

# Proceedings of the Master Seminar on Event Processing Systems for Business Process Management Systems

Anne Baumgraß, Andreas Meyer, Mathias Weske (Hrsg.)

**Technische Berichte Nr. 102**

des Hasso-Plattner-Instituts für  
Softwaresystemtechnik  
an der Universität Potsdam





Technische Berichte des Hasso-Plattner-Instituts für  
Softwaresystemtechnik an der Universität Potsdam



Technische Berichte des Hasso-Plattner-Instituts für  
Softwaresystemtechnik an der Universität Potsdam | 102

Anne Baumgraß | Andreas Meyer | Mathias Weske (Hrsg.)

**Proceedings of the Master Seminar on Event Processing  
Systems for Business Process Management Systems**

Universitätsverlag Potsdam

### **Bibliografische Information der Deutschen Nationalbibliothek**

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.dnb.de/> abrufbar.

### **Universitätsverlag Potsdam 2016**

<http://verlag.ub.uni-potsdam.de/>

Am Neuen Palais 10, 14469 Potsdam

Tel.: +49 (0)331 977 2533 / Fax: 2292

E-Mail: [verlag@uni-potsdam.de](mailto:verlag@uni-potsdam.de)

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652

ISSN (online) 2191-1665

Das Manuskript ist urheberrechtlich geschützt.

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam

URN <urn:nbn:de:kobv:517-opus4-83819>

<http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-83819>

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:

ISBN 978-3-86956-347-3

# Preface

The increased availability of sensors disseminated in the world has led to the possibility to monitor in detail the evolution of several real-world objects of interest. GPS receivers, RFID chips, transponders, detectors, cameras, satellites, etc. concur in the depiction of the current status of monitored things. However, traditionally, Business Process Management Systems only execute and monitor business process instances based on events that originate from the process engine itself or from connected client applications.

The Green European Transportation Service (GET Service) project<sup>1</sup>, analysed typical logistics processes, explored their environments, and developed approaches in which information from the environment in which processes are executed are considered for process execution and monitoring as well as decision support in logistics. Thus, the project presents techniques and systems to plan transportation routes more efficiently and to respond quickly to unexpected events such as adverse weather or strikes, during transportation.

In the master seminar “Event Processing Systems” organized by the Business Process Technology group at Hasso-Plattner-Institut in the winter term 2014/2015, students studied in depth examples and data from real-world use cases of the GET Service project. These show opportunities for relating business processes and event processing. The corresponding prototypes showcase the variants of using UNICORN<sup>2</sup> and its extensibility for serving different scenarios in logistics. At the same time, they exemplify benefits of event processing in the domain of logistics.

In this context, this technical report documents six different logistics scenarios. First, Wong and Bülow investigate approaches for automatically revealing the influence of events in logistics processes, which are executed in a batch. Second, Rösler, Kirsten, and Günther present an approach which gives insights into the monitoring and progress calculations of multiple executions for the same business process. Third, Mensing, Reschke, and Sportleder implement a propagation algorithm to detect and visualize how single deadline changes influence multiple activities in

---

<sup>1</sup> <http://www.getservice-project.eu>, last accessed September 2015.

<sup>2</sup> <http://bpt.hpi.uni-potsdam.de/UNICORN>, last accessed September 2015.

interdependent processes. Fourth, Beck, Brehm, and Eichenberg show the progression of delays in activities to the subsequent logistics steps in a non-linear fashion with respect to their dependencies and additional time-dependent resource- and status-constraints of the activities. Fifth, Jung and Zwerg demonstrate how location-based information such as routes or traffic events can be incorporated in order to enrich data at hand and improve monitoring and planning capabilities of logistics processes. Finally, Omar and Richter introduce a novel approach to enrich location-based process monitoring in logistics with unexpected weather events, while also considering the state of the execution. The screencasts of these scenarios are available at <http://bpt.hpi.uni-potsdam.de/UNICORN/UNICORNUsage>.

We would like to express our gratefulness to all the authors for their valuable contributions, and the GET Service European Project for inspiring and promoting real-world examples of logistics processes.

September 22, 2016

Prof. Dr. Mathias Weske,  
Dr. Anne Baumgrass,  
Dr. Andreas Meyer



# Contents

<b>Preface</b>	<b>v</b>
Monitoring Batch Regions in Business Processes . . . . .	1
<i>Tsun Yin Wong and Susanne Bülow</i>	
Multi Instance Monitoring . . . . .	11
<i>Kerstin Günther, Kristina Kirsten, and Florian Rösler</i>	
DRAGON – Deadline Propagation Transcending the Boundaries of Processes	23
<i>Michelle Mensing, Jakob Reschke, and Tim Sportleder</i>	
Non-linear Delay Propagation of Event Based Business Processes . . . . .	35
<i>Heiko Beck, Maximilian Brehm, and Marius Eichenberg</i>	
Location-Based Process Monitoring . . . . .	47
<i>Pascal Jung and Thomas Zwerg</i>	
Location-based Process Monitoring: Location and Weather . . . . .	59
<i>Lucie Omar and Marvin Richter</i>	



# Monitoring Batch Regions in Business Processes<sup>\*</sup>

Tsun Yin Wong and Susanne Bülow

Hasso-Plattner-Institut

{TsunYin.Wong, Susanne.Buelow}@student.hpi.uni-potsdam.de

Recently, batch activities have been introduced to improve the execution of business processes by collectively performing batch activities that belong to different process instances. Using traditional techniques to monitor processes with batch activities leads to inadequate representation of process instances, since monitoring is unaware of batch activities. This paper introduces an approach to monitor batch activities, which also takes into account exceptions in batch clusters at different levels of abstraction. The concepts and techniques introduced are evaluated by a prototypical implementation using real-world event data from the logistics domain.

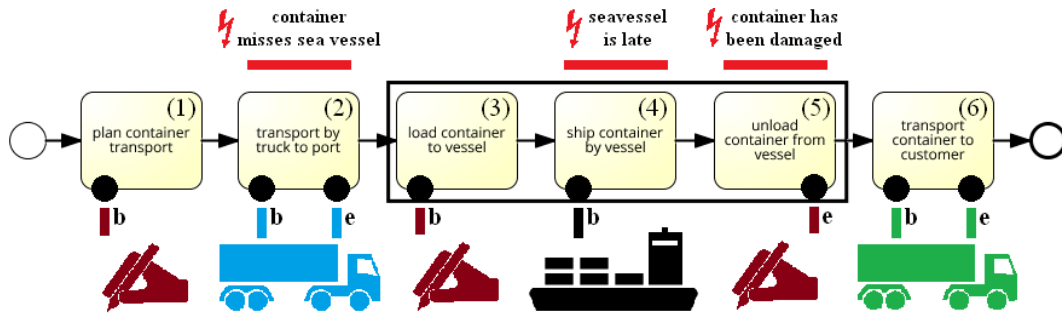
## 1 Introduction

Many organizations in business and administration represent their working procedures as business processes to improve them and to monitor their execution [9]. Recently, batch activities [7] and batch regions [6] have been proposed to collectively execute activities of different process instances. While methods and techniques for monitoring individual business processes have been proposed, these are inadequate to monitor batch activities. This paper introduces novel concepts and techniques for monitoring batch activities, which also take into account exceptions. The approach is evaluated by a prototypical implementation using real-world event data from the logistics domain.

A batch region [6] of a process model consists of activities that are executed collectively as a batch. We find batch activities in many domains, including health care (many blood samples are analyzed in a batch) and logistics (containers in a vessel are transported together). Since processes are performed non-automatically in these environments, process monitoring uses events that occur while the process is being executed. Events include the arrival of a vessel in a harbor with certain containers or the completion of a blood sample analysis in a hospital.

---

<sup>\*</sup>The research leading to these results has received funding from the European Union's Seventh Framework Program (FP7/2007–2013) under grant agreement 318275 (GET Service).



**Figure 1:** Process from logistics domain. Events from various sources are related to monitoring points ('b' for begin event, 'e' for end event of activity).

If traditional techniques for process monitoring were used in these settings, the number of monitoring events would be overwhelming. Using information about batch regions, the number of events to monitor can significantly be reduced. Furthermore, monitoring approaches need to expose the occurrence of irregular behaviour, such as exceptions. Therefore, we also provide a classification of different types of exceptions of business processes, involving individual process instances and all process instances in a batch, respectively.

The remainder of this paper is organized as follows: First, the need for batch monitoring is illustrated by a motivating example in Section 2. Then, a conceptual approach of batch monitoring is introduced in Section 3. In Section 4 the prototypical implementation is explained. In Section 5 we use the batch monitoring approach for the motivating example. Finally, Section 6 concludes this paper.

## 2 Motivating Example and Requirements

To exemplify the approach, we introduce a real world use case inspired by the GET Service project<sup>1</sup>, which is funded by the Seventh Framework Program of the European Union. GET Service aims at supporting efficient transportation planning to reduce both transportation times and empty miles, leading to a reduction of CO<sub>2</sub> emission.

The respective process model is shown in Figure 1; it consists of six sequential activities: At first, the transport planner schedules a container for transport (activity 1).

<sup>1</sup> <http://www.getservice-project.eu>, last accessed September 2015.

The container is later picked up at the warehouse and transported to the port by truck (2), where the container is loaded on a sea vessel (3). The container is then shipped to another port (4), where it is unloaded (5). Finally, the container is transported to the customer by another truck (6).

As a sea vessel transports multiple containers to the same port, activities (3), (4) and (5) are executed as a batch. Therefore we define a batch region involving those activities. Each container is represented by a process instance, whereas a sea vessel is represented by a batch cluster. To facilitate process monitoring, we assign monitoring points to each activity, which define its start and its end event. In the use case, the corresponding events are provided by port logistics (1) (3) (5), truck (2) (6) and shipping (4) companies, resp.

In the use case, three exceptions may occur:

- *Container misses sea vessel*: A container arrives with excessive delay at the port of origin and cannot be transported on the sea vessel for which it was scheduled.
- *Sea vessel is late*: The calculated arrival of the ship at the port of destination is after the planned arrival.
- *Container has been damaged*: During the unloading of the containers at the port of destination, customs notice that the container is unsealed and therefore needs further inspection.

From our scenario, we can identify two main requirements for batch monitoring:

**R1** In the traditional process monitoring approach, events indicate information about single process instances. In our use case, each container transport would represent one process instance and events about each container would be monitored individually. However, as soon as the container is loaded onto the sea vessel, it would be sufficient to be updated about the progress of the vessel, instead of the progress of the hundreds of containers on the vessel. To monitor the vessel, the events arriving for each container must be aggregated. To enable monitoring of a batch cluster, we therefore need a *batch aggregation strategy* for the events on the process level.

**R2** Exceptions occurring in batch regions need to be handled differently than exceptions during normal process executions. For example, the exception “Ship is late” would normally result in one exception for each container on the ship. In batch monitoring, it would be sufficient, to mark the sea vessel as having an exception. On the other hand, the exception “Container has been damaged” detected for a container should not result in an exception of the whole vessel, but only in an exception for the affected container. Thus, a handling for different *batch exceptions* has to be examined.

### 3 Batch Monitoring Approach

In this section, an approach for batch monitoring is introduced. To connect events to processes, monitoring points are introduced. A *monitoring point* is a binding of an event type to an activity of a process model. Monitoring points are used to measure the progress of process instances [4].

A *batch region* is a coherent part of a process, in which several process instances are executed together as *batch clusters*. Process instances with equal values in a certain group of attributes, the grouping value, will be executed in the same batch cluster [6]. *Exceptions* indicate an erroneous execution of a process instance. Several exception types on different levels of a process can be distinguished [8]. Events and event types are required but not part of our concept wherefore we refer the interested reader to [3].

On the basis of this preliminary work, the novel approach for batch monitoring is described, covering requirements R1 and R2, presented in Section 2. To allow monitoring of batch clusters as described in R1, we introduce two batch aggregation strategies for process instance events:

- *Complete Event Set Strategy*: Only if events for all process instances of a batch cluster have been observed, the cluster progress will be recognized. This is a cautious approach that needs additional exception handling in case of missing events.
- *Single Event Strategy*: The first event connected to one process instance within a batch cluster determines the cluster progress. We here assume that the correct execution of one instance directly implies the correct execution of the whole cluster. For our implementation, we chose this approach.

As far as *batch exceptions* (R2) are concerned, those have to be differentiated in exceptions outside of a batch region, which would be normal *process exceptions* and exceptions within a batch region, resp. Moreover, we consider the following two levels for exceptions in a batch region:

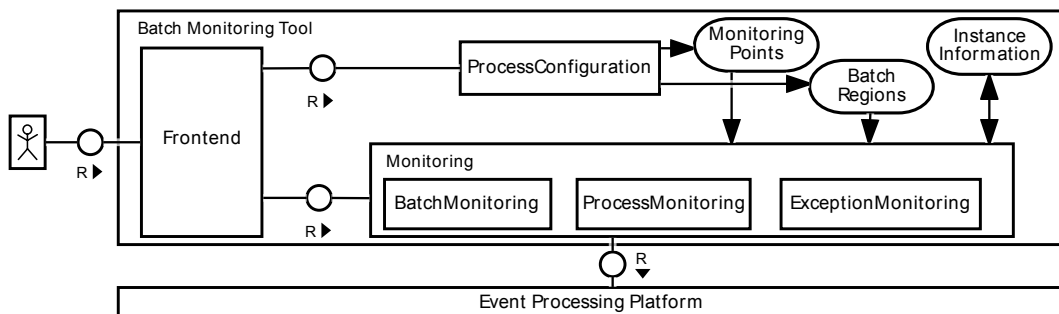
- *Batch-level Exceptions*: If the exception affects the whole batch cluster, namely all contained process instances, it is an exception on batch level.
- *Instance-level Exceptions*: If the exception affects only one process instance, this instance is then in exception and cannot be further executed together with the remaining, correct process instances in the batch cluster. It is therefore removed from the cluster and has to be handled separately. This is an exception on process level.

## 4 Batch Monitoring Tool

In this section, we present the prototypical implementation of the batch monitoring approach, including the overall architecture and a more detailed description of the implementation of batch monitoring.

### 4.1 Architecture

An overview of the system architecture is presented in Figure 2. It contains three main components. The *ProcessConfiguration* is accessed by the *Frontend* to create the monitoring points and batch regions as part of a process model. *Monitoring* includes the monitoring of process instances, batch clusters (as described in R1 of Section 2) and exceptions (as described in R2) using monitoring points and batch regions specified in the *Frontend*. The *Monitoring* is explained in detail in Section 4.2 It communicates with the Event Processing Platform (UNICORN<sup>2</sup>) introduced in [2] that consumes events provided by process engines executing the process model. Information about process instances and batch clusters are propagated to the *Frontend*. The *Frontend* allows the visual configuration of the process which is then handled by the *ProcessConfiguration*. Moreover, it offers an intuitive visualization of the progress of process instances as well as batch clusters and visualizes occurring exceptions using the information propagated from the *Monitoring*.



**Figure 2:** System Architecture of Batch Monitoring Tool

<sup>2</sup> <http://bpt.hpi.uni-potsdam.de/UNICORN>, last accessed September 2015.

## 4.2 Implementation

In this section, we focus on the implementation of batch monitoring shown as a class diagram in Figure 3. The monitoring of process instances and batch clusters as well as their exceptions are realized through monitoring points (*MonitoringPoint*). A monitoring point connects an *activity* of the process model and an *event type*. Furthermore, it can be marked as an exception (*isException*) and whether its monitored activity lies within a batch region (*isInBatchRegion*) and is therefore relevant for monitoring of batch clusters. A *MonitoringListener* triggers the monitoring point, once an event of the appropriate event type occurs. This is realized using corresponding event processing queries registered in the EPP. The monitoring point then updates the monitoring status of an existing process instance or batch cluster (*updateMonitoring()*) in the *MonitoringInformation* according to the triggered activity or creates a new process instance (*createNewInstance()*). In case of an exception, batch clusters or process instances can be marked as having an exception (*updateException()*) and whenever a batch exception on process level (*exceptionLevel*) occurs, the affected process instance will be removed from the batch cluster (*removeInstanceFromCluster*).

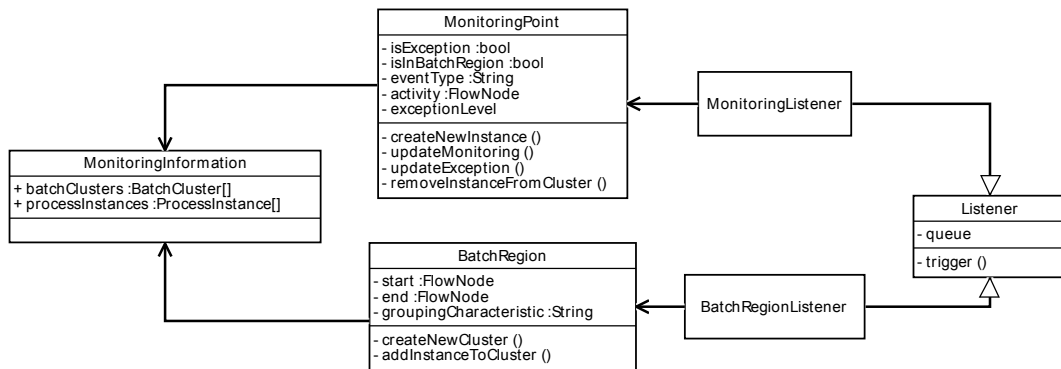


Figure 3: Class diagram of the monitoring implementation

A *BatchRegion* is defined by a *start* and *end* activity, as well as a *groupingCharacteristic*. The grouping characteristic consists of one or more attributes of an event type. On occurrence of a corresponding event, the *BatchRegionListener* triggers the batch region. The batch region determines the process instance belonging to the triggering event and adds it to an existing cluster (*addInstanceToCluster()*) or create a new one (*createNewCluster()*) in the *MonitoringInformation*, depending on its grouping characteristic value.



