A Framework for Integrating Real-World Events and Business Processes in an IoT Environment

Sankalita Mandal(✉), Marcin Hewelt, and Mathias Weske

Business Process Technology Group, Hasso Plattner Institute,
University of Potsdam, Potsdam, Germany
{sankalita.mandal,marcin.hewelt,mathias.weske}@hpi.de

Abstract. Business process management is essential for companies to document, execute, monitor, and optimize their business processes. These processes are often influenced by external events occurring in the process context, especially when considering Internet of Things (IoT) scenarios. Modeling constructs for different types of events are part of the Business Process Model and Notation (BPMN) standard. However, when the integration of external events needs to be supported by process-oriented information systems, the gap between conceptual process model and its implementation needs to be bridged. We elicited the requirements for this integration using an use case from the IoT domain. Based on them, we propose a framework that outsources the management of events to an event processing platform that the process engine subscribes to. The BPMN process model is extended with annotations to specify the type of expected events. Further, we implement a system that realizes the proposed integration..

Keywords: Process execution · Event processing · BPMN

1 Introduction

Business processes are omnipresent in companies. In today's digital age, huge amount of data is being produced every moment and processes try to take advantage of those streams of dynamic event data. In an Internet of Things (IoT) scenario, sources like smart devices and sensors generate tons of events, which can be filtered, combined, and aggregated to trigger and drive business processes. Proper aggregation and analysis of the events makes the processes more flexible, robust, and efficient. BPMN 2.0 (Business Process Model and Notation) offers a rich variety of constructs to model different types of events, e.g. start, intermediate, and end events that can be further differentiated into throwing and catching events

To support a business process with IT, e.g. by enacting it in a process engine, the gap between the conceptual level of the model and the detailed, technical level required for running the process needs to be bridged. For catching events, which represent that a process instance waits and reacts to some environmental occurring, this means to specify how this event can be detected by the process engine, how its information is extracted and mapped to process variables, and how it is correlated to the correct process instance. Throwing events are produced by the process instance, hence they do not need to be detected and correlated. However, process variables have to be packaged into the produced event. The gap between process model and its implementation hinders the fast deployment and subsequent optimization of business processes in a company.

Our contribution aims at bridging this gap by (a) providing a conceptual framework for the integration of events and processes and (b) implementing the proposed framework in the process engine Chimera [8]. We gather the requirements to come up with the framework with reference to a use case from IoT scenario. Namely, we address the following major aspects in our framework:

- Separation of concerns between process behavior and event processing
- Aggregation of events and representation of event hierarchies
- Execution of event integration into business processes

Complex event processing and business process management are individually well explored fields. But the integration of these two worlds is still in its early stage. Our framework establishes the required steps for enabling the communication between events and processes. The prototypical implementation offers an end-to-end solution that encompasses (a) the modeling of processes, data, and event types, (b) the deployment of process models into the process engine and of event types and event annotation into the event platform, and (c) the execution of process models integrated with external events.

We suggest to outsource detection and correlation of catching events to a dedicated event processing platform, in our case Unicorn [19]. This supports separation of concerns and hides the complexity of dealing with external events, especially event adapters and aggregation, from the process engine. High-level events [6] aggregated from primitive events encapsulate complexity and serve as interface for the process model. When modeling event nodes, modelers can refer to the expected high-level event, e.g. a positive market trend, instead of having to deal with hundreds of individual stock tick events, because the event platform takes care of aggregating those into a higher-level event. This eases process modeling and keeps the annotations required for process execution simple. Also, the separation of process control and event processing logic improves maintainability, in case there is a change in event aggregation rules.

The paper is structured as following. Section 2 illustrates the fundamentals of business process management and complex event processing as the ground of our framework. Section 3 elicits the requirements using a motivational use case. The related works in both the fields relevant to our requirements are described in Sect. 4. Section 5 presents the conceptual framework for using events in processes and describes how we address the elicited requirements in the proposed framework. The prototypical implementation is detailed in Sect. 6. Finally, Sect. 7 concludes the paper and mentions the future research possibilities.

2 Foundations

In this section, we present the fundamental knowledge required to build our framework. There are two main domains addressed in the work, namely, business process management and complex event processing. The basic concepts from those two fields are discussed below.

2.1 Business Process Management

A business process is a sequence of activities performed in an organizational context. These activities collectively achieve a business goal [21]. The activities and their orchestration are represented with business process models. BPMN 2.0 is the de facto standard for modeling business processes. A business process model can be considered as a blueprint for multiple process instances. Similarly, an activity model can be instantiated for a set of activity instances.

An activity instance goes through several state transitions as shown in Fig. 1. Each activity in the process is initialized and is in state *init* as soon as a process instance is started. When the incoming flow of an activity is triggered, the instance is in state *ready*. The state changes to *running* once the activity starts execution. Finally, the activity ends and goes to *terminated* state. If the activity instance is *not started* but before it starts the process instance follows a different path, then it directly goes to *skipped* state. The occurrence of an attached boundary event can change the state of a running activity instance to *canceled*.

