Model-Driven Event Query Generation for Business Process Monitoring*

Michael Backmann, Anne Baumgrass, Nico Herzberg,
Andreas Meyer, and Mathias Weske

Hasso Plattner Institute at the University of Potsdam
Prof.-Dr.-Helmert-Str. 2–3, D-14482 Potsdam, Germany
Michael.Backmann@student.hpi.uni-potsdam.de
{Anne.Baumgrass,Nico.Herzberg,Andreas.Meyer,
Mathias.Weske}@hpi.uni-potsdam.de

Abstract. While executing business processes, a variety of events is produced that is valuable for getting insights about the process execution. Specifically, these events can be processed by Complex Event Processing (CEP) engines to deliver a base for business process monitoring. Mobile, flexible, and distributed business processes challenge existing process monitoring techniques, especially if process execution is partially done manually. Thus, it is not trivial to decide where the required business process execution information can be found, how this information can be extracted, and to which point in the process it belongs to. Tackling these challenges, we present a model-driven approach to support the automated creation of CEP queries for process monitoring. For this purpose, we decompose a process model that includes monitoring information into its structural components. Those are transformed to CEP queries to monitor business process execution based on events. For illustration, we show an implementation for Business Process Model and Notation (BPMN) and describe possible applications.

Keywords: Business Process Management, Complex Event Processing, Business Process Monitoring, Event Pattern Language Query Generation.

1 Introduction

During business process execution, various systems and services produce a variety of data, messages, and events that are valuable for gaining insights about business process execution [13]. This data enables business process monitoring, e.g., to ensure a business process is executed as expected. However, nowadays, business processes are executed in different places, times, and by a variety of people or devices leading to more mobile, flexible, and distributed business processes. In a business process, activities are executed automatically and manually, where manual execution may be supported by information systems. In this environment, the different systems used to execute business processes generate a large amount of events (e.g., Global Positioning System (GPS) signals of driving trucks) that can be used to enable the monitoring of business processes across enterprise boundaries [6, 11, 15].

* The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement 318275 (GET Service).

A.R. Lomuscio et al. (Eds.): ICSOC 2013 Workshops, LNCS 8377, pp. 406–418, 2014.

The detection and processing of events originating from different systems can be handled by Complex Event Processing (CEP) engines [9,13]. In contrast, the orchestration and enactment of business processes is in the control of Business Process Management (BPM) systems [16, 19] in semi-automated environments. Existing works argues that the incorporation of modeling process logic and describing complex event patterns is essential to capture the overall process context [2]. Thus, there is the need to complement the modeling and execution of business processes in semi-automated execution environments with CEP capabilities.

In this paper, we demonstrate the combination of BPM with CEP for monitoring business process execution in semi-automated environments. We utilize the Refined Process Structure Tree (RPST) [17] to decompose process models into their structural components. Subsequently, these components are automatically transformed into CEP queries using an Event Pattern Language (EPL). For process modeling, we use Business Process Model and Notation (BPMN) [14] enriched with process event monitoring points (PEMPs) [11] that specify where which events are expected during execution. The corresponding CEP queries derived from the BPMN model are represented by Esper EPL [3, 8]. Although we use BPMN and Esper, the general approach presented in this paper is not restricted to a certain process modeling notation nor an EPL. The concept of PEMPs allows to exactly specify which parts of a process model shall be monitored, while existing approaches aim at monitoring each construct of the process model. This may lead to unexpectedly incomplete event logs resulting in severe issues with respect to CEP. In summary, the query generation presented in this paper can be conducted without the need to learn a specific syntax of an EPL and helps if either only parts of a process are of interest or some parts are not observable due to missing sensors, for instance.