The tool is implemented in Java 7, using the Apache Wicket Framework for the frontend. For process import of BPMN-alike signavio.xml-files, we use the libraries jBPT and promniCAT. The ability of this tool for batch and exception monitoring is demonstrated with an example in Section 5 as well as in a screencast of the tool<sup>3</sup>.

## 5 Example Use Case

Figure 4 provides a screenshot of the batch monitoring tool, showing an example execution of the scenario described in Section 2. Each table row refers to a batch cluster or a process instance. The current activity is indicated by the column in which it is located, so the container moves to the right as it is transported. Events are utilized to monitor the begin (b) and end (e) of an activity. A circle denotes that the activity is in execution, a green tick marks the completion of the activity.

Batch regions: TransportByVessel | Selected instance | Instance with exception | Highlight batch cluster: Sea vessel 5 | Refresh dropdown

Overall progress Sort by Last update ▾

plan container transport (1)	transport by truck to port (2)	Loading container to vessel (3)	ship container by vessel (4)	Unloading container from vessel (5)	transport by truck to customer (6)	Last update
Container 12						2015-02-05 09:42:24.717
	✓ Container 11					2015-02-05 09:42:22.389
	✓ Container 10					2015-02-05 09:42:20.075
			⚠ Sea vessel 4			2015-02-05 09:42:11.622
	⚠ Container 3					2015-02-05 09:41:57.691
				⚠ Container 2		2015-02-05 09:41:49.201
					✓ Container 1	2015-02-05 09:41:00.232

**Figure 4:** Visualization of process instances and batch clusters. Containers are represented by process instances, whereas a seavessel is represented by a batch of all containers on that seavessel.

<sup>3</sup> <https://owncloud.hpi.de/public.php?service=files&t=f02387692aaa880428905d30e3f9ab89>, last accessed September 2015.

When a freight transport company schedules a container for transport, it also arranges the seavessel to export its container. On this basis, we identify process instances of the same batch cluster by taking the estimated time of departure of a seavessel as the grouping characteristic of the batch region. In Figure 4, *Container 10* and *Container 11* that have arrived at the port are expected to be grouped in the same batch cluster (follows R1 from Section 2). The transport planning for *Container 12* is ongoing, but as soon as it finishes, a possible batch cluster will be assigned to the container.

The three exception types described in Section 3 cover the exceptions in our use case (follows R2).

- *Container misses seavessel*: The corresponding process instance must be removed from the batch cluster for which it was planned. In our example, this applies to the process instance regarding *Container 3*.
- *Seavessel is not moving*: The whole batch cluster *Seavessel 4* is affected during the execution. Its details are shown in Figure 5. The batch cluster contains five process instances. The exception is triggered by an event of the type *ShipNotMoving* which is bound to the activity *ship container by vessel*.
- *Container has been damaged*: The process instance regarding *Container 2* is removed from its batch cluster and remains at the port.

With our batch monitoring concept that considers executions on both process and batch level, a transport planner sees the progress on a single view.

Batch instance Sea vessel 4		View process:
Attribute	Value	
Grouping characteristic	2013-05-16T11:00:00.000+0200	Container 6 ▾
Batch region	TransportByVessel	Choose One
Current state	RUNNING	Container 5
Creation time	2015-02-05 09:41:18.242	Container 6
Exceptions		Container 7
		Container 8
		Container 9
Event 'ShipDelay' occured in 'ship container by vessel'	2015-02-05 09:42:17.717	

**Figure 5:** Details of batch cluster *Seavessel 4* with five process instances. Its exception has been triggered by an event of type *ShipDelay*.

## 6 Conclusion and Future Work

In this paper, we have presented an approach with the corresponding implementation which enables the monitoring of batch executions, including their exceptional behaviour. The progress monitoring is driven by monitoring points triggered by events; a direct interaction with our tool to handle exceptions is not in its scope.

As of now, a BPMN process model is loaded into the monitoring tool and then complemented with monitoring points and batch regions afterwards. Future work includes the support of annotations in XML files for BPMN process models as mentioned in [1].

The batch concept presented in [5] includes the application of threshold rules and Event-Condition-Action (ECA) rules. They are currently not considered in our concept and we intend to integrate them to enable the detection of exceptions such as the exceeding of batch clusters.

Since the concept of the monitoring tool is loosely based on workflow exception patterns [8], research in how these patterns are supported in batches is required.

## References

- [1] A. Baumgrass, N. Herzberg, A. Meyer, and M. Weske. “BPMN Extension for Business Process Monitoring”. In: *Enterprise Modelling and Information Systems Architectures*. GI, 2014, pages 85–98.
- [2] S. Bülow, M. Backmann, N. Herzberg, T. Hille, A. Meyer, B. Ulm, T. Y. Wong, and M. Weske. “Monitoring of Business Processes with Complex Event Processing”. In: *Business Process Management Workshops*. Springer, 2013, pages 277–290.
- [3] O. Etzion and P. Niblett. *Event Processing in Action*. Manning Publications Co., 2010.
- [4] N. Herzberg and M. Weske. *Enriching Raw Events to Enable Process Intelligence - Research Challenges*. Technical report 73. Hasso Plattner Institute at the University of Potsdam, 2013.
- [5] L. Pufahl, N. Herzberg, A. Meyer, and M. Weske. “Flexible Batch Configuration in Business Processes Based on Events”. In: *Service-Oriented Computing*. Springer, 2014, pages 63–78.
- [6] L. Pufahl, A. Meyer, and M. Weske. *Batch Regions: Process Instance Synchronization based on Data*. Universitätsverlag Potsdam, 2014.

- [7] L. Pufahl and M. Weske. "Batch Activities in Process Modeling and Execution". In: *Service-Oriented Computing – 11th International Conference, ICSOC*. 2013, pages 283–297. DOI: 10.1007/978-3-642-45005-1\_20.
- [8] N. Russell, W. van der Aalst, and A. ter Hofstede. "Workflow Exception Patterns". In: *Advanced Information Systems Engineering (2006)*, pages 288–302.
- [9] M. Weske. *Business Process Management: Concepts, Languages, Architectures. Second Edition*. Springer, 2012.

# Multi Instance Monitoring

Kerstin Günther, Kristina Kirsten, and Florian Rösler

Hasso-Plattner-Institut

{Kerstin.Guenther, Kristina.Kirsten, Florian.Roesler}@student.hpi.uni-potsdam.de

For monitoring multiple instances in a business process, we introduce a concept to aggregate the progress of single instances to get the overall progress of a transportation process. Our implemented system is addressed to the planner of logistic companies and gives insights into the actual status of all orders and sends notifications about expected and unexpected events in real-time. The software helps to keep the overview and to react quickly to events in order to ensure a trouble-free process and to provide early information about the delivery.

## 1 Introduction

Businesses are based on processes, which consist of a set of activities in order to support and represent their business goal. The management of business processes is an important item for business administrators who are mostly interested in improving the operations of the company [14].

Especially in logistic companies an efficient management of business processes is crucial because of the intensive supply chain in which multiple activities and participants depend on each other and correlate. In today's logistics companies, there is one position that covers most directive tasks for the transportation business – the planner. Planners distribute orders among available executive units like trucks. They also continuously monitor the progress of the active transport routes and reschedule certain steps upon encountering problems. The more orders a planner is supposed to overview, the more a technical solution is inevitable to remain efficient. Such tools exist and offer functionalities to track units during the execution of orders. For that purpose event-driven systems are designed to immediately process and react to events when they occur. They enable to continuously track the status of the processes and to observe or, as the case may be, to respond to situations [2], like the re-planning of routes because of unexpected events.

The complexity of monitoring increases drastically when orders are split into multiple instances and distributed to various executive units. In that case, the planner has to keep track of all involved units to form an overall progress of the order and needs the possibility to monitor each unit individually on demand to react to unexpected situations.

As complex event processing is a relatively new field of research (cf. Gartner Hype Cycle, August 2013, [10]) the literature does not provide specific approaches for monitoring multi instances. Nevertheless literature reviews show the interest for this topic. In general, the identification of correlated events is important to understand modern business processes [7]. Besides, various patents [4, 8] demonstrate the actual demand to correlate multiple events, especially for security aspects. Moreover, it becomes obvious that the correlation of events respectively the aggregation of instances is an interesting and important approach for different areas. Equally an event processing system which is able to aggregate several instances to one single order and simultaneously evaluate events is needed in order to provide an overview and notify the planner of irregularities. Therefore, in this paper, a solution is presented that allows the planner to instantaneously grasp the progress as well as arising problems of a multi instance order, which increases efficiency and proactivity.

The general concept is described in Section 2 which is followed by the implementation in Section 3. A showcase with the description of two concrete use cases is demonstrated in Section 4. Finally the paper is concluded in Section 5.

## 2 Concept

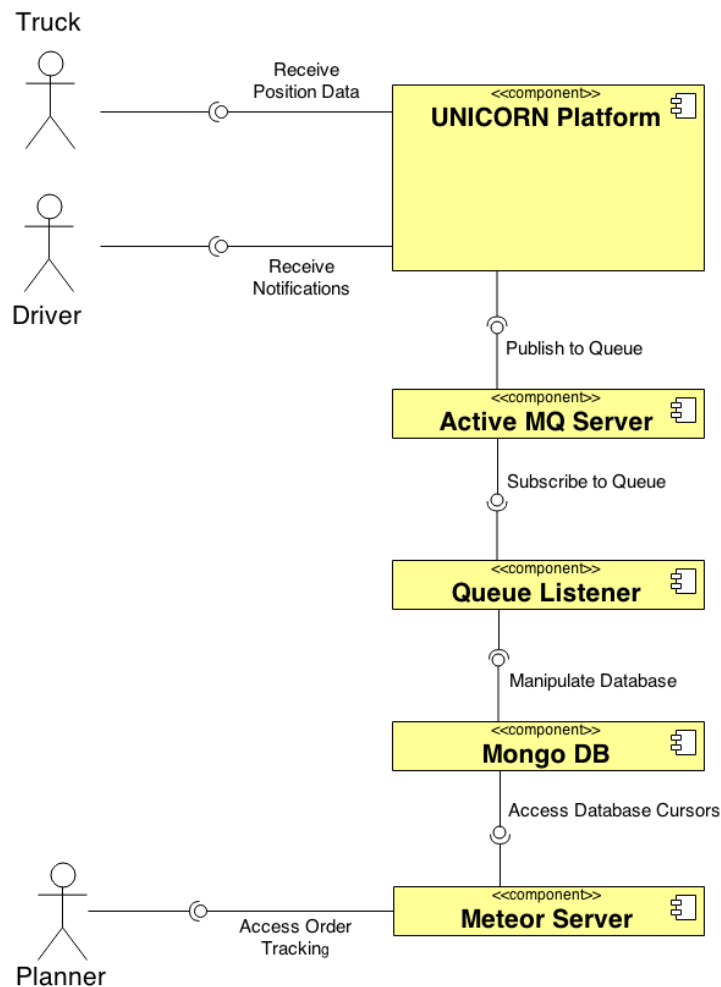
For the multi instances scenario, inspired by GET Service [3, 11], special requirements have been derived. The planner splits up the cargo into multiple instances transported by different transportation instances, e.g. trucks, and potentially on different routes. Expected as well as unexpected events during the transportation process that may apply only to a subset of all instances must be monitored in real-time keeping the planner informed about the transportation progress. The *progress* is the target-performance comparison of the order's status. To get the overall progress of the order, the progresses of each single order are added up. Moreover, the planner has to be notified about unexpected events instantaneously for re-planing the affected instance(s) to still complete the order meeting deadlines.

Based on the described requirements for the scenario, we modeled a BPMN model [9], shown in Figure 4 in the appendix, describing the work flow of our concept. The modeled process always terminates either by completing the order, i.e. all instances complete their sub-orders, or by aborting the monitoring of the order due to missing information of one or more instances.

Out of the BPMN we identified the planner's order, the sub-orders for each instance and the *traces* [12] sent by the instances (in this case trucks) as essential data objects and therefore, modeled these data items in a data model (see Figure 3 in appendix. In our implementation, the orders are given in XML format once at the start of the

transportation progress by the planner. The *traces* are position updates and optional additional notifications of the driver in XML format sent periodically by each truck. Secondary, we derived an event hierarchy, illustrated in Figure 5 in the appendix, including 13 event types and 17 event aggregations we have identified as relevant for our implementation.

Inspired by these models, we developed the architecture of our implementation, shown in the model in Figure 1. An event traverses through our system in the following way: The position data is sent by a sensor system in the truck and can be enriched with information by the truck's driver. The UNICORN platform receives



**Figure 1:** Architectural Model

this data, transforms this XML document into one event and transmits the resulting event to the internal Event Processing Agent (EPA) which applies our aggregation rules defined in Esper [4], e.g. an event of type *OrderProgress* and an event of type *TruckOrderStarted* are aggregated to an *OrderCompleted* event if the number of completed units in the *OrderProgress* event is identical with the number of units in the *TruckOrderStarted* event, i.e. all items that had to be delivered have arrived at the destination. Then, the aggregated events are transmitted over queues and further transformed to be finally visualised in our front-end.

A detailed description of the transmission and transformation as well as the design and functionalities of our front-end is given in the next section.

### 3 Implementation

For our specific implementation, as seen in Figure 1, we utilize the provided UNICORN platform [1, 13] as our central event processing system, being connected to an Active MQ server. In order to persist emitted events, we use MongoDB that is connected to our front-end, which is hosted on a Meteor server. The following paragraphs shall clarify the detailed architecture as well as the reasons for the respective product choices.

#### 3.1 Queue Listener

One major functionality of the UNICORN platform is the construction of queues on a connected Active MQ server, where processed events can be stored in various queues, depending on previously specified Esper queries. In order to persist the events, it is important to retrieve emerging events and store them in a database. Therefore, we subscribe to existing queues for the respective event types and handle incoming events accordingly with “Queue Listeners”.

The core task of the Queue Listeners is to continuously retrieve emitted events, transform them and write them to a database. As the event messages are in a proprietary format, which is not deserializable by common libraries, a transformation to JSON is required first. After that the JSON messages are deserialized into event objects by Gson<sup>1</sup>, a Java library by Google. As each event type requires individual changes to the database, the listeners execute specific program logic for each type. E.g. an “OrderCreated” event should trigger the creation of an order in the database

---

<sup>1</sup> more information on <https://code.google.com/p/google-gson/>, last accessed September 2015.



whereas a “TruckLocation” event is supposed to update the location of the respective truck in the database as well. The updated records are meant to be queried by the front-end to display the orders to the client.

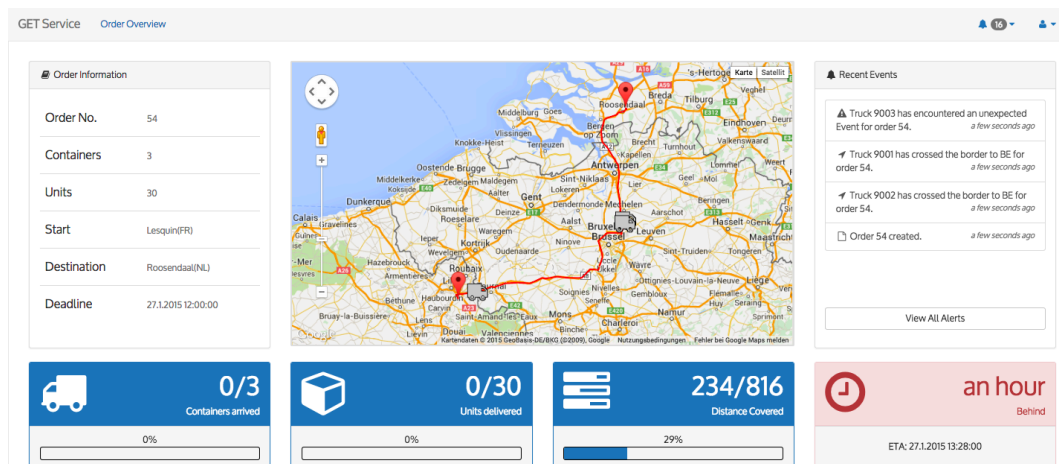
### 3.2 Meteor and MongoDB

As previously explained, one major requirement is the immediate delivery of updated information to the planner. Most web technologies like Active Server Pages (ASP) or JavaServer Pages (JSP) utilize a pull mechanism by the user to retrieve the most recent data. This means the user either has to refresh the page manually or the implementation automatically refreshes the page after a defined period of time. On the opposite in recent years several frameworks have emerged that focus on establishing connections that allow updates to be pushed to the client as soon as they are introduced on the server.

One system that offers the described functionality is Meteor, which at the time of this writing is available in its stable version 1.0.3 [5]. As Meteor is a full-stack framework based on Node.js, applications are written in JavaScript both on the server as well as on the client. Once a client connects to a Meteor server, a WebSocket connection is established, which is kept alive as long as the user remains on the page. Whenever the database that is connected to Meteor undergoes changes to one of its records, these changes are pushed to the client via the WebSocket, which is then able to update its respective user interface.

In its current release Meteor only supports MongoDB, which enables Meteor to effectively identify changes to the stored records. As MongoDB is a document-oriented database [6] we store each monitored order in a single document, which is created and continuously updated by the Queue Listeners. The Meteor server depicts the end of our architectural chain and is the system that displays relevant information to the user through a web page. A screenshot of the front-end that can be used by the planner to monitor a particular order is shown in Figure 2.