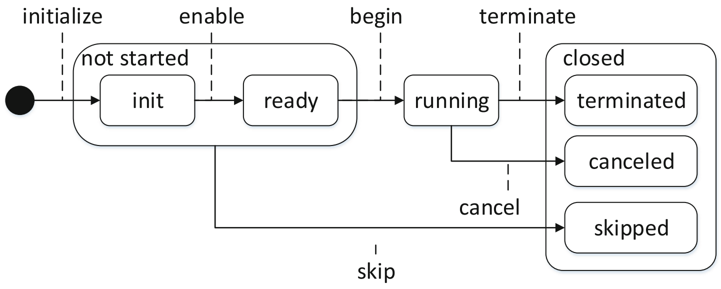


Fig. 1. Activity instance life cycle

The process flow can be enriched by information about the occurrences in the environment represented as events. BPMN describes different usage of events based on the position of the event in the process, namely, *start*, *intermediate* or *end* events. Start events are used to trigger a process instance. Intermediate events are produced or consumed by the process to use the information for further execution. If the event is received in the process, then it is called a *catching event*. The event produced by a process is named as *throwing event* in contrast. We will focus on the catching events as they are generated in the environment and used in the process.

Intermediate events such as boundary events or event-based gateways can be used to determine the process flow. Boundary events are associated with an activity and they can be interrupting or non-interrupting. If the event occurs after the activity is started and before it finishes, then an exceptional path is triggered. In case of interrupting boundary event, the ongoing activity is canceled. An event-based gateway is a decision gateway that depends on event occurring instead of data. The first occurrence among the events after the gateway causes that branch to be executed further.

2.2 Complex Event Processing

Using BPMN, one can model processes with catching events that are needed in the course of process execution. But BPMN does not talk about the source of these events, the information carried by the events or how to receive these events. On the other hand, such concepts are well explored in the event processing field [15]. Events are the environmental occurrences that can be relevant for a process. Event objects represent these occurrences in a computing system [12]. While an event can be a road accident or a sudden temperature fall, the representative event objects can be a traffic update from traffic API or a weather update from a sensor, respectively. The terms *event* and *event object* are often used interchangeably. Atomic events do not take time to occur, e.g., events generated by sensors. Atomic events can be aggregated to generate higher-level or complex events. A set of temporally totally ordered associated events form an event stream. Operations can be performed on single events (Simple Event Processing), multiple events in a single event stream (Event Stream Processing) or multiple events in multiple event streams (Complex Event Processing).

It is important to distinguish among three different kinds of events, which are denoted by the same term, although they are largely different. First, the term event can refer to the modeling construct of BPMN that is used to model catching or throwing events. We will use the term *BPMN event* for disambiguation. Second, it can refer to state transitions in lifecycles, e.g. the event `begin` that puts an activity instance into state *running* (see Fig. 1). In the remainder, we will use the term *lifecycle transition* to denote this meaning. Finally, event can mean an external event object that is present in the event processing platform and thus represents some real-world happening. Hence, these events are already abstractions of real-world happenings represented in an IT system. We will be using the term *external event* to refer to this kind of events.

3 Requirements Analysis

The goal of this contribution is to find an end-to-end solution for integrating real-world events into business process execution. Now, to come up with the framework it was necessary to elicit the requirements for the integration. Therefore, we explored the state-of-the-art of standard process engines as well as complex event processing techniques. The BPMN specification [17] built the foundations

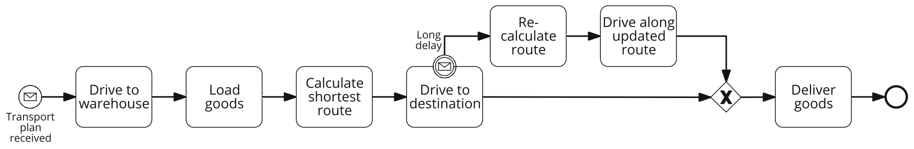


Fig. 2. Use case from logistics domain

for usage of activities and events in a process model. On the other hand, the literature survey discussed in Sect. 4 gave us insight about several concepts needed to be considered while integrating external events into processes. The project partners and domain experts from both academia and industry contributed vastly to extract use cases in IoT environment.

The process model in Fig. 2 represents one of those use cases from logistics domain. The process starts when the truck driver receives the transport plan from the logistics company. Then she drives to the warehouse to load goods. After the goods are loaded, the driver follows the shortest route to the destination. While the driving activity is ongoing, the driver gets notified if there is a long delay caused by an accident or traffic congestion. If the notification for long delay is received then the driver stops following the same route. Rather, she calculates alternate routes which might be faster and follows the best of those. Once the destination is reached the goods are delivered and the process ends. Based on the above scenario, the requirements for using events in processes are identified and described in the rest of this section.

Requirement 1: Separation of Concerns

Using external events in business processes is essentially connecting the two fields of business process management and complex event processing. As seen in the use case, event information can improve the process execution with respect to flexibility, monitoring and efficiency by reacting on occurrences in the environment in a timely manner. Process engines could directly connect to event sources by querying their interfaces, listening to event queues, or issuing subscriptions. However, from a software design perspective this design decision would dramatically increase the complexity of the engine and violate established principles like single responsibility and modularity. Therefore, we consider separation of concerns between process behavior and event processing as major requirement.