The remainder of this paper is structured as follows. Section 2 introduces the set of basic notations that we use throughout the paper. Next, Section 3 describes the scenario including its process model which is used in this paper to demonstrate our approach. Our automation of the query generation from process models is given in Section 4. It includes the three necessary steps to generate CEP queries and the description of their implementation. Afterwards, the application areas of our approach are shown in Section 5 followed by the comparison of our approach with related work in Section 6. Finally, Section 7 concludes this paper.

2 Preliminaries

Working with CEP requires a profound understanding of events and their utilization. An event is a real-world happening occurring in a particular point in time at a certain place in a certain context [13]. Capturing an event in an information system requires the transformation of an event into an event object, each being classified by an event object type [11]. In the process context, we define both concepts as follows.

Definition 1 (Event object). An *event object* $\mathcal{E} = (type, id, P, timestamp, \mathcal{C})$ refers to an event object *type* $\mathcal{ET}$, has a unique identifier *id*, refers to a set P of *process instances* being affected by the event object, has a *timestamp* indicating the occurrence time, and contains an additional *event content* $\mathcal{C}$. ◇

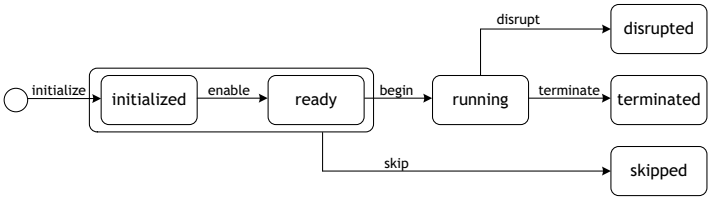

Fig. 1. Activity life cycle (cf. [19])

Definition 2 (Event object type). An *event object type* $\mathcal{ET} = (name, cd)$ refers to a unique *name* indicating the object type identifier and has a *content description* cd of a particular event being of this event type. ◇

As indicated in Definition 1, event objects affect one or several process instances by indicating, for instance, process state changes. Each process instance refers to exactly one process model, which we define as follows.

Definition 3 (Process model). A *process model* $M = (N, F, \eta, \mu, \psi)$ contains of a finite non-empty set $N \subseteq A \cup E \cup G$ of flow nodes being *activities* A, *events* E, and *gateways* G. Events $E \subseteq E^S \cup E^I \cup E^E$ are distinguished into start events E^S, intermediate events E^I, and end events E^E. $F \subseteq N \times N$ represents the *control flow relation* which constraints the partial order of nodes. Functions $\eta : E^S \cup E^E \rightarrow \{plain, message\}$ and $\mu : E^I \rightarrow \{message, time, cancel, error\}$ assign a type to each event. Function $\psi : G \rightarrow \{xor, and\}$ assigns a type to each gateway. ◇

We require each process model to be structural sound and block-structured[1]. Process monitoring deals with capturing events based on node execution; but monitoring on node level may be too coarse-grained. Assuming, there exist waiting times between the execution of two activities, the termination of the first activity does not indicate the start of the second one such that multiple measures are needed. Therefore, we utilize the concept of life cycles for nodes of a process model [19] and attach PEMPs to the state transitions of the node life cycles [11]. Formally, we define a node life cycle as follows.

Definition 4 (Node life cycle). A *node life cycle* $L = (S, T, \varphi)$ contains of a finite non-empty set S of states and a finite set $T \subseteq S \times S$ of state transitions. Let $\mathcal{L}$ be the set of all node life cycles defined for the nodes N of process model M. Then, there exists a function $\varphi : N \rightarrow \mathcal{L}$ assigning a node life cycle to each node $n \in N$ of M. ◇

Fig. 1 depicts the life cycle L_A for activities consisting of states *initialized*, *ready*, *running*, *terminated*, *disrupted*, and *skipped* connected by transitions *(i)nitialize*, *(e)nable*, *(b)egin*, *(t)erminate*, *(d)isrupt*, and *(s)kip*. For events and gateways, we utilize a subset of these states removing states *running* and *disrupted* and the transitions leading to

[1] The process model contains exactly one start and one end event and each node is on a path from that start event to the end event. Each activity has exactly one incoming and one outgoing edge, each start event has one outgoing and no incoming edges, each end event has one incoming and no outgoing edges, each gateway has at least three edges with either exactly one incoming or exactly one outgoing edge, and for each merging gateway there exists a splitting one.