The overview (see Figure 2) is split into four parts. Firstly on the left hand panel general information are displayed about the respective order, like the destination and deadline. Those information are static and will most likely not change throughout the processing of an order. In the center the planner can follow the progress on a map, which contains markers for the current positions of the involved trucks. The right side contains a list of events that belong to the monitored order that arise throughout the execution. In our specific implementation such a list would for example contain events for a border crossing or an unexpected event that is encountered by a truck. The four boxes below display the current status of the order, allowing a quick overview through status bars that visualize the progress. The left box indi-



**Figure 2:** Screenshot of the front-end of the system visualizing the status of one order

cates how many trucks have arrived and consequently how many items have been delivered. Furthermore, the third box shows the total distance which all trucks have covered so far. The rightmost box indicates the overall delivery status of the order using the difference between the planned deadline and the current expected arrival time. If the order is on-time the box is colored green, on the other hand if one or more trucks are no longer on schedule and therefore the order will not be on-time, the box changes to red. Next to these boxes, the estimated time of arrival is displayed.

Additionally, the front-end has a built-in notification functionality that is triggered once a new event is emitted, no matter which order is concerned. In that case a little pop-up notification appears in the top right corner, informing the planner, which order is affected by what kind of event. We therefore make sure that the planner even gets updates of orders that he is currently not actively monitoring and can react quickly.

## 4 Showcase

This section describes two specific demo stories, which can be realized with the presented solution for monitoring multi instances whereby the features and characteristics will come out. In both stories the planner divides one order into three partial-orders which are assigned to three different trucks. Each truck has to carry 10 items. The freights have to be transported from Lesquin (France) to Roosendaal (Netherlands).

### 4.1 Demo Story 1

The first story presents a delivery on-time. All three trucks start at the same time in Lesquin. The current position of all trucks belonging to this order is visualized on a map which *can* be observed by the planner. One after another notifications are shown up that a truck has crossed the border to Belgium. To keep the clarity for the planner all notifications are also visible and appear as pop-up in the system even if the planner is currently observing another order. Therefore the planner can switch to the order on demand and sees in this story a green-colored ETA (estimated time of arrival) box which signals that the order can be delivered on-time. Later, the planner is notified about the border crossing to France for every truck. The three trucks arrive consecutively at the destination before the expiration of the deadline and therefore the order is delivered on-time. Again, the planner is notified about the arrival. At first for each truck a notification pops up and consequently an *Order 42 has been completed!* message shows up, assuming an order with the identification number 42.

### 4.2 Demo Story 2

In contrast to the first demo story, this story describes a situation where a delivery cannot be delivered on-time anymore. Again, the three trucks start at the same time in Lesquin. Two of the trucks cross the border to Belgium and the planner is again notified. However, the third truck gets unexpectedly into a border check. The truck waits for hours in front of the border to Belgium during the two others are moving forward to the destination Roosendaal. Consequently, the planner is informed that the order is not anymore on-time. Because the third truck does not move in the last three position updates<sup>2</sup> and no regular break is scheduled, an unexpected event is generated and sent to the system. Thereby the planner has the possibility to, for instance, contact the driver, re-plan the route or inform the customer about the delay. As long as the third truck does not arrive at the destination the order will not be completed even if the first two trucks have already been arrived.

---

<sup>2</sup> The number of position updates that have to be missed for a truck to generate an unexpected event can be configured.

## 5 Conclusion

In business management, monitoring a process consisting of multiple instances becomes more and more relevant to optimize the work flow. Based on the given scenario, we have developed a concept to aggregate the progress of each single instance to get an insight in the overall progress of all instances.

Our solution gives an overview of the details of an order and informs the planner about the current position of each instance belonging to that order, visualized on a map. Furthermore, the planner knows the current overall progress of the order at every time during the transportation and gets notified about unexpected events in real-time. In that way, the planner can react on these events on time and re-plan the route or inform the respective persons about the delay.

The event aggregations defined by us are based on the periodic input of *traces* and uses the attributes defined in [12], e.g. the type 44 denotes a border check. Our solution works without knowing such types, but without this information special event types like *BorderCheck* or *BreakStarted* will not be generated.

Other features like congestion detection, delay propagation or using additional information like weather data can be easily integrated in our system to improve the accuracy of the estimated time of arrival and enrich the events shown to the planner.

## Appendix

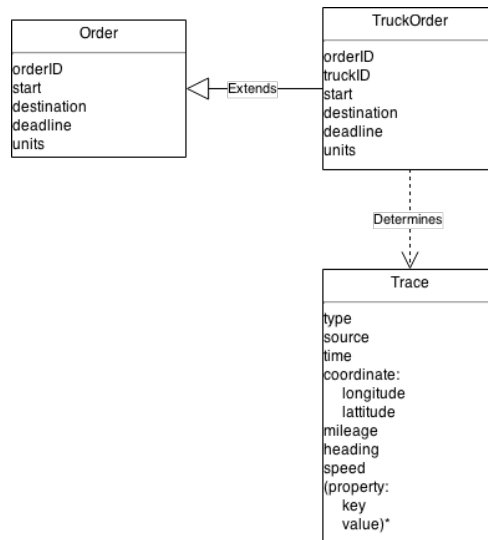


Figure 3: Data Model

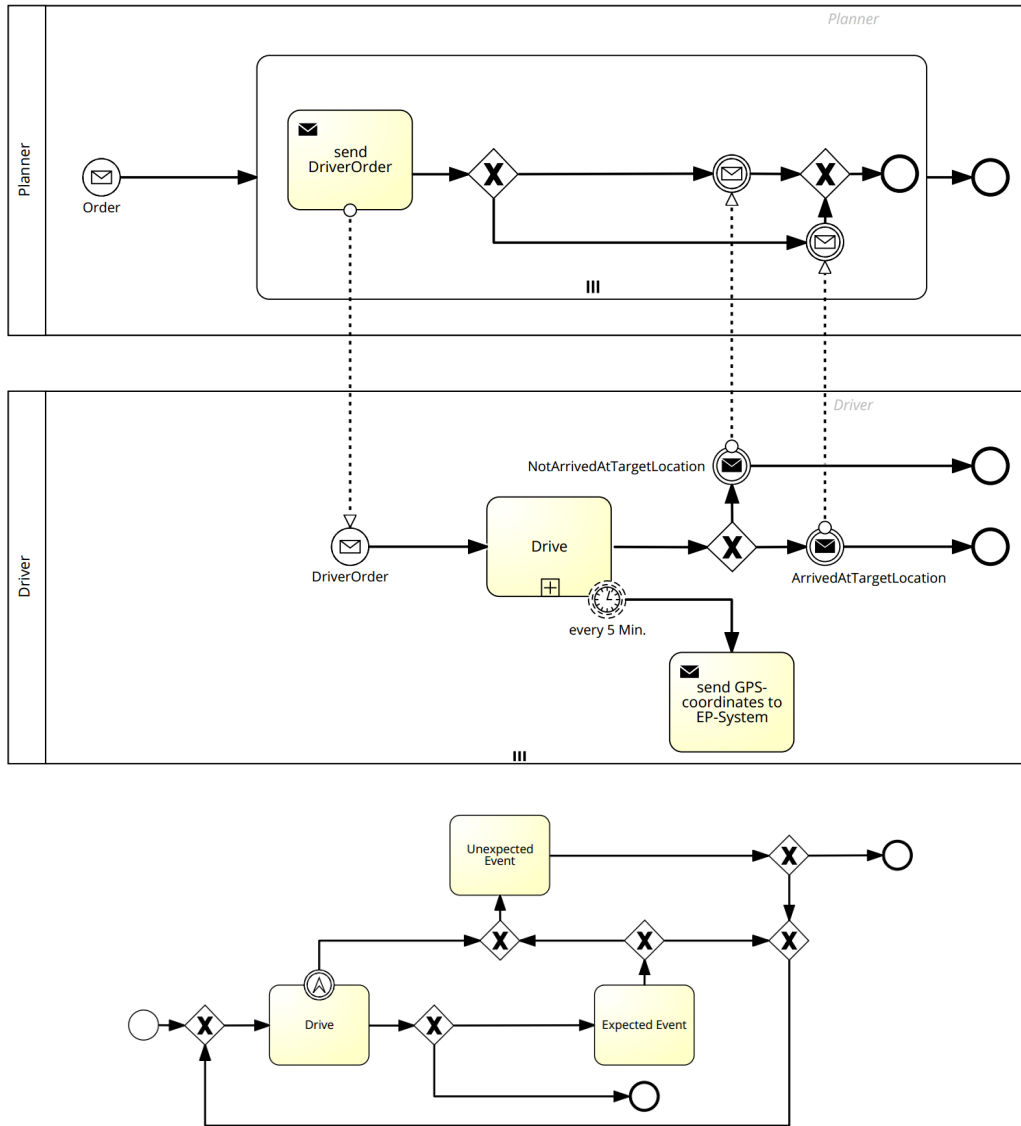


Figure 4: BPMN: Overview and task *Drive*

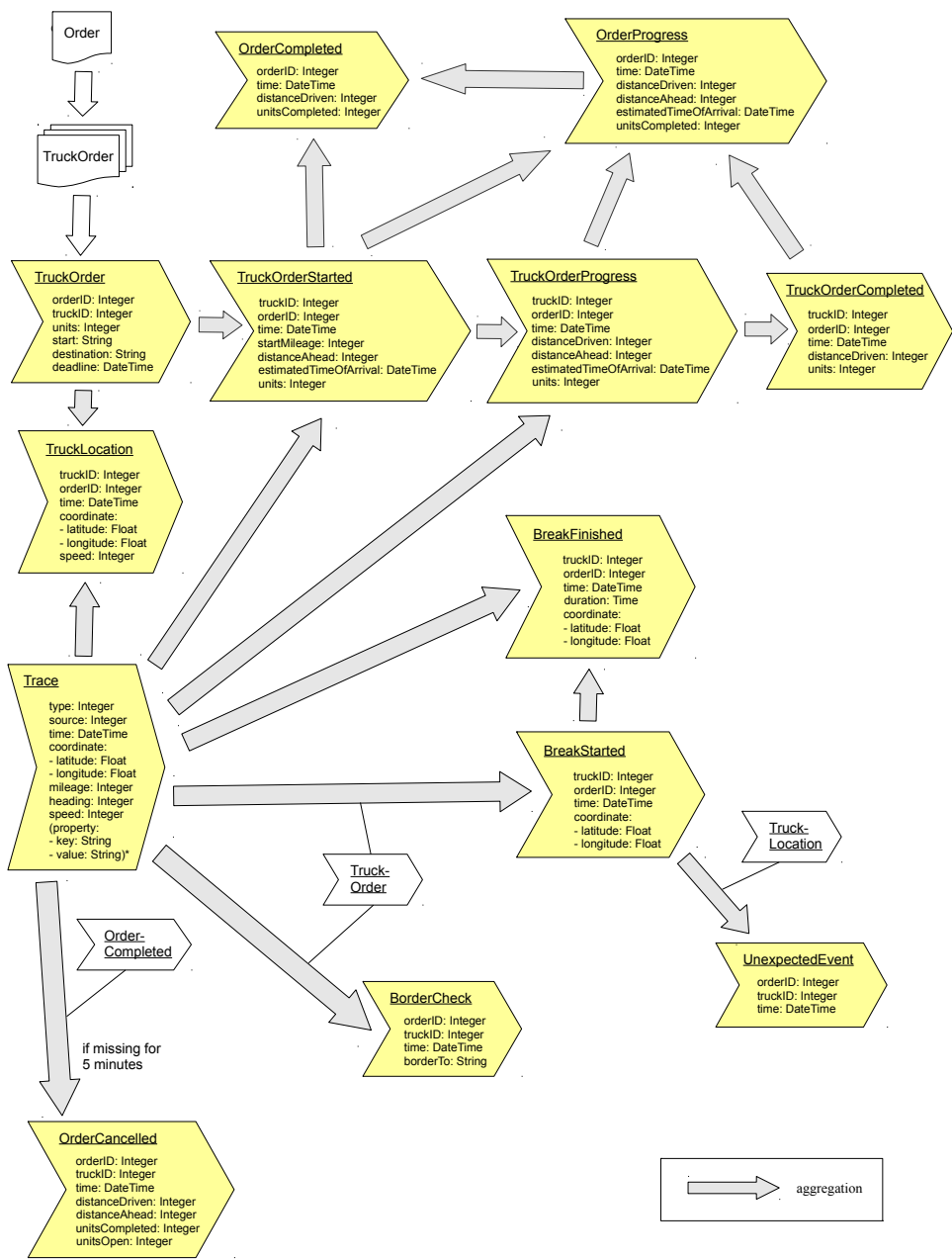


Figure 5: Event Hierarchy

## References

- [1] Baumgrass et al. *GET Service: Deliverable D6.3 – Prototypical Implementation of the Information Aggregation Engine*. 2014.
- [2] O. Etzion and P. Niblett. *Event Processing in Action*. 1st. Greenwich, CT, USA: Manning Publications Co., 2010.
- [3] GET Service. Website. Available at <http://getservice-project.eu/>, last accessed September 2015. 2014.
- [4] R. LeFaive, S. Sodem, J. Scarpelli, D. Ketcham, and A. Garg. *Method and system to correlate a specific alarm to one or more events to identify a possible cause of the alarm*. US Patent 7,131,037. Oct. 2006.
- [5] Meteor Development Group. Website. Available at <https://www.meteor.com/>, last accessed September 2015. 2015.
- [6] MongoDB Inc. Website. Available at <http://www.mongodb.org/>, last accessed September 2015. 2015.
- [7] H. Motahari-Nezhad, R. Saint-Paul, F. Casati, and B. Benatallah. “Event correlation for process discovery from web service interaction logs”. English. In: *The VLDB Journal* 20.3 (2011), pages 417–444. DOI: 10.1007/s00778-010-0203-9.
- [8] H. Njemanze and P. Kothari. *Real time monitoring and analysis of events from multiple network security devices*. US Patent 7,376,969. May 2008.
- [9] OMG. *Business Process Model and Notation, Version 2.0*. Website. Available at <http://www.omg.org/spec/BPMN/2.0/>, last accessed September 2015. 2011.
- [10] J. Rivera and R. van der Meulen. *Press Release*. Available at <http://www.gartner.com/newsroom/id/2575515>, last accessed September 2015. Aug. 2013.
- [11] Treitl et al. *GET Service: Deliverable D1.1 – Use Cases, Success Criteria and Usage Scenarios*. 2014.
- [12] Trimble - Transport & Logistics. *Fleet Integrator Guide, Version 1.14, Revision 172*. 2012.
- [13] UNICORN Platform. Website. Available at <http://bpt.hpi.uni-potsdam.de/UNICORN>, last accessed September 2015. 2015.
- [14] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. second. Springer, 2012.





# DRAGON – Deadline Propagation Transcending the Boundaries of Processes

Michelle Mensing, Jakob Reschke, and Tim Sportleder

Hasso-Plattner-Institut

{michelle.mensing, jakob.reschke, tim.sportleder}@student.hpi.uni-potsdam.de

Practical business processes are often associated with deadlines. In a logistics context these deadlines are subject to frequent changes. While there is work on the prediction of the execution time of individual activities or single process instances, research on the effect of a deadline shift on multiple activities and interdependent processes is still lacking. Event processing can be used to communicate relevant influencing factors for activities as soon as they are detected. This also applies to deadline changes. We describe a demo software, Dragon, which implements a flexible deadline propagation algorithm to predict imminent deadline violations caused by a deadline shift across multiple process instances. It also features a graphical user interface which provides multiple views on the scheduled activities to visualize the effects of a deadline shift.

## 1 Introduction

Business processes and activities often have time constraints such as deadlines. If a deadline of an activity is changed, consequences may arise for the preceding activities. For a single process the propagation of such a change is easy to handle. In practice, processes are interdependent due to shared resources or because of the personnel involved. Therefore, the consequences of a deadline shift are hard to predict and may result in deadline violations because of undetected dependencies.

In this paper we present Dragon, a system that detects likely deadline violations before they occur using event processing technologies. To realise Dragon, we assigned deadlines and execution times to activities, implemented a propagation algorithm and a visualization of the consequences of a propagated deadline shift. Based on real processes discovered in the context of the GET Service project<sup>1</sup>, we describe how a single deadline change influences multiple process instances. Dragon is able to

---

<sup>1</sup> GET Service: Efficient Transportation Planning and Execution.  
<http://www.getservice-project.eu>, last accessed September 2015.

react to deadline shifts by using the event processing platform UNICORN<sup>2</sup> for the detection of a deadline change and notification of likely deadline violations.

Due to the lack of BPMN 2.0 [3] in explicit modeling of deadlines and execution times for activities, existing papers focus on introducing time constraints to activities [1] and predicting their execution times [4, 5]. The propagation of a deadline shift can make use of both. Dragon uses the execution times of activities and the inner-process and inter-process (through personnel disposition) relations between activities to realize the propagation.

In Section 2, we introduce our use case by describing a real world scenario in which a deadline shift occurs. Section 3 presents the data model and architecture of Dragon and how it reacts to deadline change events. The algorithm which propagates a deadline change is presented in Section 4. Section 5 deals with the graphical user interface of Dragon. Finally, Section 6 summarizes our results, discusses conceivable extensions of our approach and gives an outlook for further use.