Different event sources produce events in different formats, e.g., XML, CSV, JSON, plain text, and over different channels, e.g. REST, web service, or a messaging system. In the example scenario, the probable event sources are the logistics company, the GPS sensor in the vehicle, the traffic API and each of them might have their own format of producing events. If the process engine were directly connected with event sources, it would need to be extended with adapters for each of the sources to parse the events. On the other hand, certain events can be interesting for more than one consumer. For example, the long delay event might be relevant not only for the specific truck driver, but also for other cars following the same route. CEP platforms are able to connect to different event sources as well as they can perform further operations on event streams [13]. Single event streams can be filtered based on certain time window, specific number of event occurrences or attribute values of the events. Also, multiple events from multiple event streams can be aggregated based on predefined transformation rules to create complex events relevant for a process.

To include all these functionalities in a process engine will increase the complexity and redundancy of the engine to a great extent. Instead, it is more efficient to use a separate event platform for complex event processing. The event consumers can then subscribe to the event platform for being notified of the relevant events. This separation of event processing logic is also efficient from the maintenance perspective. If there is a need to change the event source or the aggregation logic, then the process model does not need to be touched.

Requirement 2: Representation of Event Hierarchies

Simple event streams generated from multiple event sources can be aggregated to create complex or higher-level events. One could argue to use BPMN parallel multiple events to represent the event hierarchy, at least to show the connection among simple and complex events. However, using that approach one cannot express the different dimensions of event aggregation such as sequence, time period, count of events or the attribute values. Different patterns of event sequences are thoroughly discussed in [15] whereas a structured classification for composite events can be found in [3]. Moreover, this would complicate the process model and defeat their purpose to give an overview of business processes for business users. As an user of BPM, one would be interested to see the higher-level event that influences the process, rather than the source or the structure of the event. For example, the driver is only interested to know if there is a long delay that might impact her journey, but she does not care what caused the delay.

Using event hierarchies, the process model includes only the high-level business events relevant for the process and easily understandable by business users. The model is not burdened with details of event sources and aggregations, which instead are dealt with by event hierarchies in the event platform. Event hierarchies also improve maintainability, because the process model need to be adapted whenever event sources or the format of events changes. Therefore, we consider event hierarchies represented through event processing techniques as requirement for successful integration.

Requirement 3: Execution of Event Integration

Incorporating the above two requirements, the logical distribution is made from the architectural point of view as well as the representations of event processing and process execution. But the technical requirements from the implementation aspects are still remaining. We define following three technical requirements to realize the integration of events and processes.

R3.1: Binding Events. The higher-level events modeled in the process model needs to be mapped with the event hierarchy defined in the CEP platform to make sure that the correct event information is fed to the process. E.g., the driver should be informed only about the delay in the route she is following.

R3.2: Receiving Events. The process engine should listen to specific event occurrences relevant for the process execution. In other words, the driver must subscribe for the `Long delay` event to get notification, as modeled in Fig. 2.

R3.3: Reacting on Events. The driver needs to decide if the alternate route is faster than the current one and for that she needs to know the duration of the delay. Therefore, information carried by the events should be stored for later use in the process.

4 Related Work

Over the last decade, the BPM community adopted concepts from the field of complex event processing (CEP) and event-driven architectures (EDA). Several approaches were presented aiming to extend BPMN with modeling constructs for concepts of CEP [1,3,11]. Some of those approaches provide execution support in the form of an engine. Decker and Mendling [9] present a conceptual framework for process instantiation based on external events. Other approaches use external events to monitor running business processes [2,5,13], predict deviations [6], check compliance to the process model [20], and calculate KPIs [11]. In this section we describe these related approaches and check them against the requirements.

Barros et al. [3] discuss events in business processes and touch on many of the topics we consider in our contribution, like event subscription, occurrence, matching, and unsubscription. The authors present a catalog of event patterns used in real-world business processes and find that most of those patterns are neither supported by BPEL nor BPMN. Similar to *Requirement 2*, they identify support for event hierarchies, i.e. aggregation of low-level events into high-level business events, as important but yet unsupported. They suggest to integrate descriptions of event patterns into process modeling languages and consequently extend engines to handle such patterns.

This contradicts *Requirement 1* to separate concerns between process execution and event processing, but can be understood in light of the limited types of events supported by BPEL (only message receive and timer events). Related concepts to *Requirement 3* are discussed in this work. The causal ordering among event subscription, event occurrence and event consumption proposed by them is shown in Fig. 3. According to it, an event can be consumed only if there exists a subscription and the event has already occurred. Though we have followed the same ordering in our implementation, the authors in [3] did not implement the suggested engine.

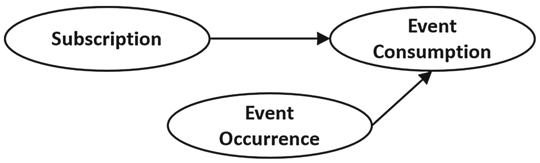


Fig. 3. Causal ordering between event subscription, occurrence and consumption

Estruch and lvaro [11] propose an IT solution architecture for the manufacturing domain that integrates concepts of SOA, EDA, business activity monitoring (BAM), and CEP. They suggest to embed event processing and KPI calculation logic directly into process models and execute them in an extended BPMN engine. This complicates the understanding of process models and contradicts *Requirement 1*. The authors sketch such an engine for executing their extended process models, but refrain from giving technical details, like handling of subscriptions or event format, thus failing to meet *Requirement 3*. However, they suggest that some processes collect simple events, evaluate and transform them, and provide high-level events for use in other process instances, realizing an event hierarchy (*Requirement 2*).