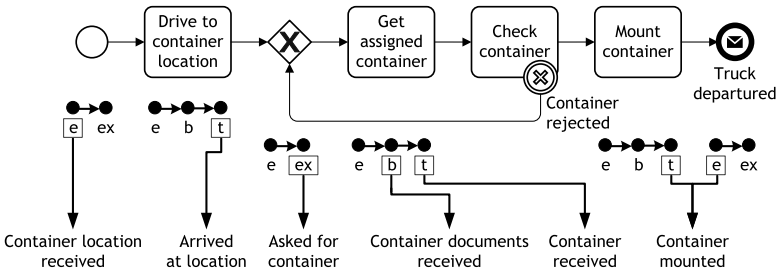

Fig. 2. Business process model of a container pick-up process modeled in BPMN with associated node life cycles and event object types

them. Further, the states *ready* and *terminate* are connected via transition *(ex)ecute*. We distinguish state transitions into the ones observable by occurring events and the ones requiring the context of the process instance to deduce their triggering. For a given node of a process model, each state transition belongs to either group. Utilizing PEMPs for process monitoring is independent from the process instance execution. Thus, a PEMP can only be attached to state transitions being directly observable.

Definition 5 (Process event monitoring point). Let M be a process model, L a node life cycle, and $O_L \subseteq T_L$ the set of state transitions not requiring process instance information. Then, a *process event monitoring point* is a tuple $PEMP = (M, n, t, et)$, where M is the process model it is contained in, $n \in N$ is the node of the process model it is created for, $t \in O_L$ is a state transition within the node life cycle L it is created for, and $et \in \mathcal{ET}$ is an event object type specifying the event object to be recognized. $\diamond$

3 Scenario

Next, a business process from the logistics domain is used as scenario to discuss the approach presented in this paper (see Fig. 2). Assume, a terminal stores containers from different companies and provides them to truck drivers as requested by the owners of the container. First, the truck driver needs to drive to the pick-up location of the terminal. Arrived there, the driver gets the container assigned to her regarding the company the driver is executing the transport for. Second, the driver checks the container she received for several aspects like, for instance, sufficient capacity, special capabilities as cooling means if required, tidiness, and intactness. While the first two mentioned checks are rather guaranteed aspects due to the booking in advance of containers, the two latter aspects are very critical. If any of the checks leads to a negative result, the task `Check container` gets canceled and the corresponding intermediate event is raised. As next step, the driver gets another container, which she checks again. If all checks succeed, the driver can mount the container to her truck and depart from the pick-up location.

To each task, event, and gateway of the process model shown in Fig. 2, a life cycle is assigned that consists of the states and transitions introduced above. For activities, the state transitions *enable*, *begin*, and *terminate* are potentially observable by

occurring events. For gateways and events, such state transitions are *enable* and *execute*. In contrast, the disruption of an activity by an attached intermediate event or the skip of an activity due to exclusiveness cannot be observed directly. For instance, the triggering of state transition *disrupt* can only be deduced by observing the execution of that attached event. Potentially, each of these observable transitions is connected to a PEMP [10] used for process observation. However, this does not mean that each observable transition is actually monitored during process execution. Transitions may be excluded from monitoring due to unavailable capturing mechanisms for corresponding event data or due to the stakeholders' interests. For instance, the start of the process is observable when the driver receives the location where she should pick up the container. For activity `Drive to container location`, we receive GPS coordinates that we aggregate to identify when the truck arrived at its destination; i.e., activity termination takes place. In contrast, activity `Check container` cannot be observed, because it is a manual task without any system interaction. