlugin* interface. It allows to intercept *preInit*, *postInit* and *postProcessEngineBuild*, that means at three different points during the engine bootstrapping. We chose to provide a custom implementation for the *preInit* method, which has an object of *ProcessEngineConfigurationImpl* as parameter. With the engine configuration available, a plethora of custom handlers, validators and listeners can be registered[5].

We implement a *BPMNParseListener* that is to be executed after a BPMN element finished parsing (*customPostBPMNParseListeners*). The BPMNParseListener interface allows us to react to the parsing of single elements based on their type by making a separate method for every BPMN element available. A helpful example for using that listener is also provided on GitHub[6]. The BPMN extension affects the intermediate and boundary message catch event, the receive task, the explicit subscription task and can also be used to specify subscription information for a message start event. It was attempted to utilize the separate methods of every element, but it is not possible to access the subscription information in the associated message elements from that point. Apart from the provided method arguments, the engine's *RepositoryService* can normally be used to view model information, but only after the deployment was finished. To overcome this limita-

4 *Process Engine Plugins*, https://docs.camunda.org/manual/7.7/user-guide/process-engine/process-engine-plugins/

5 Some arbitrary examples are pre-/post-deployers, event-handlers, post-variable-serializers, command-interceptors. Full list available from: https://docs.camunda.org/javadoc/camunda-bpm-platform/7.7/org/camunda/bpm/engine/impl/cfg/ProcessEngineConfigurationImpl.html

6 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

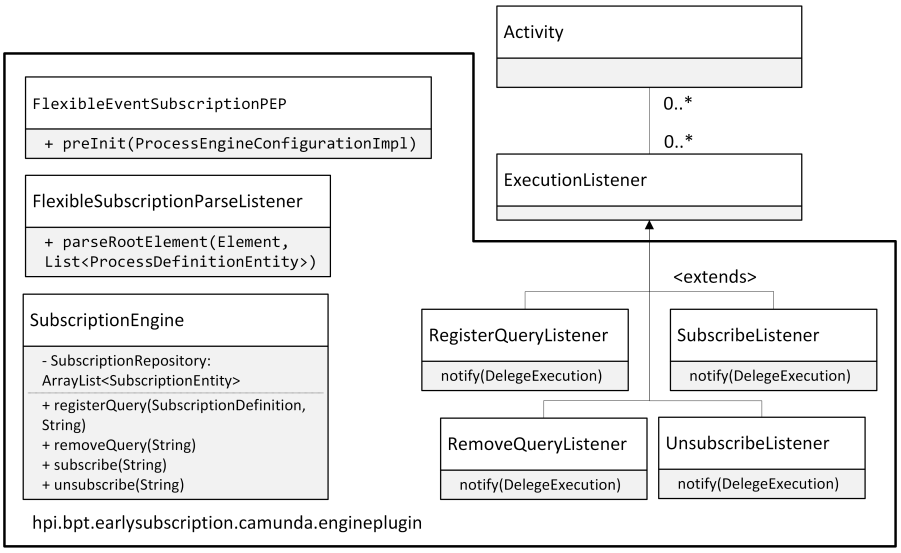


Figure 24: Excerpt of the UML class diagram of the Camunda Process Engine Plugin

tion, we implement the method *parseRootElement*, to which a representation of the root element *bpmn:definitions* is passed, including all message and extension elements. Before the content of this method is described, two more aspects need to be introduced, the *SubscriptionEngine* and the *ExecutionListeners*. An excerpt of the related UML class diagramm is depicted in Figure 24.

SUBSCRIPTIONENGINE A new class *SubscriptionEngine* has been introduced, that encapsulates most of the functionality needed to communicate with the API of the event engine. It offers method representations of all four API-calls, *registerQuery*, *subscribe*, *unsubscribe* and *removeQuery*, translating them into the according HTTP calls which are then executed. The second, more important functionality is the *SubscriptionRepository*, a list of all available query and subscription identifiers and a mapping to the related process definitions or instances. As described in Section 6.1.2, active queries and subscriptions can only be deleted using their unique identifiers. When issuing a subscription or registering a query, these identifiers are stored in the repository until the removal call is executed. Based on the repository information, the SubscriptionEngine can determine the subscription identifiers for provided event element information and execute the remove and unsubscribe operations on the event engine.