Processes from the logistics domain, which contain long-running activities, e.g. a task 'shipment by truck', needs continuous monitoring. In these scenarios external events, e.g. GPS locations sent by a tracking device inside the truck, can provide insight into when the shipment task will be completed. Appel et al. [1] integrate complex event processing into process models by means of event stream processing tasks that can consume and produce event streams. These are used to monitor the progress of shipments and terminate either explicitly via a signal event or when a condition is fulfilled, e.g. the shipment reached the target address. While these tasks are active, they can trigger additional flows if the event stream contains some specified patterns. The authors provide an implementation by mapping the process model to BPEL and connecting the execution to a component called eventlet manager that takes care of event processing. Thus *Requirement 1 and 3* are fulfilled, however, *Requirement 2* is not supported. Authors in [16] facilitate an integrated architecture for using events to monitor and predict process execution. The butterfly architecture analyzes the need of external event input in the process and generates CEP rules from historical data. But they do not talk about the conceptual and technical challenges of the integration.

For processes, in which some tasks are not handled by the POIS, monitoring of events can be used to determine the state of these tasks, e.g. to detect that an user task terminated. When a process is not supported by a POIS at all, monitoring can still capture and display the state of the process by means of events. For example, Herzberg et al. [13] introduce Process Event Monitoring Points (PEMPs), which map external events, e.g. a change in the database, to expected state changes in the process model, e.g. termination of a task. Whenever the specified event occurs, it is assumed that the task terminated, thus allowing to monitor the current state of the process. The authors separate the process model from the event processing and allow the monitored events to be complex, high-level events thus fulfilling *Requirements 1 and 2.* The approach has been implemented, however the event data is not used by and does not influence the process activities. Rather the engine uses them to determine the current state of the process instance. Therefore, we consider *Requirement 3.3* to be unfulfilled.

A framework for predictive monitoring of such continuous tasks in processes is presented in the work by Cabanillas et al. [6]. The framework defines monitoring points and expected behavior for a task before enactment. Then event information from multiple event streams are captured and aggregated to have a meaningful interpretation. These aggregated events are then used to train the classifier and later the classifier can analyze the event stream during execution of the task to specify whether the task is following a safe path or not. [20] derive event queries from the control flow of a process model, deploy them to a event engine and use them to find violations of the control flow. A similar derivation of event queries from the process model is done by [2]. These work have in common that they use external event processing (*Requirement 1*) and are implemented. However, just like [13], the events are not used to drive the process instance, but rather to find out something about it.

On the other hand, Decker and Mendling [9] conceptually analyze how processes are instantiated by events. They propose a framework named CASU which specifies when to create new process instances (C), which control threads are activated due to this instantiation (A), which are the remaining start events that the process instance should still subscribe to (S), and when should the process instance unsubscribe from these events (U). Because of its conceptual nature, *Requirement 3* is not fulfilled and we cannot judge *Requirement 1*, as the paper does not mention an architecture. However, the CASU framework satisfies *Requirement 2*, although partially. They focus on process instantiation and therefore, concentrates only on single or composite start events. On the other hand, we consider not only the start events, but also the intermediate or boundary events as well as the event based gateway.

Finally, we consider the state-of-the-art for implementing a process engine that supports event integration. Namely, we look into the popular open source process engine Camunda [7]. Using events for executing processes is also an area which has gained a lot of interest in past few years. The standard process engines like Camunda support BPMN events for starting a process instance or to choose between alternative paths following a gateway. However, Camunda does not care about the receiving part of the message event. The engine has interfaces that can be connected to a JMS queue or a REST interface but the reception of messages is not implemented. Also, there is no existing process engine that supports complex event processing. On the other hand, the event processing platforms do not have any engine to implement the generated events. Moreover, the mapping between external events and BPMN events is not there.

5 Conceptual Framework

This section presents the conceptual framework for integrating events into processes. Keeping in mind the requirements specified in Sect. 3, we discuss the aspects to be considered. Also, the proposed solutions that we came up with for each aspect are mentioned in the context of the use case presented before.

5.1 Event Generation and Aggregation

In our use case, we need two events for the process execution, a catching start event and a catching interrupting boundary event. The start event is created based on the input from the logistics company. The transport plan contains the location of warehouse to load goods, the destination for delivery and the deadline for delivery. This is an example of simple event which might be sent to the truck driver via email or even as a text message directly from the logistics company. The boundary event, on the other hand is definitely a higher-level event. Considering *Requirement 1: Separation of Concerns* this complex event is created in the event processing platform. Since we did not have access to real "truck positions", we used the sensor unit Bosch XDK developer kit[1], a package with multiple integrated sensors for prototyping of IoT applications. The unit sends measurement values over wireless network to a gateway. The gateway then parses the proprietary format of the received data and forwards it to Unicorn using the REST API. The traffic updates was received from *Tomtom Traffic Information*[2]. If there is a delay above a threshold and the location of the source of delay is ahead of the current GPS location of the truck, then a `LongDelay` event is produced.