## 2 Use Case

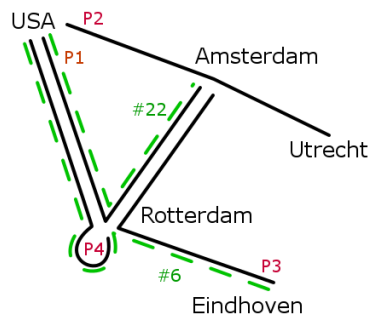
A planner of a logistic company has to deal with and plan multiple customer orders. For each order exists a predefined process model and the planner assigns operators to the activities defined in the process model. To meet specific loading and unloading time windows for goods, start and end deadlines are specified for activities. However, deadline shifts may occur due to deliberate replanning or replanning because of unforeseen incidents. This can possibly lead to a reassignment of the operators. Because an operator's schedule is not constrained to a single process instance a deadline shift can affect multiple instances.

In our scenario there are four orders with their related processes as shown in Figure 1. Goods are delivered from the USA via Rotterdam to Amsterdam in process 1 (P1), from the USA via Amsterdam to Utrecht in process 2 (P2) and from Eindhoven via Rotterdam to Amsterdam in process 3 (P3). In this example we assume that the goods of P1 and P3 are delivered by the same truck between Rotterdam and Amsterdam. Finally, process 4 (P4) includes the customization of goods at an external terminal in Rotterdam and the subsequent shipping to the USA.

Several operators are assigned to these processes. For example operator #22 ships the goods for P1 from the USA to Rotterdam, loads the goods and those of P3 on a truck and drives from Rotterdam to Amsterdam. Furthermore, operator #6 transports

---

<sup>2</sup> <http://bpt.hpi.uni-potsdam.de/UNICORN>, last accessed September 2015.



**Figure 1:** Overview of our logistics scenario with processes (solid lines) and selected operator assignments (dashed lines)

the goods of P3 from Eindhoven to Rotterdam, unloads them and proceeds with the execution of P4.

Due to an earlier incoming ship of P1 at the port of Rotterdam the start deadline for loading the goods of P1 onto a truck is shifted forward. As a result all preceding activities of each process and operator involved may be affected. Here, operator #22 has to deliver the goods of P3 earlier so that the storing time can be minimized.

### 3 Data Model and Architecture

Dragon is based on a simplified process model and distinguishes between model- and runtime of a process. In modeltime each process in the data model is defined by a set of activities which reference other activities by predecessor or successor relationships. Each activity includes a description, relative start- and end deadlines and a default duration. In runtime a process instance is created and every activity is represented by at least one *Task*. The assignment of an operator to a task is modeled by an *OperatorTask*. Tasks and OperatorTasks have predecessors and successors just like activities. All OperatorTasks which are performed by a particular operator constitute the schedule of the operator which can thus relate to multiple process instances.

Dragon is implemented as a Spring Boot<sup>3</sup> application consisting of the following components: A graphical user interface (GUI), an event replayer, an event publisher, a deadline propagator and an in-memory database containing the data (see Figure 2). For retrieving and sending events we use the event processing platform UNICORN,

<sup>3</sup> <http://projects.spring.io/spring-boot/>, last accessed September 2015.

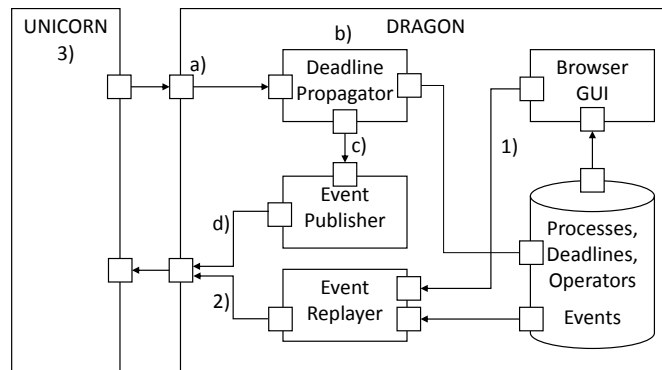


Figure 2: Architecture of Dragon

from which events can be received via publish-subscribe and which offers a webservice interface to publish new events and event types.

We defined an event hierarchy consisting of four event types. The DeadlineShift event type (deadline shift for a single task), the DeadlineReplanning event type (notification of a change in task planning parameters affecting deadline propagation) and the DeadlineChange event type which serves as a generalization of the first two event types. To make sure that DeadlineShift and DeadlineReplanning events will also cause a DeadlineChange event to be published, we register an aggregation rule in UNICORN. This way Dragon only needs to subscribe to DeadlineChange events. The fourth event type is the DeadlineHazardNotification type for communicating the result of the deadline propagation for a single task.

When Dragon receives a DeadlineChange event from UNICORN (a) the deadline propagator starts calculating the propagation of the received deadline shift (b) (see Section 4). After the calculation is finished the results are presented in the GUI (see Section 5) and are published by the event publisher as DeadlineHazardNotification events (c, d) for every affected task. For the demo, we prepared a DeadlineShift event that is replayed to UNICORN when triggered from the GUI (1, 2). Our aggregation rules will transform that event into a DeadlineChange event (3) which is then again received by Dragon (a).

## 4 Propagation algorithm

We focus on deadlines which have been shifted to an earlier instant and propagating that change to preceding tasks. The propagation starts from the task which is named

in the DeadlineChange event by its database ID. An example event can be found in Listing 2 in the appendix. This task will be called the *origin* of the deadline propagation. Starting from the origin task, the goal is to verify that its deadlines and those of its preceding tasks can still be met. A high-level version of our algorithm can be seen in Algorithm 1. First, the execution time of a task is predicted, starting from the currently planned starting time (line 1). For each task the predicted execution interval must start before the start deadline and end before the end deadline, otherwise there is an imminent deadline violation. In case of a violation the start time of the task is modified to meet the deadline again (line 3). New overlaps with preceding tasks must be detected and handled by the propagation.

To control the extent of the propagation we assess how probable the violation of a deadline for a given task is (line 2). When a task will finish after the deadline, it is certain to violate that deadline. If it finishes a small amount of time before the deadline, a deadline violation might still be possible if certain other hindering events occur, e.g. traffic congestion. In both cases a planner wants to be informed so the task can be replanned. The outcome of the risk assessment and replanning is communicated to the environment on line 4.

We introduce an *uncertainty level* into the propagation which is initially zero and increased for a propagation path whenever a task with an unviolated but tight deadline is encountered (line 5). Everytime the uncertainty level is increased, the subsequently detected violations become more speculative. Thus, if the uncertainty level reaches a certain threshold, the propagation will stop (line 8).

When a task which has already been completed is encountered, the propagation will also not pursue it and its predecessors (line 12).

The propagation traverses all relevant tasks by the recursion on line 14. For this to work correctly and comprehensively, the preceding tasks (line 11) are derived from the task's predecessors and from its OperatorTasks' predecessors. This is shown in listing 1 as an OCL<sup>4</sup> constraint.

#### Listing 1: Obtaining the predecessors of a task for propagation in Dragon

```
context Task::propagationPredecessors: Set(Task)
derive: predecessors->union(
    containedOperatorTasks->collect(predecessors)
    ->flatten()->collect(correspondingTask)->asSet())
```

<sup>4</sup> Object Constraint Language – <http://www.omg.org/spec/OCL/>, last accessed September 2015.

**Algorithm 1** Deadline propagation outline

---

```

Require: originTask : Task, uncertaintyLevel : Integer,
           uncertaintyThreshold : Integer
           duration  $\leftarrow$  predictDurationOf(originTask)
2: risk  $\leftarrow$  assessDeadlineViolationRiskFor(originTask, duration)
   replanExecutionTimesFor(originTask, duration, risk)
4: emitNotification(originTask, duration, risk)
   if risk < CERTAIN then
6:   uncertaintyLevel  $\leftarrow$  uncertaintyLevel + 1
   end if
8: if risk = UNLIKELY or uncertaintyLevel  $\geq$  uncertaintyThreshold then
   return {stop propagation}
10: end if
   for all precedingTask  $\in$  originTask.propagationPredecessors do
12:   if not precedingTask.isFinished then
       precedingTask.endDeadline  $\leftarrow$  originTask.startTime
14:   propagateDeadlineChange(precedingTask, uncertaintyLevel) {recursion}
   end if
16: end for

```

---

In Algorithm 1 there are four high-level calls marking variation points enumerated below. We implemented these with the Strategy Pattern [2, p. 267] so different implementations can be plugged in by reconfiguring the application. These variation points are: 1. The prediction of a task’s duration (line 1), 2. The assessment of a deadline violation risk (line 2), 3. The replanning of a task to avoid a violation (line 3) and 4. The notification about a task and its deadline violation risk (line 4).

In our proof-of-concept implementation Dragon, the duration is taken from a *defaultDuration* attribute of the Activity in our database. A deadline violation is considered “possible” (between “certain” and “unlikely”) if there is less than a configurable amount of time, like one hour, between the predicted end time and the end deadline. If there is more time, a violation is deemed unlikely. A task is replanned such that the predicted end coincides with the end deadline. The notification is implemented by adding a record for it in the database so it can be used by the GUI (see Section 5) and by calling the event publisher to publish DeadlineHazardNotification events via UNICORN (see Figure 2 c, d).

An important optimization of the algorithm as outlined above is to not traverse the same path of tasks twice without changing execution times any more. This issue arises when a task has multiple predecessors which have a common ancestor task. The propagation could traverse to and beyond the common ancestor from the first predecessor of the origin task. Eventually the propagation would return to the second predecessor and maybe reach the common ancestor from there again. If the risk assessment and replanning operations decide that no further modification to the common ancestor’s execution times is necessary, the propagation should not proceed again with the predecessor tasks of the common ancestor. However, if the replanning

strategy decides to change the execution times again, the propagation cannot be pruned and must traverse the predecessors again until no further adjustments are made.

## 5 Application

We provide a web-based GUI to visualize the effects of deadline shifts. When the Dragon webpage is loaded, data on process instances and operators is loaded asynchronously from the server. Using AngularJS<sup>5</sup>, the GUI reflects the data via double data binding. This means, changes in the data cause corresponding changes in the GUI and vice versa. In order to get information on deadline shifts, the server-side of Dragon is continuously polled for deadline hazard notifications stored in the database.

Data objects are serialized via JSON and stored as JavaScript objects. We faced two problems which arise due to circular references among the objects. On the server side, the serialization failed due to infinite recursion. On the client side, the data binding with AngularJS failed for the same reason. Therefore, we had to transform the data graphs by replacing some references with object identifiers.

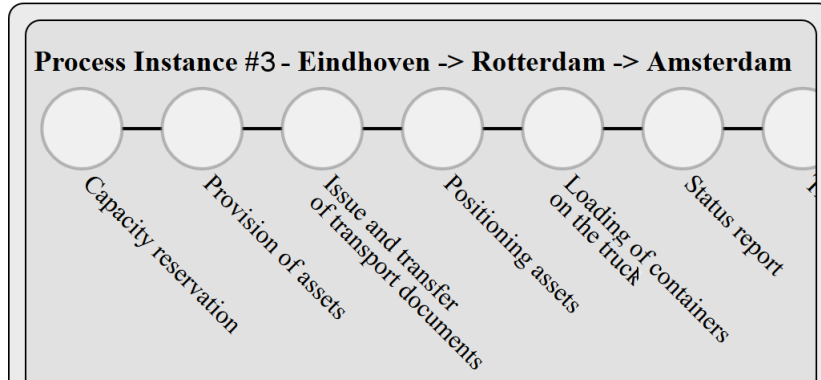
The GUI features an instance view and an operator view. The instance view focuses on process instances. From the list of process instances P1 to P4 introduced in section 2, any may be chosen to be visualized as a graph. That graph depicts the tasks of that instance as circles with descriptive labels. All graphs are rendered using Scalable Vector Graphics (SVG) which is embedded in the Document Object Model (DOM) of the web page, allowing for easy manipulation via CSS and JavaScript. Figure 3 shows a part of the graph of an instance of P3. If a task is selected by clicking on the corresponding node, an overlay shows information on that task, including time constraints like the planned start and end time as well as the operators who take part in the task.

Operators can be inspected via the operator view in an analogous manner. The OperatorTasks in an operator's schedule can have several predecessors and successors as an operator may transport wares from several tasks at once. Information on a selected OperatorTask include time constraints from the corresponding task and the related process instance.

For the purpose of demonstration, our GUI provides a button to evoke the necessary event processing (see Figure 2, steps 1 and 2). When the deadline hazard

---

<sup>5</sup> <https://angularjs.org/>, last accessed September 2015.



**Figure 3:** Part of the graph of the P<sub>3</sub> instance. Each node represents a task.

notifications are returned by the server, a warning message is shown. For each notification, the stored data is traversed to find the respective process instance and task using the identifiers included in the notification. The violation probability for all affected objects, including the operators of the task, are updated. Listing 3 (in the appendix) shows a serialized notification with a certain violation, new time constraints and a reference to the respective task and process instance.

There are three degrees of severity of deadline hazards based on the uncertainty level introduced in Section 4 and each represented in the GUI by a color. Violations with uncertainty level zero are certain and colored red. If the uncertainty level reached the threshold, the violation is unlikely and colored green. All other violations are deemed possible and colored yellow. These colors are used to mark affected process instances and operators in each list as well as tasks and OperatorTasks in each graph. For instance, Figure 4 (in the appendix) shows details on an OperatorTask of the origin task affected by our demo DeadlineShift event.

## 6 Conclusion and Outlook

In this paper we described an algorithm for the propagation of event based deadlines changes to detect imminent deadline violations across multiple process instances. Our demo implementation, Dragon, interacts with UNICORN to simulate deadline changes and be notified of them. The results of the propagation are displayed in a graph of tasks which are highlighted with colors to indicate deadline violations.

Our algorithm includes four variation points where different behavior suitable to the application can be plugged in. In a real logistics planning system it would be advisable to employ more advanced implementations compared to those in Dragon. For example, the prediction of activity durations could make use of the approaches



presented in [4, 5]. Also, other event processing applications such as awareness to weather forecasts and delay propagation could be handy.

Dragon's GUI could conceivably support more views to allow for more specialized insight in the effects of a deadline change, e.g. a graph showing all tasks affected by a deadline change regardless of operators and process instances. However, it would become increasingly difficult to arrange the resulting graph in an easily comprehensible manner.

We focused on deadlines which have been shifted to an earlier instant. Future work could investigate applications for the propagation of postponed deadlines and a propagation from the origin task into the future. While that should not reveal deadline violations which are caused by the deadline change, both approaches would probably be able to predict optimizable gaps in activity and operator schedules and indicate opportunities for replanning.

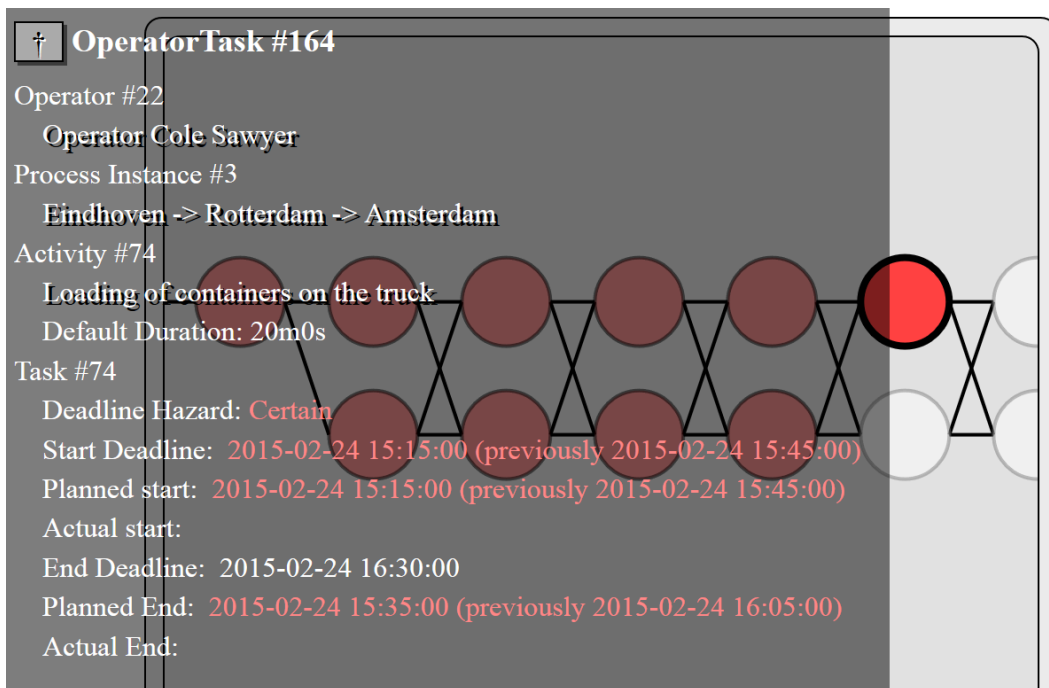
## Appendix

Listing 2: A DeadlineChange event serialized as JSON

```
{
  "timestamp": "2015-02-26T14:08:05.000",
  "taskId": "74",
  "oldDeadline": "2015-03-01T15:00:00",
  "newDeadline": "2015-03-01T13:00:00",
  "deadlineType": "start",
  "operatorId": "22"
}
```

Listing 3: A deadline hazard notification serialized as JSON

```
{
  "violationProbability": 2, // 2 = certain
  "createdAt": 1424420516102,
  "task": {
    "processInstance": {
      "id": 4
    },
    "id": 74
  },
  "newStartDeadline": 1424441700019,
  "newPlannedStartTime": 1424441700019,
  "newPlannedEndTime": 1424442900019,
  "id": 1,
  "newEndDeadline": 1424446200405
}
```



**Figure 4:** Part of the graph of operator Cole Sawyer. The selected OperatorTask faces a certain deadline violation with changed temporal constraints, marked with red

## References

- [1] S. Cheikhrouhou, S. Kallel, N. Guermouche, and M. Jmaiel. "Toward a Time-centric modeling of Business Processes in BPMN 2.0". In: *Information Integration and Web-based Applications & Services*. ACM. 2013, page 154.
- [2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.
- [3] OMG. *Business Process Model and Notation (BPMN), Version 2.0*. OMG Standard. Jan. 2011.
- [4] M. Polato, A. Sperduti, A. Burattin, and M. de Leoni. "Data-aware remaining time prediction of business process instances". In: *Neural Networks (IJCNN)*. IEEE. 2014, pages 816–823.
- [5] A. Rogge-Solti and M. Weske. "Prediction of remaining service execution time using stochastic petri nets with arbitrary firing delays". In: *Service-Oriented Computing*. Springer, 2013, pages 389–403.