EXECUTIONLISTENERS The concept involves four execution listeners, one for each API-call, which are attached to activities through the BPMN parse listener. Execution listeners can be triggered by the camunda-internal end- or start-event of an activity and can therefor be used to implement the event subscription handling. Their imple-

mentation is concise: each of them is a separate class implementing the *ExecutionListener* interface. They only have one method *notify*, which is called by the engine when the end- or start-event fires. Within the method, each listener class issues an API-call using the functionality exposed by *SubscriptionManager*.

IMPLEMENTATION OF PARSEROOTELEMENT After introducing the *SubscriptionEngine* and the *ExecutionListeners*, this paragraph describes how the listeners are attached to the activities in the course of the execution of *parseRootElement* in our custom *BPMNParseListener*. By the help of the model representation passed ot the method as an argument of type *Element*, all relevant xml elements can be extracted based on the element names. This results in a list of all intermediate and boundary message catch events and receive-tasks and the messages that they reference. The subscription definition is available as a child element of each message. For each element in the list, the execution now proceeds as outlined in Section 5.2.2. Depending on the specified subscription time, ExecutionListeners are added at the beginning or end of activity executions or process executions. If the specified subscription time is *Process Deployment*, the provided query is registered immediately using the SubscriptionManager.

This can be exemplified by an intermediate message catch event with subscription time *on process instantiation*:

- An instance of *RegisterQueryListener* is added to the start event of the process. The *subscriptiondefinition* is provided to that listener.
- The *SubscribeListener* is attached to the start event of the activity representation of the intermediate message event.
- To the same activity, objects of *UnsubscribeListener* and *RemoveQueryListener* are registered to be triggered by the end event.

It is worth noting that the available code-base can easily be used to implement the event subscription for message start events, so that processes can be automatically instantiated by events delivered from the event engine. Considering that process definitions in Camunda are rarely removed completely, the removal of queries on process undeployment was not implemented. The lack of an obvious intercept point to inject code at that time makes an implementation cumbersome.

Altogether, the enhanced CEP platform and business process engine enable the automatic handling of information provided through the BPMN extension for flexible event subscription. The event engine exposes functionality for buffered event handling which is accessed through execution listeners during the process execution in Camunda. The Camunda extension can be flexibly re-used across environments

thanks to its implementation in the form of a Process Engine Plugin. Given these extended features, process designers can conveniently incorporate subscription information for external message events in their executable BPMN models.

7 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 [31]. The discipline is investigated from different perspectives. In event-driven business process monitoring and business activity monitoring, the process engine takes the role of an event producer that publishes information about its process executions to an event engine [5, 10, 21]. This information can be used to detect process violations, evaluate execution performance and derive key performance indicators during process execution [24, 25], whereas the field of process mining intends to derive process knowledge from historic information [41]. In their comprehensive survey on EdBPM [29], Krumeich et. al. distinguish a second application possibility, Event-driven Business Process Control (EdBPC), where process executions consume events to control the execution flow. This particular area of research is related to in this thesis. An example application is shown by Estruch et. al. [17] who incorporate events in a manufacturing process in order to increase process performance. [11] and [3] apply the idea to logistics. Pufahl [38] presents a more generic application of business process control in her work on the re-evaluation of decisions based on events.

The requirement for an appropriate integration of the CEP component into the control of the business processes is highlighted in [13]. An essential aspect when interacting with an event engine is to consider the semantics according to the publish-subscribe paradigm [31]. While the need for event subscription is recognized, for example by Decker and Mendling [14] in their investigation of process instantiation mechanisms and also by [38] and [1], the need for a flexible time of subscription is not considered in these works. Instead, they assume according to their interpretation of the the BPMN specification, that the subscription is only issued once the event element is enabled. The inferred possible loss of process efficiency, which also provides the basis of argumentation for the present work, has first been pointed out by Mandal, Weidlich and Weske in their work on early event subscription [32]. They argue that a late time of event subscription might lead to missed events and hence the delay or complete halt of the process execution. To address the issue, they introduce an extension to the BPMN service task, which can be used to explicitly model the subscribe-operation in the process flow. They derive the need for a temporary store of events and introduce formal execution semantics

though not further elaborating on how buffer instances are correlated to event elements, process instances and processes.