In Unicorn, event aggregation rules are written accordingly to generate the high-level event `LongDelay`. Since Unicorn has the Esper engine at its core, we used Esper Event Processing Language (EPL) [10] for writing event aggregation rules. The event types can be registered in Unicorn as following:

```
CREATE schema Disruption
(latitude float, longitude float,
reason string, delay double);
CREATE schema CurrentLocation
(latitude float, longitude float, destination string);
CREATE schema LongDelay
(reason string, delay double, destination string);
```

[1] see http://xdk.bosch-connectivity.com.
[2] see https://www.tomtom.com/en_gb/sat-nav/tomtom-traffic/.

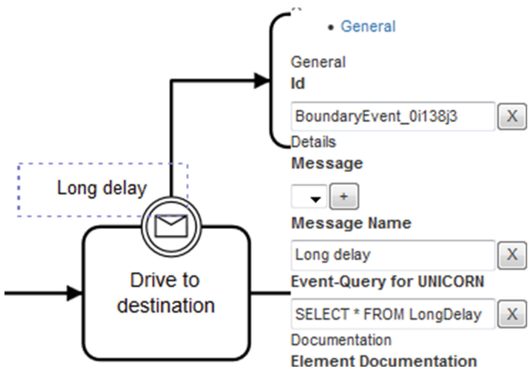
The aggregation rule for creating `LongDelay` may look like the following. The function `distance()` is not defined in EPL though. We implemented it to find out if the disruption is ahead of the truck or not.

```
INSERT INTO LongDelay
SELECT d.reason as reason,
d.delay as delay,
l.destination as destination
FROM pattern[every d=Disruption-> l=CurrentLocation
WHERE distance(d.latitude, d.longitude, destination)
< distance(l.latitude, l.longitude, destination)];
```

5.2 Event Binding Points

Event binding points are those elements of process model where events with different properties (see Sect. 2) are mapped to each other. For example, the external event `LongDelay` is needed to be mapped to the BPMN event `Long delay` that has been modeled in the process. To enable that, process models have to be extended by *event annotations* that are used as event binding points. These event binding points specify which events to listen to at what point in the model. In BPMN, external events are usually modeled with the help of catching message events (see Fig. 2). To receive these events, we need to make sure that subscriptions for the events are made. Therefore, a subscription query for each event is added to the process model in design time. Receiving the subscribed events allows to create new process instances, similarly to process instantiation in [9], and to react on the intermediate events from external event sources. To simplify the annotation language, simple queries are added in the model to subscribe to the aggregated high-level events. More complex event queries to produce these high-level events are generated by aggregation rules inside the event processing platform, as per *Requirement 2: Representation of Event Hierarchies*. For example, the annotation for the start event looks like `SELECT * FROM LongDelay` which abstracts from the complexity of event queries dealt in CEP platform. Figure 4 shows an example of modeling event queries represented as event annotation.

On the other hand, lifecycle transitions, like `terminate` in the activity lifecycle, also provide event binding points [21]. These event binding points are required to automatically change the state of an activity instance, thus enabling their monitoring [5]. The start and end of each activity like `Load goods` or `Deliver goods` are required to monitor the status of the shipment. Whereas, the cancellation of the activity `Drive to destination` suggests that the previously calculated delivery time might be postponed due to delay on the way.



General
Id
BoundaryEvent_0i138j3
Details
Message
Message Name
Long delay
Event-Query for UNICORN
SELECT * FROM LongDelay
Documentation
Element Documentation

Fig. 4. Modeling of event queries

5.3 Event Subscription and Correlation

Before they can be executed, process models have to be deployed from the modeling component to the process engine. During the deployment, event annotations, like the other information in the process model, are parsed and stored in a database. Once the queries are registered to the CEP platform, whenever a matching event occurs, the platform notifies all the subscribers.

When it comes to registering event queries the *lifespan* of event annotation plays a central role. By lifespan we mean the time between registering an event query at the event processing platform and removing the subscription. A process model works as a blueprint for several process instances [21] and subscription can be done by a process model or a process instance. The annotation of the start event binding point needs to be registered right after deployment, as it is needed for process instantiation. So, subscription for `Transport plan received` is done at process deployment. In our case, the truck driver might register to the mailing list of the logistics company to receive transport plans. Other annotations, e.g., annotation for event binding point of `Long delay` is registered later for each process instance separately.

Often, event queries have a limited lifespan during a process execution. For example, we no longer need to listen to a boundary event that might occur after the activity it was attached to has terminated and can unregister the query. In our use case, the driver stops listening to `Long delay` once she changes the route or reaches the destination. On the other hand, process trigger queries can only be unregistered when the process model is undeployed from the engine.

To handle the subscription, we extend the *execute*-method of the event nodes. When the process execution flow reaches this event node, the subscription query and a notification path is sent to Unicorn. Unicorn sends an UUID in response which is then stored in Chimera as a correlation key. When an event occurs, Unicorn checks if there is an existing subscription for this event. If a matching query is found, then Unicorn sends the notification to the provided path along with the UUID. Chimera matches this UUID to the one stored before and correlates

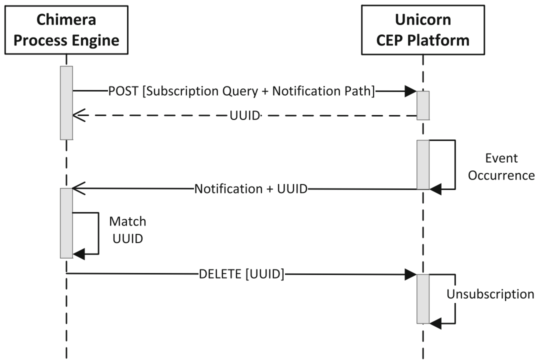


Fig. 5. Event subscription and correlation using Chimera and Unicorn

the event to the process instance. Once the event is consumed, the *leave*-method of the event node performs a `DELETE` operation for unsubscription. The above sequence is depicted in Fig. 5.