# Non-linear Delay Propagation of Event Based Business Processes

Heiko Beck, Maximilian Brehm, and Marius Eichenberg

Hasso-Plattner-Institut

{Heiko.Beck, Maximilian.Brehm, Marius.Eichenberg}@student.hpi.uni-potsdam.de

Workflows in an enterprise are usually determined by business processes and corresponding process models. To manage these processes, workflow management systems are used. For certain processes, there is a need to precisely model resource- and status-dependencies and resulting time-dependencies of its interlinked process steps. The progression of delays caused at an initial process step to the subsequent process steps may be non-linear due to these dependencies and additional time-dependent resource- and status-constraints of the process steps. This is especially true when modelling and monitoring complex real-world workflows. In this paper, we propose a method for non-linear delay propagation considering resource- and status-dependencies and -constraints of sequentially interlinked process steps, and a visualization for live monitoring process progression and delay propagation of processes. As a running example, we consider processes from the field of logistics.

## 1 Introduction

Workflows in an enterprise are usually determined by business processes and corresponding process models. To manage these processes, workflow management systems are used [3]. These systems are seldom using external information sources to calculate dependencies, like resource- or status-dependencies, among related process steps. Especially in the field of event based business processes there is a high need of additional data to model the real world as good as possible. Also live monitoring of a running process is required. In this paper we focus on delay propagation, as one case of interaction between process steps with the assistance of information from external resources. We also take resulting deadline shifts and live monitoring into account.

As a running example, we use the delay propagation in the field of logistics. In particular, a scenario of cargo transport from London to Frankfurt is considered. We split the process into four steps, beginning with a flight from London to Amsterdam, followed by a loading activity to shift the cargo onto a truck. The truck then brings the freight to Frankfurt and in the last step the cargo is unloaded at a

plant in Frankfurt. The transport is compromised by unexpected events, like severe weather or traffic jam, as the reason for process delays. We also take additional information from external resources, so called constraints, into account. These status- and resource-based constraints are necessary to model real world characteristics like traffic density or loading capacities of a plant according to different start times.

In the paper we show that calculating dependencies, such as delays, among related process steps is not trivial, if one takes this additional information into account. The consideration of these status- and resource-dependencies may result in non-linear delay propagation.

The remainder of this paper is structured as follows: Section 2 introduces the foundations for our approach. Section 3 explains the non-linear delay propagation while Section 4 shows our implementation. Section 5 reviews related work and Section 6 gives a conclusion.

## **2 Foundations**

The general concept we are dealing with are processes. Each process consists of smaller units called process steps. In this paper, we consider only sequential steps. This means that it is not possible to start a new process step if its predecessor is not finished. So the time an executing process needs to finish, becomes a main part of this general concept.

The progress of processes is determined by events, whereas an event is something that has happened or is contemplated as having happened [1]. The handling of such events is called event processing. While handling these events, they can be transformed, combined or split into many other events. These aggregations are modeled with regulations, called aggregation rules. The result of such a aggregation can be one or more new events.

A special kind of event is the unexpected event, which is something that happened unplanned and was not predictable. Unexpected events affect the flow of the process and will result in a recalculation of the process time. This can be either in a positive or negative way. During the recalculation, additional external information, so called constraints are taken into account. As an event happens at a certain point in time, a constraint describes the capacity of one or more resources at this time. Therefore, the capacity of resources can affect the process time, too.

### 3 Delay Propagation for sequential Processes

In this chapter we will introduce our general approach for delay propagation for business processes. A naive approach for delay propagation would be to sum up the delays of the individual steps and use the result as delay for the whole process instance. Such an approach is not sufficient to model real world processes because these can depend on different constraints as mentioned in the previous chapter. To model these types of real world processes, we developed a non-linear delay propagation. In the first section we will first explain the basic method of delay propagation. Then, in the second section we will make use of event processing in order to realize our approach.

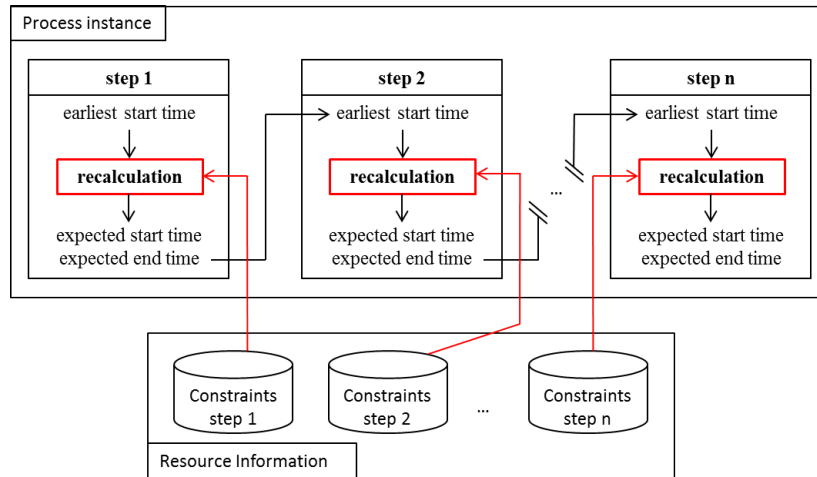
#### 3.1 Fundamental principle

In order to calculate delay propagation, we expanded all process steps by three attributes:

1. *earliest start time*: Specifies the earliest time, that a process may start at. Since we are considering only sequential steps, the earliest start time is equal to the end time of the previous step.
2. *expected start time*: Specifies when a process step actually starts. In contrast to the earliest start time, the expected start time takes constraints into account.
3. *expected end time*: Specifies when a process step will be finished. The expected end time depends on the expected start time, constraints and unexpected events. The duration of a step results from the difference between expected start and end time.

Based on this process specification, we can introduce our approach which consists generally of two parts: a recalculation part and a propagation part. Whenever a process step is expected to be influenced by an unexpected event, a recalculation is triggered. The aim of the recalculation is to update the expected start and end time of the process step. If the step is already running, only its expected end time will be updated. If a recalculation is finished, the new expected end time is propagated as earliest start time to the following step. This principle of recalculation and propagation is carried out until the entire process chain is updated.

The non-linearity of our approach is part of the recalculation. This is performed using external information from different data sources that can be time dependent. The result is therefore also time-dependent. It is possible that a step cannot be started at its earliest start time due to a constraint or that the duration of the step differs



**Figure 1:** Visualization of the recalculation and propagation principle

dependent on the start time. A premise for our recalculation is that all constraints for a process step were previously defined and that the information about these constraints is machine-readable.

In Figure 1, our approach is visualized. The process instance consists of an arbitrary number of sequential steps. Every step has an earliest start time, an expected start time and an expected end time. The expected times of a step are the result of a recalculation. The recalculation uses the earliest start time and information about constraints as input. For each step, the various constraints which may influence one step depend on the type of the step. The expected end time of one step flows as earliest start time into the following step and triggers another recalculation.

### 3.2 The use of event processing

Our approach is based entirely on event processing. We regard the previously defined process times as event types. Additionally we use aggregation rules to recalculate these times as well as for the propagation of times. Therefore we distinguish between three kinds of aggregation rules:

1. *unexpected event rule*: Whenever an unexpected event impacts a step, the expected times of the particular step are updated. The result of such a rule is a new expected start and a new expected end time event. If the affected step is already running, no new expected start time event is created.



2. *propagation rule*: Whenever the expected end time of a step is updated, a propagation rule creates a new earliest start time event for the following step.
3. *calculation rule*: Whenever a new earliest start time event is received by a process step, a recalculation rule updates the expected start and end time for the particular step dependent on the earliest start time and process dependent constraints.

To perform the delay propagation for a specific process instance, some context-sensitive knowledge about step-sequences and available constraints is required. Once an unexpected event rule updated a process step, the propagation rule must know the succeeding step of the affected process instance to propagate a new earliest start time. We use a unique ID for every transport step and a successor-function: The unique ID of a process step is passed as parameter to the function that then returns the ID of the subsequent step. Also, the calculation rule must include a function call to calculate the process times considering all constraints. This function gets the unique ID and the earliest start time as parameter and returns the updated expected start and end time for the particular step.

Thus, our approach is fully described. We map unexpected events to process steps and propagate the expected end time of one step as earliest start time to the following step. Whenever a new earliest start time for a process step is available, the expected start and end time for this step is recalculated using an event aggregation rule and function calls. How this concept can be implemented is content of the next section. The implementation is part of the next section.

## 4 Implementation

As a proof-of-concept, the *Noldep* (short for *Nonlinear delay propagation*) application for process monitoring in the context of logistics was developed. The application features delay propagation as an reaction to unexpected events that may emerge during the progress of an active process. The core application is written in Java and a web interface is based on web standards.

*Noldep* interacts with the *UNICORN platform*<sup>1</sup> for process monitoring and process calculation. *UNICORN* is a service for transportation planning and monitoring developed by the *Business Process Technology Group*<sup>2</sup> at the *Hasso Plattner Institute* as part of an European Research program. The service connects various transportation man-

<sup>1</sup> <http://bpt.hpi.uni-potsdam.de/UNICORN>, last accessed September 2015.

<sup>2</sup> <http://bpt.hpi.uni-potsdam.de/Public/>, last accessed September 2015.

agement systems from various transportation partners, logistics service providers and authorities and enables the exchange of selected information between these partners. The service is based on *Esper*<sup>3</sup>, an Event Processing Platform for Complex Event Processing and Event Series Analysis developed by EsperTech.

### Logistic chain

Noldep provides mechanisms for the specification of a sequential logistic chain as a process model to model real-world logistics scenarios. In the application, a logistic chain is specified as a list of the process steps *flight*, *truck*, *loading* and *unloading* that are to be processed in sequential order. We apply the principles of Section 3 to our process model; thus each of the process steps has a *unique identifier* and the attributes *earliest start time*, *expected start time*, and *expected end time* to implement delay propagation.

### Architecture

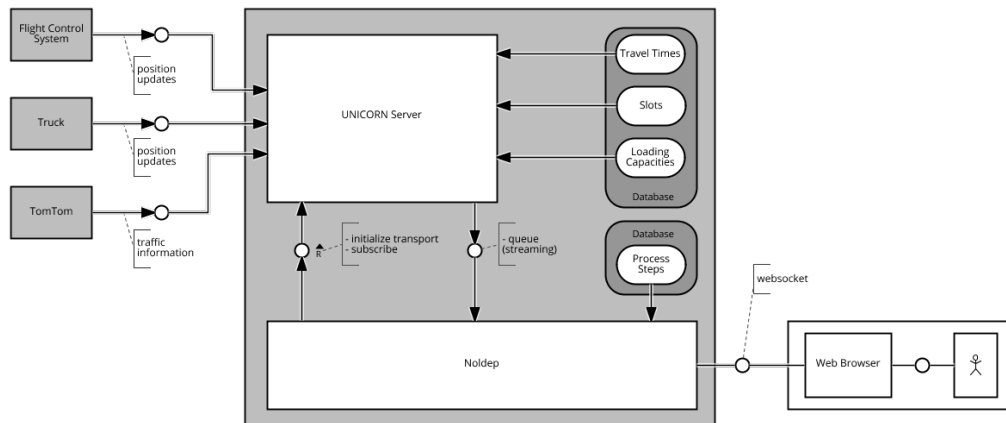
In Figure 2, the architecture of Noldep and its interaction with UNICORN is modeled in a system architecture diagram using Fundamental Modeling Concepts<sup>4</sup>. Noldep derives the specifications of the logistic chains that are to be modeled from a local database. For each of the process steps of a process, it registers relevant progress event types and select queries that are used for process monitoring with the UNICORN service. It also registers relevant delay event types and aggregation rules that are used for the delay calculation and propagation with the service. As the type of progress events (e.g., progression by truck, train or plane), unexpected events (e.g., severe storm or congestion) and reaction to these types (e.g., process halt, circumnavigation) is specific to a process step, this registration process is executed for each process step of a process individually.

After this initial phase, UNICORN is then ready to monitor and process the progress and delay events of each of the registered processes. The platform receives any of these events through external logistic services and service providers. In our case, we connect queues for plane status updates, truck status updates and traffic status updates dependent on the process specification. The first two provide detailed status updates from transportation services while the latter returns road congestions. The UNICORN platform may access various databases for the estimation of process durations that are specified in the event aggregation rules registered by Noldep. In our application, these are databases for transport capacities of logistics services,

---

<sup>3</sup> <http://www.espertech.com/products/esper.php>, last accessed September 2015.

<sup>4</sup> <http://fmc-modeling.org/>, last accessed September 2015.



**Figure 2:** System Architecture Diagram (FMC) of the Noldep application and interaction with external services

estimation of travel times for flight and land transport services, and loading and unloading slots of cargo exchange services.

Noldep receives any events for its processes through streaming queues established with UNICORN based on the select queries that were registered during the initial registration of the process models. The core application of Noldep then pipes the monitoring information of these queues to the Noldep web server. This web server was set up at the start of the application and allows for local or remote interactions with Noldep via a web browser.

### Event Processing

In this section, we describe the Noldep implementation of the delay propagation in Esper as motivated in Section 3. Noldep registers the event types *EarliestStartTime*, *ExpectedStartTime*, *ExpectedEndTime* and *ExpectedProcessTimes* with the UNICORN platform. These types share the attribute *timestamp* that symbolizes the time of occurrence of event instances and *operatorId* that relates instances with their respective process step. An instance of the first three types has either one attribute *startTime* or *endTime* based on the type while the latter has both. These attributes stand for new estimates of the earliest or expected times of the respective step. However, only the first three update the respective step in our application; the latter is the result of the process time calculation based on *EarliestStartTime* and triggers the events *ExpectedStartTime* and *ExpectedEndTime*.

In Algorithm 2, we see the respective Esper query that creates the event *ExpectedProcessTimes* and implements the calculation rule as mentioned in Section 3. We

make use the Esper feature *Joining Methods*<sup>5</sup> that allows the creation of join data in the *from* clause via external Java calls. In this case, *updateProcessTimes* returns a single event *ExpectedProcessTimes* based on the attributes *operatorId* and *startTime* of the event *EarliestStartTime*. The function *updateProcessTime* queries external data bases or web services based on its parameters to return an updated estimation of the expected times of the affected step.

As mentioned before, *ExpectedProcessTimes* then triggers the creation of *ExpectedStartTime* and *ExpectedEndTime* that are used to update the expected times of a step in *Noldep*. *ExpectedEndTime* triggers an event *EarliestStartTime* of the successive step or none, if the step is the last in the sequence. To determine the identifier of the successive step, we use the Esper feature *User-Defined Functions (UDF)*<sup>6</sup>. UDFs may be used to return single Java Objects in the *select* and *where* clauses of the queries. In this case, the method is a callback to our *Noldep* application that returns the identifier of the successive process. Thus, these aggregation rules implement the sequence that is used for non-linear propagation delay propagation as mentioned in Section 3.

---

**Algorithm 2** Creates *ExpectedProcessTime* based on the *EarliestStartTimeEvent* of the same process step

---

```
select tr.operatorId as operatorId,
times.startTime as startTime,
times.endTime as endTime,
currentDate() as timestamp
from
EarliestStartTime.std:lastevent() as tr,
method:de.uni_potsdam.hpi.esper.Processes.updateProcessTimes(
tr.operatorId, tr.timestamp) as times
```

---

In order to trigger the sequence, a process step needs to be affected by an unexpected event or constraint that results in a delayed end time of the process step. In Algorithm 3, a query is listed that creates an event *ExpectedEndTime* as a result

---

<sup>5</sup> [http://esper.codehaus.org/esper-5.0.0/doc/reference/en-US/html/epl\\_clauses.html#joining\\_method](http://esper.codehaus.org/esper-5.0.0/doc/reference/en-US/html/epl_clauses.html#joining_method), last accessed September 2015.

<sup>6</sup> [http://esper.codehaus.org/esper-5.1.0/doc/reference/en-US/html\\_single/index.html#epl-function-user-defined](http://esper.codehaus.org/esper-5.1.0/doc/reference/en-US/html_single/index.html#epl-function-user-defined), last accessed September 2015.

of a traffic congestion. The event `CongestionAhead` is provided by the UNICORN platform as a result of a congestion event returned by the external queue for traffic status updates. We calculate the new expected end time based on the last event `ExpectedEndTime` and the expected delay that is caused by the congestion. We use an UDF to cast a date representation of type `Java Long` to `Java Data`. As mentioned previously, these unexpected events are specific to a process step of a process and need to be defined and registered for each of the steps individually.