Following a congruent initial line of argumentation, this thesis first investigates the capabilities of the BPMN to express flexible event subscription semantics before assuming that the necessary modeling goals cannot be achieved, leading to a more comprehensive list of requirements and the formulation of shortcomings of current solutions. The BPMN extension presented in Chapter 5 primarily builds upon adding subscription-related information to the Message-element to allow an implicit management of event subscriptions as opposed the explicit option that is provided through the event subscription task. To enable the implementation of automatic subscription handling, guidelines on how to adapt the process engine and the event handling module are provided as well as a reference implementation at the example of Camunda and the Esper-based event engine Unicorn. While [32] does not further elaborate on the options to correlate an event buffer to an event element, process instance or process model, the present work defines the scope of a buffer depending on the chosen subscription semantics. It is revealed, how the Camunda process engine can be extended to execute custom operations on process deployment, process instantiation and the enabling of the event element by using a portable and conveniently customizable process engine plugin.

To enable flexible event subscription in BPMN models, a BPMN extension in accordance with the built-in extension mechanism is presented. Krumeich et. al. [29] recognize this to be the means of choice in many related contributions. In [8] the authors Braun and Esswein present an overview and classification of extensions proposals made in the research environment, revealing the state-of-the-art of extension techniques and identifying the need for an integrated methodological support. This leads to a detailed analysis of the extension mechanism in [7], considering that there is an explicit and an implicit extension mechanism in BPMN. Other notable approaches to cope with event processing in business processes aim at including streams and stream processing artifacts right into the business process model [2, 6]. While offering a more generic and thus more powerful option to represent event processing semantics, the concepts contradict with the goal of this work to simplify event subscription handling given an existing event processing engine. The same argumentation holds for Juric and Matjaz [26], who developed WSDL and BPEL extensions for Event Driven Architectures.

With a closer relation to the field of Complex Event Processing, the present work discusses the need for a temporary event storage. Seen from another perspective, the usage of an event buffer implies that historic events are consumed. However, the maximum age of the events depends on the described Lifespan Policy. Event persistence within CEP platforms can be seen as a means to implement event

buffering. The topic of event persistence is considered by Roth et. al. [39] who present an architecture for event data warehousing, whereas Buchmann et. al. [9] describe an attempt to integrate information from logistics and manufacturing. Both apply a persistent event store to allow historic analysis. More generally, [30] presents a concept to retrieve historic information in a publish-subscribe work flow. While these approaches can certainly be used to implement some kind of event buffer, this would still not diminish the need to consider subscription-related information in process models and, arguably, a persistent data-store poses performance limitations to implementing buffers in complex event processing. With regards to CEP, 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 or even out short term load peaks [12].

8 CONCLUSION

As organizations have adopted comprehensive business process management solutions, they are constantly seeking to improve process capabilities, quality and performance. Integrating Complex Event Processing into their business work flows is a popular means towards these goals. As event-driven architectures mostly operate according to the publish-subscribe paradigm, the support of this interaction pattern is increasingly important when executing business processes. The industry-standard Business Process Model and Notation (BPMN) offers comprehensive support for events but does not address subscribe operations specifically. The common understanding is that the subscription to an event only takes place when the process element gets enabled, resulting in earlier-occurring events not being available for use in the process.

Motivated by the possible process execution issues implied by the BPMN specification, the present work has investigated the topic of event subscription in BPMN. As a first step, the motivating scenarios and related work were translated into three requirements, *R1-R3*, that a business process meta model must fulfill to address the previously stated problems. The core of these requirements is the necessity of the possibility to influence the event subscription time through model elements. The necessary options and requirements are stated taking into consideration the introduced event occurrence scenarios. Given that event subscription and consumption are not performed at the same time, the need for a temporary storage of events was identified.

It follows Chapter 4, which assesses for each requirement to what extent it can be expressed in the BPMN. For most of the aspects, it was possible to find acceptable solutions, though the complexity of the models did increase in every case. This became especially apparent when analyzing the event occurrence scenarios before instantiation and before deployment. A complex business process model involving two additional auxiliary processes was presented to enable event subscription and event buffering in the corresponding cases. This model was also implemented in the Camunda business process engine to evaluate its applicability. The results of that analysis were summarized in three shortcomings, *S1-S3*, criticizing the increase in model complexity, the misuse of BPMN for infrastructure tasks and the possible performance limitations when implementing flexible subscription and event buffering in BPMN.

Together, the requirements and the shortcomings form a general requirements framework to build solutions addressing the lack of subscription semantics in BPMN.

Building on that framework, Chapter 5 reveals a novel BPMN extension that allows an explicit and flexible use of event subscription in business process. The extension builds upon the addition of subscription-related information to the BPMN Message element. It requires that all information necessary to issue a subscription can be obtained from the model. The extension aims to offer a convenient use of subscription operations by requiring automated subscription handling in an adapted process engine. The additional requirements to process engine and event handling are outlined as part of the extension.