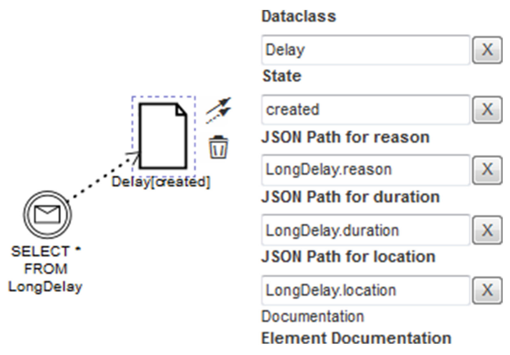
The attribute values of events can be used to filter out events irrelevant for the current process instance. For example, only events that occur in the same route as the truck is following might be interesting for that particular transport. Therefore, the annotated event queries may contain expressions referring to event attribute values. These expressions follow the dot-notation known from object oriented programming, e.g. `Disruption.route`. In some cases, however, the correlation is less direct and the filter criterion is either not available or not restricted inside the scope of the process instance. In such cases we assume that the event processing platform provides a correlation key, as described above.

5.4 Reaction on Events

In many cases, receiving an event notification from the event processing platform simply causes an BPMN event to occur. The further reaction follows the BPMN execution semantics [17]. The notification of a start event can start a new instance of a process. For example, each customer complaint can start one handling process for a manufacturer. Again, a notification of a boundary error event causes the abortion of the associated running activity and enables error handling, as discussed in our example process.

While in these cases only the fact that the event occurred is relevant, in other cases the content of the event is also of interest. E.g., the `Long delay` event certainly abides by the BPMN behavior for boundary events, but the driver might look into the information carried by the event to know how much delay has been caused and whether the alternate route will be faster or not. Therefore, notifications need to have a defined structure that allows to access the contained data for further use in the process.

To use event data in the further execution, we suggest to map the data contained in the notification to the attributes of a data object. This data object



Dataclass
Delay
State
created
JSON Path for reason
LongDelay.reason
JSON Path for duration
LongDelay.duration
JSON Path for location
LongDelay.location
Documentation
Element Documentation

Fig. 6. Event data is written into newly created data object

might already exist at the time the event notification is received, or it can be created anew. The mapping is specified in the outgoing data object, which for each of its attributes has an expression specifying how to derive the attribute value from the notification. This is depicted in Fig. 6 which shows the boundary event and the property editor for the data object. Technically, this mapping is achieved by representing event notifications in the JSON notation[3] and giving a path expression for each attribute of the target data object that defines how the value can be derived from the notification.

An alternative would have been to directly use the event object to reuse data later on in the process, instead of mapping it to a data object. We decided against this option, because events are singular occurrences, whereas data objects have a lifecycle and can be changed again. For example, each temperature measurement of the sensor is an unique event that might cause the sending of an event notification when a matching query is registered. However, for the duration of a process instance execution we would like to have one data object that holds the current temperature, which of course changes, when new notification arrives for sensor events.

The third kind of reaction that the engine can perform upon receiving an event notification is to conduct a lifecycle transition. In context of our use case, the GPS location of the truck is checked against the coordinates of the destination. When the locations match, a higher-level event is generated to notify the process engine that the driver has reached the destination. The engine then changes the state of the activity `Drive to destination` from *running* to *terminated*.

6 Implementation

This section briefly describes the implemented systems used to realize the integration of processes and events, and their interplay. A coarse architecture is

[3] see http://goessner.net/articles/JsonPath/.

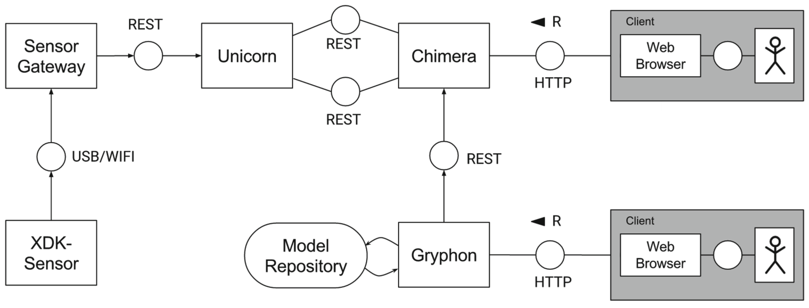


Fig. 7. Architecture

depicted in Fig. 7 with a sensor gateway as one specific example of an event source (left side).

6.1 Unicorn Event Processing Platform

Unicorn, first described in [13], is the event processing platform of choice for the implementation of our approach. It is build around Esper[4] and manages event types, event queries, and notifications both via a web-based UI and a REST API. Notifications are delivered via email, the Java Message Service (JMS), by calling back registered REST endpoints, or viewed in the UI.