---

**Algorithm 3** Creates `ExpectedEndTime` as a result of the traffic update `CongestionAhead`

---

```

select t.operatorId as operatorId,
       de.uni_potsdam.hpi.esper.Utills.getDate(
         t.timestamp.getTime() + c.predictedDelay) as timestamp
from pattern
[every t=ExpectedEndTime ->
  c=CongestionAhead(operatorId=t.operatorId)
  and not ExpectedEndTime(operatorId=t.operatorId)]

```

---

### Interface

In Figure 3, the main view of the Noldep web interface is provided that displays the process progression and status of a specific process. In this view, a line block graph is displayed to visualize the current status of the selected process, its history and its future. The sequential process step of the selected process are displayed from bottom to top on the y-axis of the graph. For each process step, the current expected start and end times are visualized by the start and end of the blue and green blocks that are on the same y-level as the process step. The blue blocks of a step indicate the sequence that was originally expected for the process step. The green blocks visualize additional sequences that were introduced as an effect of delay propagation. The red blocks visualize additional sequences that resulted from unexpected events. At last, the grey bars in the background of the chart visualize the originally expected process progression, that is, before any unexpected events and delay propagation.

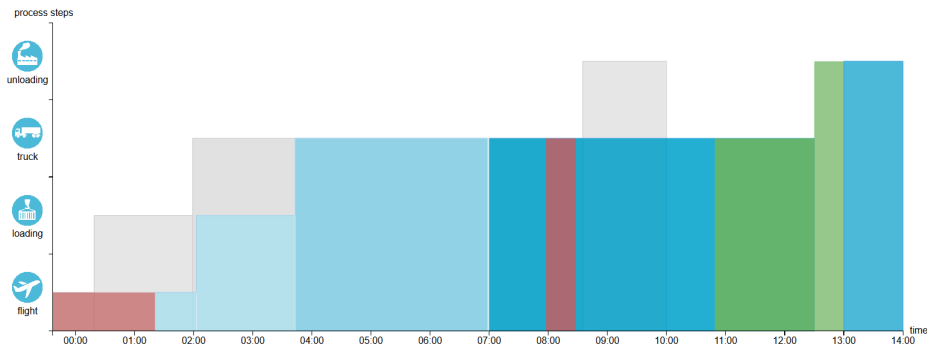


Figure 3: The main view of the Noldep web interface

## 5 Related Work

With the growing number of Process Aware Information Systems, the possibilities for time predictions increase. Different approaches like [2, 4] take advantage of historical data from event logs to predict the remaining time for a running case. They model processes by transition systems and annotate all states of the transition system with time information from former process instances. For a running case, all information from events are collected and the transition system is used to predict the remaining time.

An approach presented in [3] uses the absence of events to predict the remaining time of a running case. Therefore the process is modeled with a petri net instead of an annotated transition system as the basis for time prediction. They not only use the information from events for time prediction but also the passed time since the last event occurred. So expected but not yet happened events can also influence the predicted time.

The overall goal of all described approaches is to predict the time of a running process instances correctly. With our approach, we had the same goal but with a focus on the propagation of predicted times through a sequential process. Instead of historical data from previous instances, we use production data from all stakeholder of the process to calculate the remaining time context-sensitively.

## 6 Conclusion

The main purpose of this paper was the enrichment of event based business processes with additional data and the delay propagation, as one case of interaction

between process steps. We proved that it is not sufficient to only consider a linear propagation of delays and that there is a need to propagate delays in a non-linear way. We introduced a new concept which deals with two different start times combined with a depending end time and an enrichment with external information to calculate and propagate delays. With the implementation, we have shown that we are able to transfer the theoretical concept into a practical application use-case.

As our system is based on object oriented programming and relational database models, it is limited by the events and external information that were implemented and registered a priori. We are not able to handle unknown and unregistered events.

To enhance our concept of delay propagation, parallelism of processes should be taken into account. It should be possible that a process step has more than one predecessor it depends on. With this it should be possible to model even more complex and more real world scenarios.

## References

- [1] A. Baumgraß. *Event processing in GET Service*. English. Available at <https://www.youtube.com/watch?v=bIVyr0rs9CE>, 0:44 minute, last accessed September 2015.
- [2] M. Polato, A. Sperduti, A. Burattin, and M. de Leoni. "Data-aware remaining time prediction of business process instances". In: *Neural Networks (IJCNN), 2014 International Joint Conference on*. IEEE. 2014, pages 816–823.
- [3] A. Rogge-Solti and M. Weske. "Prediction of remaining service execution time using stochastic petri nets with arbitrary firing delays". In: *Service-Oriented Computing*. Springer, 2013, pages 389–403.
- [4] W. M. Van der Aalst, M. H. Schonenberg, and M. Song. "Time prediction based on process mining". In: *Information Systems* 36.2 (2011), pages 450–475.





# Location-Based Process Monitoring

Pascal Jung and Thomas Zwerg

Hasso-Plattner-Institut

{Pascal.Jung,Thomas.Zwerg}@student.hpi.uni-potsdam.de

In the world of logistics, the use of real-time information provides a great opportunity. Event Processing has thus become a crucial means. Combining event data with information from external service providers yields further possibilities to leverage the potential of these systems. In this paper, we therefore demonstrate how location-based information such as routes or traffic events can be incorporated in order to enrich data at hand and improve monitoring and planning capabilities.

## 1 Introduction

In times of Big Data and real-time information flow, Event Processing has become a crucial tool for data management. Especially in the scope of logistics, a lot of data comes in form of events, as for instance truck positions that are being emitted by GPS sensors. In this paper, we investigate the integration of location-based information in order to automatically process real-time events relevant to an on-going transport.

In our chosen use case, a truck is transporting goods from one site to another. Our focus are not primarily the goods but the planning of the transport. From a transportation planner's perspective, it's crucial to know when a certain transport will be finished in order to be able to plan ahead. Knowing when a truck will deliver helps to schedule that truck for further transportations, for example by plane or train. There may be restrictions on the departure times, so in case a transport is late for the transport by plane, a rescheduling is necessary. Knowing when a truck arrives at its destination is a very central requirement in logistics that significantly impacts efficiency. Obviously it's not always possible to have a perfect estimated time of arrival (ETA) from the beginning, since unexpected events such as accidents or traffic jams can happen anytime. Thus it's important to constantly reassess routes and estimations.

To process all the information the scenario provides, we are using an event processing platform called UNICORN [11]. Basically the platform's functions are the sending, receiving and processing of events. Input events for our chosen scenario are for example truck orders and position updates which among other things help to locate a specific truck. Knowing where the truck is and where it's supposed to drive helps to calculate the remaining driving distance and based on that the re-

maintaining driving time. From the remaining driving time, an estimated time of arrival can be derived. At the moment the ETA is only a very rough guess because it's not calculated based on an actual route not to mention actual traffic conditions. In the current solution, the remaining driving distance is based on the shortest distance between the current position and the destination. In other words it's calculated as the crow flies which for obvious reasons does not apply for a truck. Our task now is to improve that calculation by incorporating additional real-time information, such as route and traffic events. Once we know what routes will lead us to the desired destination, those routes can be checked for traffic incidents that may impact our choice of the best route. The incorporation of traffic events is particularly challenging, because it requires the proper filtering and selection of potentially relevant events (see Section 3.3). Based on an actual route and the according traffic conditions we can calculate an accurate ETA. Besides, this approach allows the detection of potential delays in case even the best route won't let us get to the destination in time.

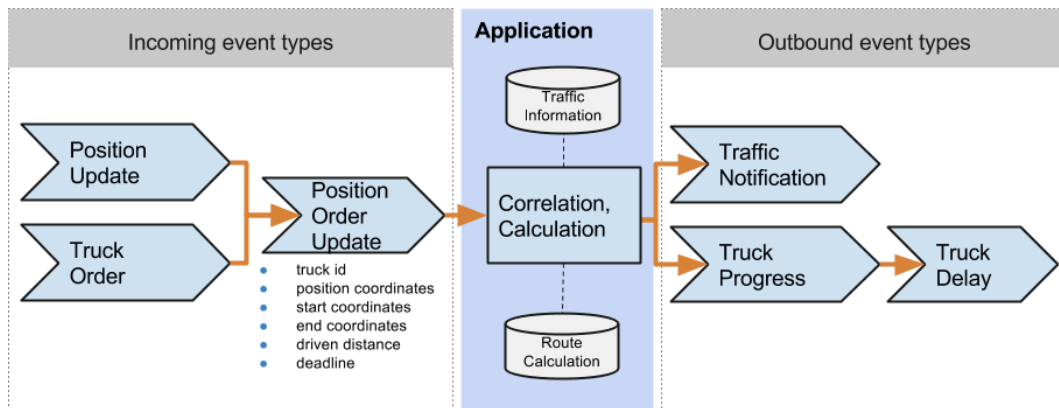
A basis for the functioning of our solution are recent developments in the field of Complex Event Processing (CEP) frameworks and the establishment of Event-Driven Architectures (EDA). Event-driven services that combine Service-Oriented Architecture (SOA) and Complex Event Processing enable us to process streams of heterogenous events on the fly. This integration builds a crucial prerequisite for the effective monitoring and timely processing of incoming events, as for instance, the GPS coordinates of a truck [3]. The approach to use Complex Event Processing for the continuous analysis of events applied to the task of tracking goods and implement a corresponding domain-specific function has been chosen before [7]. Our work builds upon that approach and can be considered a continuation that is also inspired by the GET Service project [4].

## **2 Architecture**

To understand the implementation details, the following section gives a high level overview on the solution's architecture and highlights some of the central ideas behind particular choices. The core implementation can be found in Section 3.

### **2.1 Back end and Front end**

Our solution separates logic and presentation. The core of the solution is the back end, taking care of all the important correlation and calculation steps throughout the entire event processing. The front end, however, serves the purpose of visualizing the results the back end provides during a transport. It displays all relevant data as



**Figure 1:** A slightly simplified overview of the system

for instance, where the truck is located, when it will arrive and what relevant traffic events there are en route. Since we are working with the event processing platform UNICORN [11], we are following the publish-subscribe pattern. Our back end and front end use the platform to benefit from its event processing capabilities and interchange event messages. Both, back end and front end, subscribe to the type of events they plan to act upon in order to receive matching event information published by the platform. It also works the other way around, because our application's results are also published back to the platform. The front end uses ActiveMQ [1], a Java Messaging Service with AJAX capabilities, to obtain order and traffic information from the platform over the web in real-time. More details on the application can be found in Section 4.

## 2.2 Event Processing System

The functions of our solution are mainly driven by events, as shown in Section 1. It shows a simplified version of the system's architecture with the set of relevant events and how they relate to each other.

The most important input event for the system is the Position Order Update (POU) event. It's a simple aggregation of the position update event and the truck order event. Hence it contains information on the truck, its location and the according order, which carries information on the start and end location and the delivery deadline. Once this event has been processed by our application, the results are published in the form of Traffic Notification (TN) and Truck Progress (TP) events. A TN simply describes a traffic incident, it gives some information on the position, the criticality and some textual description. The TP is the most important output of

our application, it describes the current progress and contains an ETA. Since all the available information on the truck itself, the routes and the traffic conditions go into the calculation of an accurate ETA, it's a rather complex task (see Section 3). Once the TP is calculated, another simple aggregation can create a Truck Delay (TD) event, in case a delay is detected. To detect a delay, the ETA is compared to the deadline. If the ETA is greater, a delay is to be expected.

### **2.3 Custom Event Processing Agent leveraging Webservices**

Our back end – an intermediate java application – takes care of the real-time event processing, based on the POU events. Everything that happens inside our application is initially triggered by the incoming events. The processing is repeated as long as there are new events coming in. Through the use of external service providers for routing and traffic information, real-time information is used to determine the best route to drive. The consideration of traffic events for the choice of the route not only helps to avoid busy or closed roads but also improves estimation accuracy because driving speed will depend on the traffic conditions. Furthermore, if the current route has to be discarded due to traffic incidents, those incidents can still be shown to the planner in order to have some additional information and understand why the route has changed.

### **2.4 Selecting a Service Provider**

For the selection of the appropriate web service provider for routing and traffic events, major providers, such as Google Maps [5], Nokia HERE [6], TomTom [10] and Bing Maps [2] have been compared. While all of them provide some sort of routing service, when it comes to traffic information, possibilities vary greatly. HERE was most beneficial due to the possibility of querying traffic events only for a certain area, plus the option to get additional traffic flow information. Since it fit the requirements best, HERE has been used throughout the whole development process including implementation, testing and demonstration of the solution.

## **3 Implementation in Detail**

The general necessity for an intermediate application is based on the problem that there are no providers publishing traffic notification data as events and that in this implementation the calculation of the remaining distance and time depends on the latest best available routing. Therefore, our application requires a trigger to repeat

the request for a route and the corresponding traffic notifications. The trigger used for recalculating the best route, distance and remaining time is received by subscribing to the POU event channel of the underlying event processing platform. Once the routing determination is completed, traffic notifications for this routing are retrieved, validated and finally published as TP and TN events.

### **3.1 Determination of Routing, Estimated Driving Time and Delay**

The basic assumption in this context is that since the application is able to determine the current best route, the truck would also be able to be supplied with this information. We can therefore assume that the truck is always taking the best available route. Even if there is a deviation, e.g. a driver leaves the best route, the next best route gets calculated immediately, triggered by the the next POU event that is received. The eventual decision of the route itself is done by the traffic service provider. Since these providers rarely provide driving speeds or durations for trucks, the time calculation is done by the application. Basically, the remaining distance and a default average speed can be used for calculating the remaining time. The default speed approach can be improved as explained in Section 3.2. Further, for each traffic notification there is a specific delay that represents the additional time needed to pass this incident. These delays are added to the basic remaining driving time. The evaluation of a possible delay is completely done by the event processing platform which compares the order's planned arrival time with the remaining driving time in relation to the current time. If the estimated arrival time, calculated by adding the remaining driving time to the current time, is later than the deadline, a TD event is published.

### **3.2 Optimizing Estimated Driving Time and Delay by Introducing Vehicle Profile**

As previously discussed, the calculation of the remaining driving time as major part of the overall remaining time can be approximated using a default average speed. Since this is an approximative result, we improved it by introducing a Vehicle Profile (VP). This profile allows the application to provide a very accurate determination of the remaining time. The VP basically consists of a mapping of Road Category (RC) to average speed as well as a default speed as a fallback value to be used in case the RC cannot be determined or the speed of a particular RC is not set. All defined RCs and estimated speed values for testing purposes are summarized in Table 1. As a fallback, 60km/h, the average speed for trucks, is used [8]. The main advantage of this VP is the high degree of specification. That is, there can be a specific speed for every single RC or even a dedicated profile per driver based on historic driving data.

**Table 1:** Road categories for vehicle profiles with example values used by the test application (based on [9])

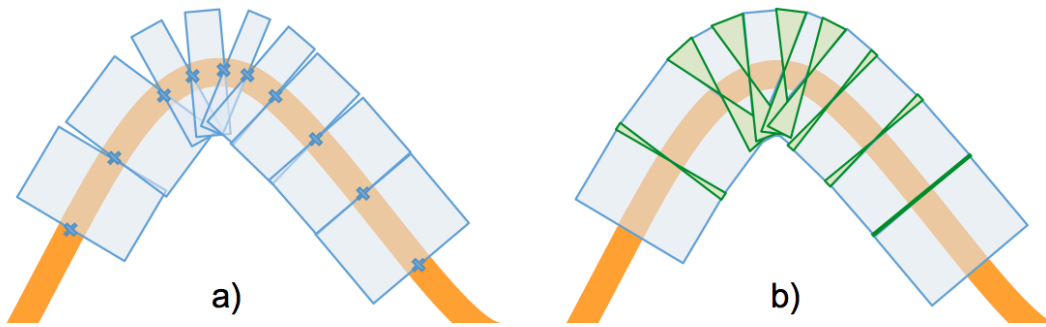
Category	Description	Speed in km/h
0	motorway (Autobahn)	85
1	federal highway (Bundesstraße)	70
2	country road (Landstraße)	60
3	freeway (Schnellstraße)	80
4	town road (innerstädtische Straße)	30

### 3.3 Relevance Determination of Traffic Notifications

The relevance of a TN is defined as being present on the current best route (A) or on the default route (B). The default route is the route that would have been taken if there had not been any incidents. The best route is defined as the best recommendation of the selected service provider. This dual approach is used in order to provide best possible information on TNs causing a delay in the case of A and to show why there had been a rerouting in case of B. Otherwise the recipient that is consuming events may not know why there had been a change of routing. Similarly, currently no service that has been reviewed offers TNs published as events. Therefore, this functionality is added by the intermediate application as well. This again results in using web services, processing its responses and creating events from the outcome of the processing step. There are many possibilities of specifying which TNs should be queried within a single web service call.

It is possible to specify a position and a radius, a geographical rectangle or in case of HERE it is also possible to provide a corridor's coordinates and a width argument to request all TNs within this specific area. The latter is what is used. However, based on this approach, some irrelevant notifications are still sent. This may happen, if a TN is located on an intersecting street or a parallel road. In order to eliminate those irrelevant notifications we perform a re-validation step. The general concept of this re-validation is based on the shape coordinates of the routes segments (cf. Figure 2).