Given the model extension and the derived requirements to process engine and event processing API, Chapter 6 describes a reference implementation at the example of Camunda and the Esper-based event processing platform Unicorn. Camunda has been extended with subscription handling functionality, issuing event subscription and unsubscription in accordance with the model extension. A process engine plugin was implemented for that purpose. It is built around a central ParseListener that triggers the execution of custom code when a business process model is deployed. Furthermore, a number of ExecutionListeners are assigned to process elements. They control subscription and un-subscription during the execution of the process. Thanks to the plugin-mechanism, the resulting artifact is easily portable and can be used across different versions of the process engine. To add the necessary functionality to the event engine, its implementation has been extended with an event buffering module and additional API methods. As a whole, the enhanced business process management system is able to process and execute process models using the newly introduced BPMN extension for flexible event subscription. The system reads the subscription-related meta information from the model and manages event subscription and un-subscription automatically.

DISCUSSION AND FUTURE WORK

This thesis provides an end-to-end perspective on the topic of event subscription in business processes while focusing on the event subscription time and its implications. The BPMN extension was designed based on a theoretical analysis of the specification and related work, but without an empirical basis to evaluate usability and applicability in real-world use cases. This opens up a significant possibility for future research, as the design should be evaluated through user studies and the analyses of additional use cases. It will also be worth discussing whether a visual representation of the extended model

properties is helpful to improve usability. One of the limitations of the presented concept is the assumption of using a single CEP engine only, while in more complex situations event subscription might have to be issued to multiple systems. Moreover, the added requirements to CEP and process engine are formulated in a very generic manner, not considering, for instance, implied performance challenges in detail. This influences the reference implementation, which exemplifies how the necessary changes can be made to the engines but does not aim to provide the maturity to be used in a production environment.

A shortcoming identified in BPMN was that even if event subscription and buffering can be expressed in a model, process designers usually do not not have the required IT competencies to handle the topic. The extension addresses this subject by automating the event subscription operations and only requiring the definition of the subscription time and buffer policies in the model. However, one might argue that it is not the subscription time itself that is of interest in the design phase, but the implicit maximum age of the events that can be used in the process. Following this line of thought, there is room for further simplification of the modeling concept. In this regard, related work has also recognized that process designers usually lack the knowledge for formulating the necessary event queries, though this issue was not addressed in the present work.

APPENDIX

A

Listing 5: XSD schema of the BPMN extension for flexible event subscription

```
<xsd:schema xmlns:flexsub="http://www.some.url/" xmlns:xsd="http://
    www.w3.org/2001/XMLSchema" targetNamespace="http://www.some.url
    /">
<xsd:element name="eventSubscriptionTask" type="
    tEventSubscriptionTask" />
<xsd:complexType name="tEventSubscriptionTask">
<xsd:complexContent>
<xsd:extension base="tTask">
<xsd:attribute name="messageId" type="xsd:QName" use="required"/>
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>

<xsd:element name="subscriptionDefinition">
<xsd:complexType>
<xsd:sequence>
<xsd:element type="xsd:string" name="eventQuery" minOccurs="1"
    maxOccurs="1" />
<xsd:element type="tSubscriptionTime" name="subscriptionTime"
    minOccurs="0" maxOccurs="1" default="Element Execution"/>
<xsd:element name="bufferPolicies" minOccurs="0" maxOccurs="1">
<xsd:complexType>
<xsd:sequence>
<xsd:element type="xsd:string" name="lifespanPolicy" minOccurs="0"
    maxOccurs="1" default="infinite"/>
<xsd:element type="xsd:string" name="consumptionPolicy" minOccurs="0"
     maxOccurs="1" default="Reuse"/>
<xsd:element type="xsd:integer" name="sizePolicy" minOccurs="0"
    maxOccurs="1" default="0"/>
<xsd:element type="tOrderPolicy" name="orderPolicy" minOccurs="0"
    maxOccurs="1" default="FIFO"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:simpleType name="tSubscriptionTime">
<xsd:restriction base="xsd:string">
<xsd:enumeration value="Process Deployment"/>
<xsd:enumeration value="Process Instantiation"/>
<xsd:enumeration value="Manual"/>
<xsd:enumeration value="Element Execution"/>
</xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="tOrderPolicy">
<xsd:restriction base="xsd:string">
<xsd:enumeration value="LIFO"/>
<xsd:enumeration value="FIFO"/>
</xsd:restriction>
</xsd:simpleType>
</xsd:schema>
```

BIBLIOGRAPHY

[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] Anne Baumgrass, Claudio Di Ciccio, Remco M Dijkman, Marcin Hewelt, Jan Mendling, Andreas Meyer, Shaya Pourmirza, Mathias Weske, and Tsun Yin Wong. "GET Controller and UNICORN: Event-driven Process Execution and Monitoring in Logistics." In: *BPM (Demos)*. 2015, pp. 75–79.