There are different ways to connect Unicorn to external event sources. Event sources can simply use Unicorn's REST API to send events or publish them to a specific JMS channel to which Unicorn listens. This is feasible if the code of the event source can be changed or an intermediary gateway is used, which collects events, for example from sensors, and forwards them to Unicorn.

Unicorn also supports active pulling of events by means of adapters, which periodically call webservices. Adapters have to be configured programmatically for each event source that should be accessed. However, Unicorn offers a framework to easily extend existing adapters.

Historic events available as comma-separated values (csv) or spreadsheets (xls) can also be parsed and imported as event stream into Unicorn. Replaying such events keeps their order and time-lag, which allows to test pattern detecting queries and aggregation rules. Finally, Unicorn has a built-in event generator that uses value ranges and distributions to generate realistic events that can be used for testing event-driven applications.

Creation of high-level events is handled by *aggregation rules*, i.e. event queries that transform a pattern of events into another, higher-level event. These aggregation rules are defined by domain experts for each business scenario.

[4] see http://www.espertech.com/products/esper.php.

6.2 Gryphon Case Modeler

The second component is Gryphon, a web-based modeler for process models. Additionally, it allows to create a data model, i.e. a specification of data classes and attributes used in process model. Each data class defines possible states and valid state transitions for their instances, i.e. data objects, at runtime, called object life cycle. Process models can be directly deployed to a running Chimera instance. Gryphon builds on a node[5] stack and uses bpmn.io[6] which is an open-source BPMN modeler implemented in Javascript, while the other components are developed by our team.

For this contribution we extended Gryphon with the functionalities to annotate process elements with event annotations and model event types. The data model editor distinguishes between event types and data classes. While both are named sets of typed attributes, the former need to be registered with the event processing platform Unicorn when the model is deployed. Event annotations can be attached to certain elements in process fragments models, corresponding to the event binding points defined in Sect. 5.2. We decided to reuse the symbol for catching message events to model waiting for external events, because event notifications can be considered messages. For lifecycle binding points, the annotations are stored as property of the transitions in the lifecycle diagram, and for model-level binding points they can be specified in the model overview. As we use the event processing platform Unicorn that builds atop of Esper, event annotations have to use the EPL query language.

6.3 Chimera Case Engine

The final component is Chimera, a process engine that can also execute case models according to the fragment-based case management approach (fCM) [14]. It supports user activities with forms for data entry, as well as automatically executed email and web-service tasks. Attributes of data objects can be used as variables in email text, web-service calls, and gateway conditions. Variables are substituted by attribute values when sending email, calling web-service, or checking which sequence flow to enable.

The front-end displays all available process models and allows users to start new cases or work on running cases. Chimera follows the common worklist approach, displaying enabled activities to knowledge workers who can select and start them. Enablement of activities depends on sequence flow, i.e. preceding activities need to have terminated. Also, the data flow i.e. required data objects need to be available in the state as specified by the data input set of an activity. When terminating a running activity, knowledge workers can enter data stored in data objects. However, the resulting state of the data object needs to conform to the data output set specified in the model.

Discussion. As the architecture was developed as an academic prototype highlighting research challenges, we abstained from implementing well-understood

[5] see http://node.js.

[6] see http://bpmn.io.

features required for business use, like user and role management, or database accessors.

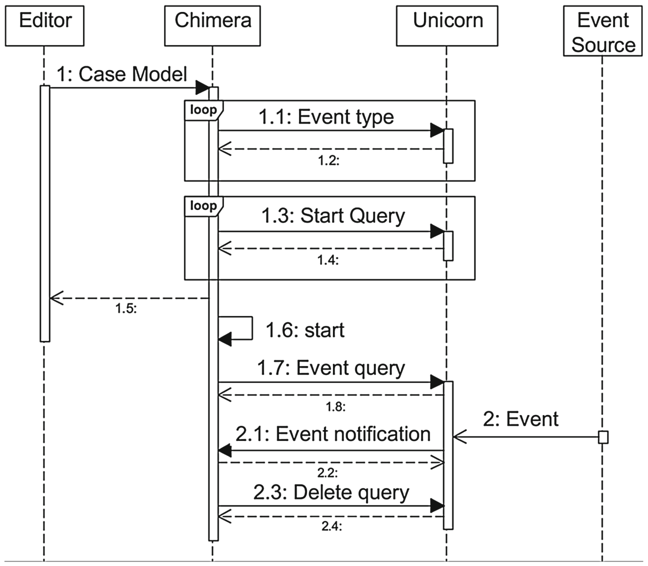


Fig. 8. Interaction sequence of components

The interaction sequence among the above described components is depicted in Fig. 8. We used the implemented architecture and depicted communication sequence between events and processes to realize several use cases from different application domains to evaluate the practicality of our framework. As already mentioned, Chimera supports case management along with process execution. Therefore, the catching start event functionality has been tested to trigger process instances as well as cases and fragments[7]. Different use cases had different usage of events such as boundary events or event-based gateways. Our architecture was able to handle all of them efficiently. To connect the event processing platform to different event sources we used sensors as well as Web API or live RSS Feeds[8]. The aggregation rules were specified according to the input extracted from the corresponding domain experts for each use case.