For each segment of a route, e.g. a particular route between two consecutive driving instructions, there are coordinates modeling the exact layout of this road. For every two consecutive coordinates we calculate a rectangle using an offset parameter that describes the distance to both sides of the road. Additionally, it uses the end coordinates of one rectangle and the start coordinates of the following rectangle in order to calculate an additional figure in between. This is important because it prevents false negative determinations since bends would cause large gaps at least



**Figure 2:** Coordinates validation concept based on road shape coordinates, schematic  
 a) simple approach calculating polygons based on shape coordinates of the route  
 b) advanced re-validation taking gaps between subsequent polygons into account to avoid false negative determinations *b)* has been used to validate the positions of TNs

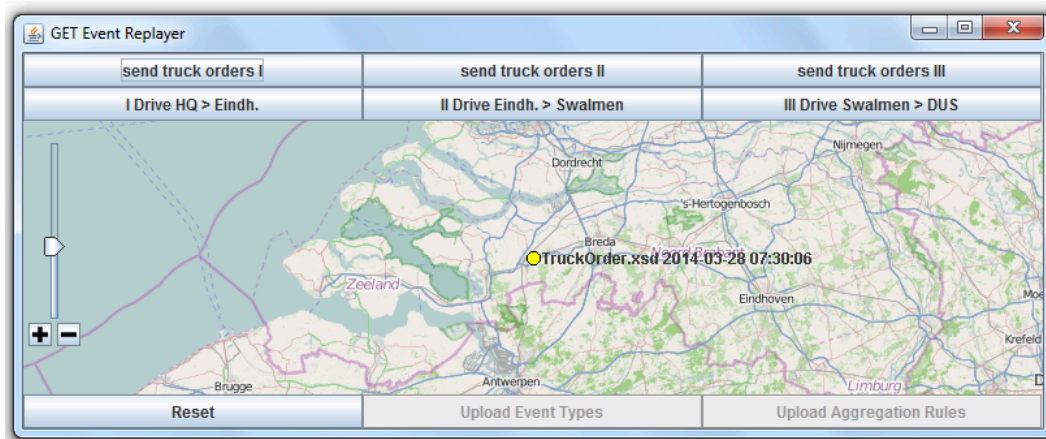
on one side of the road. Eventually it checks whether the start and end coordinates of the TN are inside one of the polygons as shown in Figure 5 (appendix).

Furthermore, using multiple polygons instead of one large polygon supports faster calculation, since it only computes polygon corner positions until TN coordinates are validated inside. In contrast to that, one big polygon would need all corner coordinates to validate the TN. Resulting from this validation, TN events are only created for notifications that are explicitly detected as being on the remaining route or the default route as declared in the beginning of this section.

### 3.4 Further Optimization

In order to increase the precision of the remaining driving time, it should be considered to create profiles specific to a certain driver or segments of a route. These values could then be reused as input for VPs. Since this data has not been available during development, it is not part of the application yet.

Another validation strategy can be to calculate the angle between the connection of TN's start and end coordinates and the connection of corresponding surrounding route coordinates. In case that angle is smaller than a certain threshold, it can be treated as relevant.



**Figure 3:** The event replayer helps to control the application for demonstration purposes

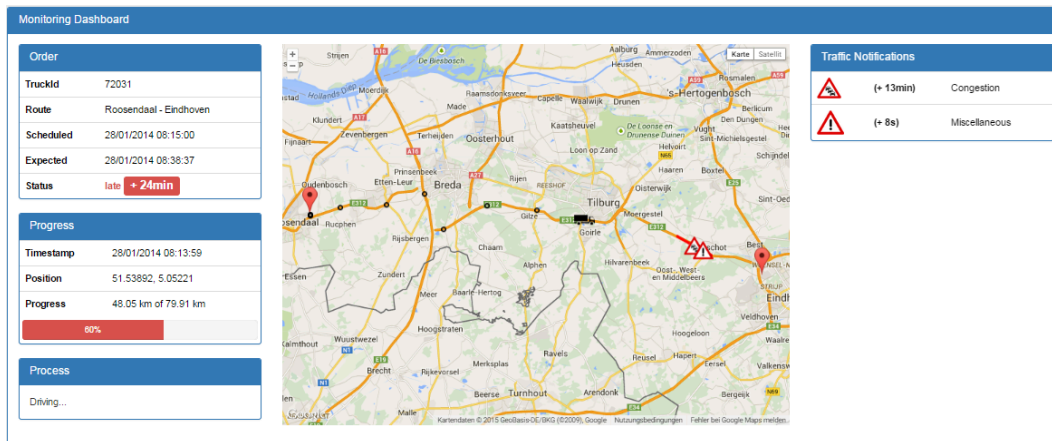
## 4 Application

In the following, the exemplary transport from Rosendaal to Eindhoven illustrates the functionality and benefits of our solution. Just as this transport, all data used by the application is taken from the real world. In order to speed up the demonstration and simplify its repeatability, the data has been cached. An event replayer helps to choose between three previously calculated scenarios (see Figure 3).

Once a scenario has been chosen, the replayer sends the truck's traces to the platform, simulating the driving of the truck. The traces are then converted into events and eventually result in POUs that go into our application. Once the processing as described in Section 3 is done, the TP and TN are published, in case of a delay, a TD as well. Throughout the entire time of the transport, the front end continuously displays the changes as soon as they have been processed. The front end therefore supports the planner by summarizing the most important information in a very concise manner (Section 4).

With the aid of the front end we fully benefit from our fairly complex and otherwise hidden solution. The reduction to a simple and clear user interface (UI) helps to get an overview of the most relevant data without having to understand what's happening in the background. The UI shows us exactly where the truck is at any time, what is to be expected on the road ahead of the truck and above all, it let's us know when to expect the delivery at the final destination (cf. Figure 4). A status indicator informs whether the transport is on time or delayed. If there are significant problems on the road they will also be displayed and explained (see Figures 6 and 7). A planner can





**Figure 4:** The front end displays important information on the transport from Roosendaal to Eindhoven

now easily monitor his transportations and see whether or not he needs to take action. Further, the ability to inform the customer early may lead to an increase of customer satisfaction and also save costs. In addition to that, the traffic-based rerouting leads to a more efficient driving and increases the competitiveness on the market.

## 5 Conclusion

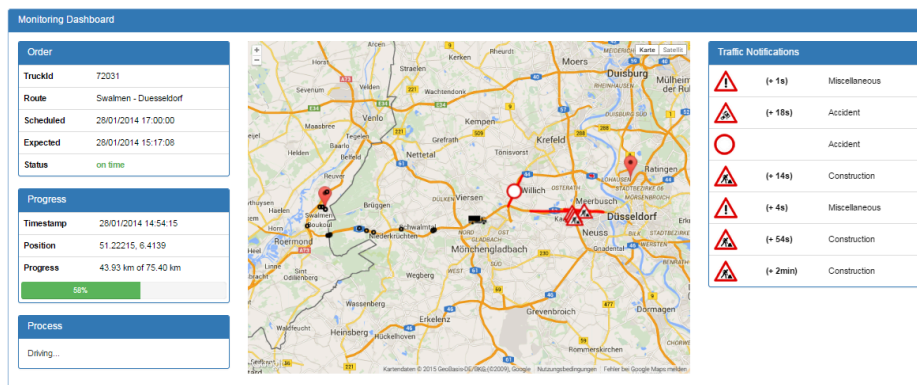
In this paper, we presented a solution to the integration of routing and traffic information for the better planning and monitoring of logistics processes. We did this by using external webservice that provided us with real data on routes and traffic conditions. In particular, we improved the planning capabilities by determining an accurate and realistic estimated time of arrival. Based on that, we further implemented a delay detection and notification system. The truck progress is a useful side product that also forms part of our solution and gives insight on how far the delivery has come at any moment. The introduction of our concept of road categories and customizable vehicle profiles has led to further improvement of the results. We have shown how important it is, to have an exact estimated time of arrival in order to increase flexibility and efficiency. Using our presented solution, it is now possible to anticipate problems on the road and do a timely replanning. Saving costs due to the avoidance of penalties or extra fees for coming late without notifying the customer and increasing customer satisfaction due to a better communication and more reliability in the estimations are side effects that are to be expected but yet have to

be verified. Last but not least, there is still room for improvements. Since this is one of the first attempts of integrating routing and traffic data in the context of event processing, further developments can be expected in the future.

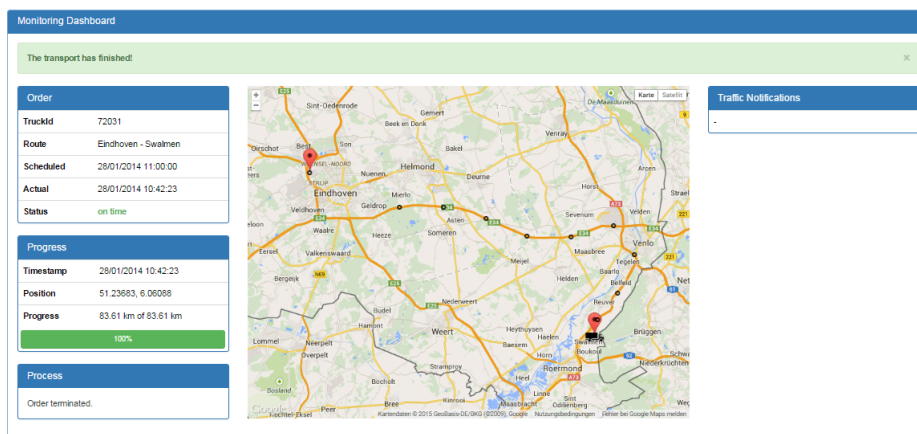
## Appendix

```
Input : Route r, Corridor Width w
Output: Set of relevant traffic notifications
tn :=  $\emptyset$ 
stn := service.requestTrafficNotifications(r.getCoordinates(), w)
foreach notification  $\in$  stn do
  start := notification.getStart()
  end := notification.getEnd()
  prev :=  $\emptyset$ 
  isIn := false
  foreach segment  $\in$  r.getSegments() do
    foreach coord  $\in$  segment.getShapeCoords() do
      if prev! =  $\emptyset$  AND
      isPntInPlygn(start, prev, coord, w) AND
      isPntInPlygn(end, prev, coord, w) then
        stn.add(notification)
        isIn := true
        break
      end
      prev := coord
    end
  if isIn then
    break
  end
end
end
return stn
```

**Figure 5:** Requisition and re-validation of traffic notifications



**Figure 6:** Due to an accident, a road that has been part of our initial route was closed. After the rerouting has been applied, the truck's estimated time of arrival is still within schedule.



**Figure 7:** A finished transport that arrived on schedule, no traffic incidents occurred

## References

- [1] *Apache ActiveMQ: open source messaging and Integration Patterns server*. Available at <http://activemq.apache.org/>, last accessed September 2015.
- [2] *Bing Maps: Choose Your Bing Maps API*. Available at <http://www.microsoft.com/maps/choose-your-bing-maps-API.aspx>, last accessed September 2015.

- [3] A. Buchmann, H.-C. Pfohl, S. Appel, T. Freudenreich, S. Frischbier, I. Petrov, and C. Zuber. "Event-Driven Services: Integrating Production, Logistics and Transportation". In: *2nd International Workshop on Service Oriented Computing in Logistics (SOC-LOG)*. San Francisco, USA, Dec. 2010.
- [4] *GET Service: Efficient Transportation Planning and Execution*. Available at <http://getservice-project.eu/>, last accessed September 2015. 2015.
- [5] *Google Maps API*. Available at <https://developers.google.com/maps/>, last accessed September 2015. 2015.
- [6] *HERE Developer Portal*. Available at <https://developer.here.com>, last accessed September 2015. 2015.
- [7] T. Metzke, A. Rogge-Solti, A. Baumgrass, J. Mendling, and M. Weske. "Enabling Semantic Complex Event Processing in the Domain of Logistics". In: *Service-Oriented Computing – ICSOC 2013 Workshops – CCSA, CSB, PASCEB, SWESE, WESOA, and PhD Symposium, Berlin, Germany, December 2–5, 2013. Revised Selected Papers*. 2013, pages 419–431. doi: 10.1007/978-3-319-06859-6\_37.
- [8] A. Pumberger Fessl. *Lkw-Geschwindigkeitsverhalten auf Autobahnen*. Available at [http://media.arbeiterkammer.at/wien/Verkehr\\_und\\_Infrastruktur\\_44.pdf](http://media.arbeiterkammer.at/wien/Verkehr_und_Infrastruktur_44.pdf). Wien, 2011.
- [9] W. H. Schulz. *Industrieökonomik und Transportsektor: Marktdynamik und Marktanpassungen im Güterverkehr*. Kölner Wissenschaftsverlag, 2004.
- [10] *TomTom Developer Portal*. Available at <http://developer.tomtom.com/>, last accessed September 2015. 2015.
- [11] *Unicorn: platform capture and process real-world events from different sources*. Available at <http://bpt.hpi.uni-potsdam.de/UNICORN>, last accessed September 2015.

# Location-based Process Monitoring: Location and Weather

Lucie Omar and Marvin Richter

Hasso-Plattner-Institut

{Lucie.Omar,Marvin.Richter}@student.hpi.uni-potsdam.de

Execution of process models can be affected by correlated external data from different sources. In the context of transportations in logistics a lot of spatiotemporal events are present with varying impact on the actual transport. We are faced with the challenge to determine relevant relations between them. This paper presents an application which prototypically closes this gap and provides the ability to extract information from event producing services that is beneficial for the transportation process in logistics. The functionality of this application is explained by the use of an demonstration.

## 1 Introduction

Modeling and improving business processes is essential for the success of enterprises. In addition being executed efficiently and correct, it is important to monitor the process execution [4]. Since the availability of event data increases and the geographical traceability of vehicles and physical objects improved with technologies such as the Global Positioning System (GPS) or Radio-Frequency Identification (RFID), a more detailed tracking of individual business tasks is enabled. These available data can be used to support process monitoring based on location [1]. In [5], it is argued that taking the particular context of geospatial information in the various lifecycle steps of Business Process Management (BPM) into account can contribute to improve the effectiveness and efficiency of process management.

On a daily basis, planners at transportation companies plan transportations for different routes and means of transportation. There, transportation resources are assigned to transportation orders. In the ideal case, the transportation takes places just the way it was planned. In reality, transportations in different locations are influenced by unexpected events happening in different places.

The location of these unexpected events is very important, e.g. a strike at an airport, storm on the water or icy conditions on the road are only relevant in case they are close to places where the transportation is executed. Here, also the state of the transportation execution as well as related objects must be considered, as information about icy roads are only important on those streets that the truck still has to drive on. That means location-based information plays an important role in transportation

processes. The mapping of concrete points to an area with changing conditions is a major issue regarding location-based information. While the truck continuously drives on a route, real time information about the location becomes necessary. That raises the question, whether unexpected events should be considered for planning and executing transportations in location-based process monitoring. Furthermore, questions about the ability to support the planner in reacting to these relevant events and determine intersections between the location-based transportation information and relevant events have to be answered.

This paper presents a novel approach to enrich location-based process monitoring in logistics with unexpected weather events, while also considering the state of the execution. Ultimately, the usefulness of having access to these information will be determined. Therefore, we create a running scenario, which will be presented in Section 2 as well as insight into the current usage of weather events. The remainder of this paper is structured as follows. Section 3 presents the architecture and implementation details of our prototypical application which is tested in Section 4. Finally, Section 5 presents the conclusion as well as an outlook.

## **2 Scenario**

In this section, we focus on narrowing down the general challenges of location-based monitoring. For this purpose, we introduce a specific scenario inspired by GET Service<sup>1</sup>. It outlines the importance of our approach and will be used as a running scenario throughout the paper, especially to test our approach and decide, if a planner would benefit from knowledge about intersections between the transportation routes and weather events during the planning and execution of transportations.

In our scenario, a transportation planner is responsible for planning and re-planning transportations. Focusing on the transportations by truck, a planner is responsible for 50–150 trucks on different routes on a daily basis. He has access to the route coordinates, since he plans the route with a routing service and the regular truck position updates that are sent by the trucks. In this planning process, there is a lack of weather information. This makes the planner's reaction regarding weather events passive, because he can only react after it occurred. A truck driver for example contacts the planner that he had an accident due to icy roads. Once the planner gets that information, he has to re-plan the transportation. That results in major delays and much additional work for the planner, which is time consuming and expensive. The

---

<sup>1</sup> <http://www.getservice-project.eu/>, last accessed September 2015.

only way that weather is used in the planning process is by decreasing the average speed in the planning process during the winter season, but that certainly not enough.

Our approach aims at providing the planner with real time information about weather alerts, which correlate with his routes, to give him the possibility to react actively instead of passively. Being able to act before the weather event occurs, supports the prevention of weather related accidents and delays. In order to disburden the planner, only relevant weather alerts should be sent. We define relevant weather alerts as the ones that intersect with the remaining transportation route, which means the truck driver still has to pass that area. Furthermore, we selected a transportation, in which a truck is the means of transportation. The route goes from Cologne to Mannheim.

### 3 Architecture

This section describes the architecture and the implementation details of our weather warning system. This implementation is used in Section 4 in the context of the scenario described in Section 2. Figure 1 provides an overview about the main components of our application. The weather warning system consists of functional components querying, transforming and producing events, in particular spatiotemporal events. Spatiotemporal events are defined as ‘events that happen in a certain location and at a certain point in time’ [3].