[5] Anne Baumgrass, Nico Herzberg, Andreas Meyer, and Mathias Weske. "BPMN extension for business process monitoring." In: *EMISA*. 2014, pp. 85–98.

[6] 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.

[7] Richard Braun. "Behind the scenes of the bpmn extension mechanism principles, problems and options for improvement." In: *Model-Driven Engineering and Software Development (MODELSWARD), 2015 3rd International Conference on*. IEEE. 2015, pp. 1–8.

[8] Richard Braun and Werner Esswein. "Classification of domain-specific bpmn extensions." In: *IFIP Working Conference on The Practice of Enterprise Modeling*. Springer. 2014, pp. 42–57.

[9] 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.

[10] Susanne Bülow, Michael Backmann, Nico Herzberg, Thomas Hille, Andreas Meyer, Benjamin Ulm, Tsun Yin Wong, and Mathias Weske. "Monitoring of business processes with complex event processing." In: *International Conference on Business Process Management*. Springer. 2013, pp. 277–290.

[11] 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.

[12] Sharma Chakravarthy and Jiang Qingchun. *Stream Data Processing: A Quality of Service Perspective*. Springer, 2009.

[13] K. Mani Chandy and W. Roy Schulte. *Event Processing - Designing IT Systems for Agile Companies.* McGraw-Hill, 2010.

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

[15] Remco Dijkman and European Union. *GET Service*. accessed 2017-08-07. 2017. URL: http://http://getservice-project.eu/.

[16] 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.

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

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

[19] Camunda Services GmbH. *Camunda BPM platform - overview*. accessed 2017-07-28. 2017. URL: https://camunda.com/bpm/features/.

[20] Business Process Technology at Hasso-Plattner-Institute. *UNICORN*. accessed 2017-08-07. 2017. URL: https://bpt.hpi.uni-potsdam.de/UNICORN/WebHome.

[21] Nico Herzberg, Andreas Meyer, Oleh Khovalko, and Mathias Weske. "Improving business process intelligence with object state transition events." In: *International Conference on Conceptual Modeling*. Springer. 2013, pp. 146–160.

[22] EsperTech Inc. *EsperTech - Esper*. accessed 2017-07-04. 2017. URL: http://www.espertech.com/esper/.

[23] *Information technology — Object Management Group Business Process Model and Notation*. Standard. International Organization for Standardization, July 2013.

[24] Christian Janiesch, Martin Matzner, and Oliver Müller. "A blueprint for event-driven business activity management." In: *International Conference on Business Process Management*. Springer. 2011, pp. 17–28.

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

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

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

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

[29] 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: *Business Proc. Manag. Journal* 20 (2014), pp. 615–633.

[30] 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.

[31] David Luckham. "The power of events: An introduction to complex event processing in distributed enterprise systems." In: *International Workshop on Rules and Rule Markup Languages for the Semantic Web*. Springer. 2008, pp. 3–3.

[32] 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).

[33] Jan Mendling. *Metrics for process models: empirical foundations of verification, error prediction, and guidelines for correctness*. Vol. 6. Springer Science & Business Media, 2008.

[34] Jan Mendling, Hajo A Reijers, and Wil MP van der Aalst. "Seven process modeling guidelines (7PMG)." In: *Information and Software Technology* 52.2 (2010), pp. 127–136.

[35] 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.

[36] Inc. Object Management Group. *OMG | Object Management Group*. accessed 2017-08-04. 2017. URL: http://www.omg.org/.

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

[38] 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.

[39] 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.

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

[41] Ashutosh Tiwari, Chris J Turner, and Basim Majeed. "A review of business process mining: state-of-the-art and future trends." In: *Business Process Management Journal* 14.1 (2008), pp. 5–22.

[42] Irene Vanderfeesten, Jorge Cardoso, Jan Mendling, Hajo A Reijers, and Wil MP van der Aalst. "Quality metrics for business process models." In: *BPM and Workflow handbook* 144 (2007), pp. 179–190.

[43] 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

Dennis Wolf