7 Conclusion and Future Work

The work presented here addresses a relevant situation of the current business world, as web services and IoT increase the amount of external events relevant

[7] see screencast: https://bpt.hpi.uni-potsdam.de/Chimera.
[8] e.g., http://www.eurotunnelfreight.com/uk/contact-us/travel-information/.

for business processes. Our work focuses on integrating such external events into business processes and making use of them for process execution. To bridge the gap between the conceptual level of the process model and the technical details necessary to execute it, certain aspects need to be considered as shown by our use case driven requirement elicitation. Based on those requirements, we present a conceptual framework for the integration of external events, defining event binding points, event annotations, as well as subscription and correlation mechanisms. These concepts are implemented into a system consisting of a modeling component (Gryphon), a process engine (Chimera), and an event platform (Unicorn), thus enabling the integration of real-world events into business processes.

Although our solution architecture handles the basic BPMN event constructs such as message or timer events, boundary events and event-based gateways, other event constructs like signal or error events have not been considered and are left to be implemented. Along with the events, decisions also play a big role in business processes [4]. The explicit use of decisions in processes becomes more popular with the recently released standard Decision Model and Notation (DMN) [18]. As discussed in the motivating example, the truck driver can check the duration of the delay caused by the disruption and decide whether to take an alternative route or not. Therefore, the next logical extension of our framework is to integrate decision management.

References

1. Appel, S., Frischbier, S., Freudenreich, T., Buchmann, A.: Event stream processing units in business processes. In: Daniel, F., Wang, J., Weber, B. (eds.) BPM 2013. LNCS, vol. 8094, pp. 187–202. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40176-3_15
2. Backmann, M., Baumgrass, A., Herzberg, N., Meyer, A., Weske, M.: Model-driven event query generation for business process monitoring. In: Lomuscio, A.R., Nepal, S., Patrizi, F., Benatallah, B., Brandić, I. (eds.) ICSOC 2013. LNCS, vol. 8377, pp. 406–418. Springer, Cham (2014). doi:10.1007/978-3-319-06859-6_36
3. Barros, A., Decker, G., Grosskopf, A.: Complex events in business processes. In: Abramowicz, W. (ed.) BIS 2007. LNCS, vol. 4439, pp. 29–40. Springer, Heidelberg (2007). doi:10.1007/978-3-540-72035-5_3
4. Batoulis, K., Meyer, A., Bazhenova, E., Decker, G., Weske, M.: Extracting decision logic from process models. In: Advanced Information Systems Engineering - Proceedings of 27th International Conference, CAiSE 2015, Stockholm, Sweden, 8–12 June 2015, pp. 349–366 (2015)
5. Baumgrass, A., Herzberg, N., Meyer, A., Weske, M.: BPMN extension for business process monitoring. In: EMISA, pp. 85–98 (2014)
6. Cabanillas, C., Di Ciccio, C., Mendling, J., Baumgrass, A.: Predictive task monitoring for business processes. In: Sadiq, S., Soffer, P., Völzer, H. (eds.) BPM 2014. LNCS, vol. 8659, pp. 424–432. Springer, Cham (2014). doi:10.1007/978-3-319-10172-9_31
7. Camunda: Camunda BPM platform. https://www.camunda.org/
8. Chimera: Case engine. https://bpt.hpi.uni-potsdam.de/Chimera
9. Decker, G., Mendling, J.: Process instantiation. Data Knowl. Eng. **68**(9), 777–792 (2009). http://dx.doi.org/10.1016/j.datak.2009.02.013

10. EsperTech: Esper Event Processing Language EPL. http://www.espertech.com/esper/release-5.4.0/esper-reference/html/
11. Estruch, A., Heredia Álvaro, J.A.: Event-driven manufacturing process management approach. In: Barros, A., Gal, A., Kindler, E. (eds.) BPM 2012. LNCS, vol. 7481, pp. 120–133. Springer, Heidelberg (2012). doi:10.1007/978-3-642-32885-5_9
12. Etzion, O., Niblett, P.: Event Processing in Action. Manning Publications Co., Greenwich (2010)
13. Herzberg, N., Meyer, A., Weske, M.: An event processing platform for business process management. In: EDOC. IEEE (2013)
14. Hewelt, M., Weske, M.: A hybrid approach for flexible case modeling and execution. In: La Rosa, M., Loos, P., Pastor, O. (eds.) BPM 2016. LNBIP, vol. 260, pp. 38–54. Springer, Cham (2016). doi:10.1007/978-3-319-45468-9_3
15. Luckham, D.C.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley, Boston (2010)
16. Mousheimish, R., Taher, Y., Zeitouni, K.: The butterfly: An intelligent framework for violation prediction within business processes. In: Proceedings of the 20th International Database Engineering & Applications Symposium, IDEAS 2016, pp. 302–307. ACM, New York (2016). http://doi.acm.org/10.1145/2938503.2938541
17. OMG: Business Process Model and Notation (BPMN), Version 2.0., January 2011
18. OMG: Decision Model and Notation (DMN), Version 1.1., June 2016
19. UNICORN: Complex event processing platform. https://bpt.hpi.uni-potsdam.de/UNICORN/WebHome
20. Weidlich, M., Ziekow, H., Mendling, J., Günther, O., Weske, M., Desai, N.: Event-based monitoring of process execution violations. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 182–198. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23059-2_16
21. Weske, M.: Business Process Management: Concepts, Languages, Architectures, 2nd edn. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28616-2