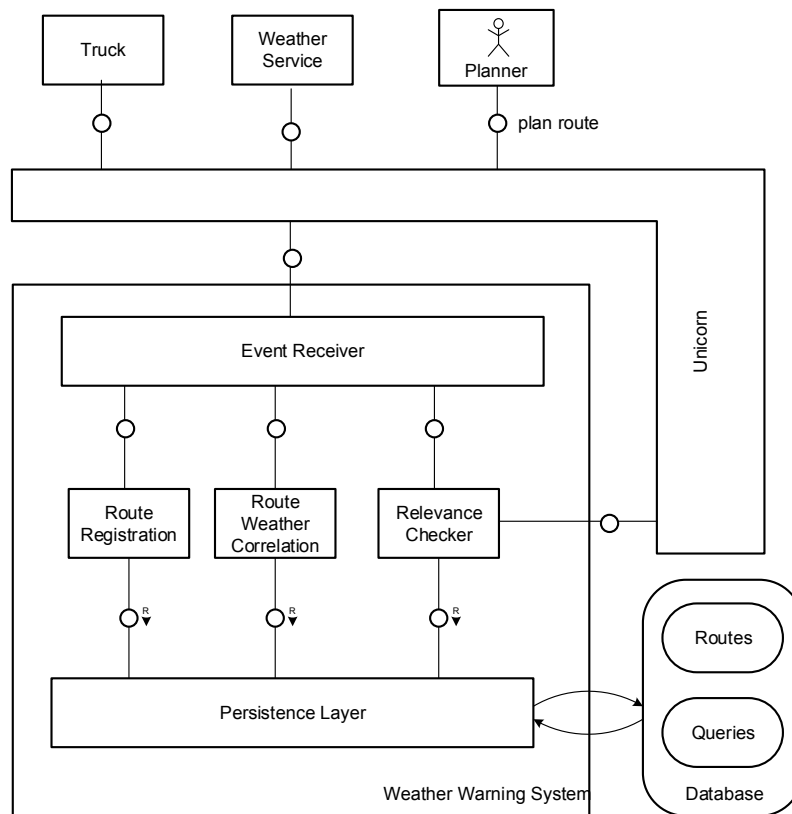
All components obtain their corresponding events from the *event receiver* which subscribes to previously registered Java Message Service (JMS) queues. In our case Apache ActiveMQ<sup>2</sup> is used as a JMS provider, which delivers events published by Unicorn<sup>3</sup>, a platform for capturing and processing real-world events using ESPER as an event processing engine<sup>4</sup>. The events are supplied in JSON format. Thus the *event receiver* is responsible for parsing and passing them on to one associated component.

There are three event producers that matter to the weather warning system: The *truck*, sending position updates frequently, a *weather service*, providing weather warnings, and the *planner*, entering the route information in form of a polygonal chain of route coordinates. In addition, the weather warning system contains a *persistence*

<sup>2</sup> Apache ActiveMQ: open source messaging and Integration Patterns server. <http://activemq.apache.org/>, last accessed September 2015.

<sup>3</sup> Unicorn: platform capture and process real-world events from different sources. <http://bpt.hpi.uni-potsdam.de/UNICORN>, last accessed September 2015.

<sup>4</sup> <http://www.espertech.com/products/esper.php>, last accessed September 2015.



**Figure 1:** Architecture of the weather warning system

layer to provide an uniform interface to access data. On the one hand administrative data in form of subscription queries are available, on the other hand mutable data in form of route information with correlated *WeatherWarning* events are made accessible. The core of the weather warning system comprises three main components: the *route registration*, the *weather route correlation* and the *relevance checking* component.

The **route registration** component provides the ability to collect 2D coordinate events referring to predefined relevant routes. These coordinates will be stored in the form of an open polygon in a local route storage via the persistence layer. After the transport is completed, these route data will be removed.

The **weather route correlation** component is triggered by each incoming *WeatherWarning* event. It is responsible for correlating the incoming event to all routes stored in the route storage. To perform necessary geospatial operations we choose



the JTS Topology Suite<sup>5</sup>. It is a Java library that provides a geometry object model, including points, lines and polygons, and implements the Open Geospatial Consortium (OGC) API consisting of essential 2D geometric functions. In our case we utilize the intersection functions on polygons to establish the correlation between the area of the *WeatherWarning*, presented as closed polygonal geometry, and the open polygonal chain of the route coordinates. Once an intersection is detected this component creates a *WeatherOnRoute* event with references to the respective route and weather event.

The **relevance checker** is used to detect if the determined *WeatherOnRoute* events make an impact on the respective truck. The component is triggered by each incoming truck position update. The truck coordinates are generally imprecise and need to be correlated to the route in order to determine the remaining route. Due to the fact that a rough approximation is sufficient (weather events have blurred borders), we choose an obvious approach. We determine the route coordinate with the minimal distance from the truck position as the start point of the remaining route. If an impact is detected a *WeatherWarningAhead* event, enriched by additional information as for instance the calculated distance between the truck and the weather area, will be produced. Since the truck and the weather events are spatiotemporal events, it is possible to derive temporal and spatial relations between these events and build up a spatiotemporal pattern and a distance network respectively, as discussed in [2]. Once a *WeatherOnRoute* event is processed at this stage, it is going to be removed from local storage.

An alternative implementation approach could have been to extend the ESPER query language by the OGC Geospatial Functions. However, that would result in a dependence on ESPER based event processing systems. Our implementation, in contrast, could be embedded into every event processing network.

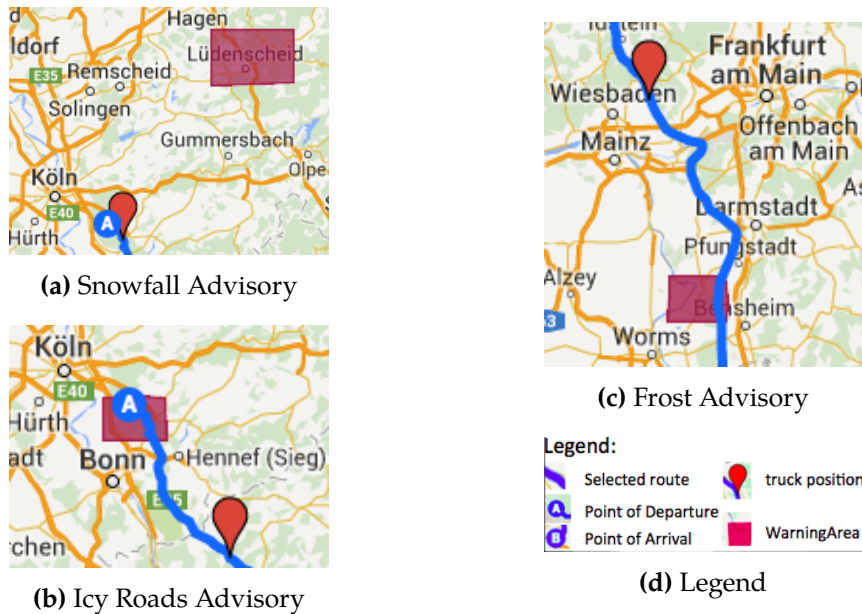
## 4 Application

In this section, we use the running scenario to test our approach. In addition to the route coordinates that represent the transportation route from Cologne to Mannheim, we use the truck position updates that the truck sends. These are the two event types, the transportation planner has access to. A screencast demonstrating our approach can be found at: <http://bpt.hpi.uni-potsdam.de/UNICORN>, last accessed September 2015.

---

<sup>5</sup>JTS Topology Suite: API of 2D spatial predicates and functions. <http://www.vividsolutions.com/jts/JTSHome.htm>, last accessed September 2015.

For this demonstration, three weather events are used, which are based on the weather events from the German Weather Service (DWD). These three weather alerts have been chosen, so three different cases can be demonstrated in the following. The result produced by our weather warning system is then compared to the visualization of the scenario. Google Maps is used to visualize the route coordinates (combined to a line), the truck position and the alert area, as seen in the legend in Figure 2d.



**Figure 2:** Visualization of the weather warning events

### Snowfall Advisory

The first weather alert is the snowfall advisory. As seen in Figure 2a, there is no intersection between the route and the alert area. Hence, it is not a relevant weather alert in our running scenario. As shown in Figure 3, the weather warning system does not associate the alert with the route, which is correct.

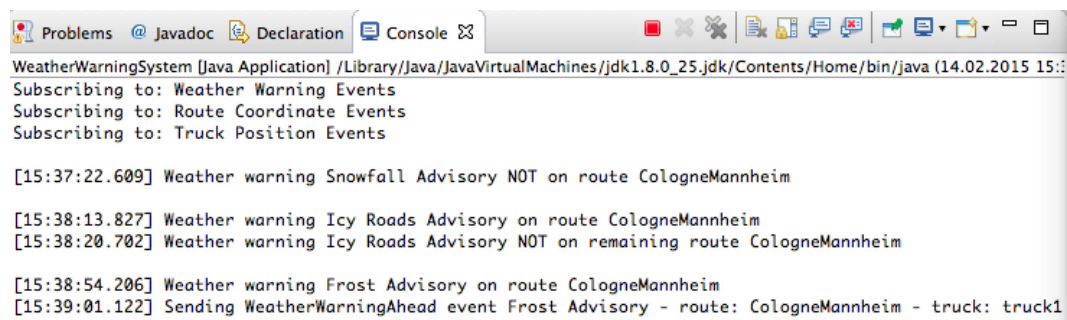
### Icy Roads Advisory

As a second weather alert we have the icy roads advisory, which is effective in the Cologne area, as shown in Figure 2b. This event has an intersection with the trans-

portation route. Since the truck is driving from Cologne to Mannheim and is already south of Cologne, it is not on the remaining route and therefore not a relevant weather alert. The weather warning system checks the whole route for an intersection with the weather alert polygon and correctly determines the icy roads advisory intersects with the transportation route. In the second step, the system computes the remaining route according to the latest truck position. In this case, it determines correctly that the alert area is not on the remaining route.

### Frost Advisory

The third weather alert is the frost alert, which is effective in the Gernsheim area. As seen in Figure 2c, there is an intersection between the route and the alert polygon. This time, the truck position is north of this area and the truck is still driving towards the frost advisory area. This makes it a relevant event and a *WeatherWarningAhead* event should be sent. The weather warning system determines that the route intersects with the alert area. In the second step, based on the remaining route, it determines that the truck is driving towards the alert area and the *WeatherWarningAhead* event is sent. This is correct, since this complies with our definition of a relevant weather alert for the transportation planner.



```

WeatherWarningSystem [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_25.jdk/Contents/Home/bin/java (14.02.2015 15:
Subscribing to: Weather Warning Events
Subscribing to: Route Coordinate Events
Subscribing to: Truck Position Events

[15:37:22.609] Weather warning Snowfall Advisory NOT on route CologneMannheim

[15:38:13.827] Weather warning Icy Roads Advisory on route CologneMannheim
[15:38:20.702] Weather warning Icy Roads Advisory NOT on remaining route CologneMannheim

[15:38:54.206] Weather warning Frost Advisory on route CologneMannheim
[15:39:01.122] Sending WeatherWarningAhead event Frost Advisory - route: CologneMannheim - truck: truck1

```

Figure 3: Weather warning system output

Even though three weather alerts were evaluated by the weather warning system (see Figure 3), only one *WeatherWarningAhead* event is sent to the planner. This is the relevant weather warning, which is enriched with information about the transportation route, the latest truck position and the distance between the latest truck position and the entrance point to the alert area. Figure 4 visualizes these information as well as a map showing the route, the alert area and the latest truck position. This gives the planner the possibility to react to the weather warning in real time. It enables the

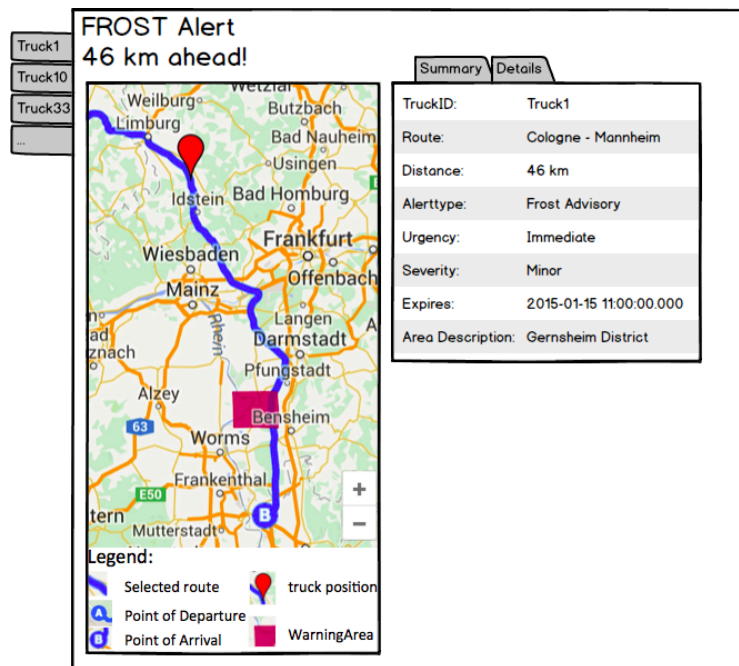


Figure 4: Possible visualization of the *WeatherWarningAhead* event

planner to actively decide, if re-planning should be used to adjust the transportation route in order to avoid the alert area. This decision can be made by means of the kind of weather alert, the distance to the alert area, the severity level and the expertise of the transportation planner.

## 5 Conclusion

In this paper, we addressed the challenge of mapping concrete points to areas with changing conditions referring to moving trucks on different routes and possible intersections with unexpected weather events. It resulted in the question, whether weather events should be considered for transportations in logistics. To cope with those challenges, we proposed an application that determines relevant weather events and creates a new event with valuable information about the weather alert, the truck and the route, enriched with distance information. To test our application, we have sent manipulated events and compared the expected results with the output of our application. All results were accurate. The application enables the planner to use relevant weather information in the planning and the execution of transporta-

tions. That gives him the chance to intervene before the event affects the truck. Thus, we recommend including weather information in the transportation process.

At the current state we have implemented a prototype that merely serves as a proof of concept. In the following we want to propose conceivable improvements. By now there are no filtering methods implemented that consider the expected impact of weather events on the transportation. On the basis of historical data and empirical knowledge an ordinal property might be introduced to describe the degree of influence. On this basis, one can filter weather events of interest, provided that a proper threshold is chosen. In our case we confine ourselves to considering weather events. Thus, a next step could be to generalize the presented concepts.

Furthermore, we merely considered the German Weather Service. In our future work we will, therefore, also investigate other weather services, among other tasks this includes additional transformation rules in the normalization phase for their integration.

## References

- [1] C. Cabanillas, C. D. Ciccio, J. Mendling, and A. Baumgrass. “Predictive Task Monitoring for Business Processes”. In: *BPM’14*. 2014, pages 424–432.
- [2] B. M. Foued Barouni. “An Extended Complex Event Processing Engine to Qualitatively Determine Spatiotemporal Patterns”. In: *Global Geospatial Conference*. 2012.
- [3] H. H. and B. Moulin. “A framework to support qualitative reasoning about COAs in dynamic spatial environment”. In: *Journal of Experimental and Theoretical Artificial Intelligence* 22.4 (2010), pages 341–380.
- [4] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Englisch. Auflage: 2nd ed. 2012. Heidelberg ; New York: Springer, 2012.
- [5] X. Zhu, G. Zhu, S. van den Broucke, J. Vanthienen, and B. Baesens. “Towards location-aware process modeling and execution”. In: *Business Process Management Workshops*. 2014, pages 186–197.



# Aktuelle Technische Berichte des Hasso-Plattner-Instituts

<b>Band</b>	<b>ISBN</b>	<b>Titel</b>	<b>Autoren / Redaktion</b>
101	978-3-86956-346-6	<b>Exploratory Authoring of Interactive Content in a Live Environment</b>	Philipp Otto, Jaqueline Pollak, Daniel Werner, Felix Wolff, Bastian Steinert, Lauritz Thamsen, Marcel Taeumel, Jens Lincke, Robert Krahn, Daniel H. H. Ingalls, Robert Hirschfeld
100	978-3-86956-345-9	<b>Proceedings of the 9th Ph.D. retreat of the HPI Research School on service-oriented systems engineering</b>	Christoph Meinel, Hasso Plattner, Jürgen Döllner, Mathias Weske, Andreas Polze, Robert Hirschfeld, Felix Naumann, Holger Giese, Patrick Baudisch, Tobias Friedrich (Hrsg.)
99	978-3-86956-339-8	<b>Efficient and scalable graph view maintenance for deductive graph databases based on generalized discrimination networks</b>	Thomas Beyhl, Holger Giese
98	978-3-86956-333-6	<b>Inductive invariant checking with partial negative application conditions</b>	Johannes Dyck, Holger Giese
97	978-3-86956-334-3	<b>Parts without a whole? : The current state of Design Thinking practice in organizations</b>	Jan Schmiedgen, Holger Rhinow, Eva Köppen, Christoph Meinel
96	978-3-86956-324-4	<b>Modeling collaborations in self-adaptive systems of systems : terms, characteristics, requirements and scenarios</b>	Sebastian Wätzoldt, Holger Giese
95	978-3-86956-324-4	<b>Proceedings of the 8th Ph.D. retreat of the HPI research school on service-oriented systems engineering</b>	Christoph Meinel, Hasso Plattner, Jürgen Döllner, Mathias Weske, Andreas Polze, Robert Hirschfeld, Felix Naumann, Holger Giese, Patrick Baudisch
94	978-3-86956-319-0	<b>Proceedings of the Second HPI Cloud Symposium "Operating the Cloud" 2014</b>	Sascha Bosse, Esam Mohamed, Frank Feinbube, Hendrik Müller (Hrsg.)
93	978-3-86956-318-3	<b>ecoControl : Entwurf und Implementierung einer Software zur Optimierung heterogener Energiesysteme in Mehrfamilienhäusern</b>	Eva-Maria Herbst, Fabian Maschler, Fabio Niephaus, Max Reimann, Julia Steier, Tim Felgentreff, Jens Lincke, Marcel Taeumel, Carsten Witt, Robert Hirschfeld







ISBN 978-3-86956-347-3  
ISSN 1613-5652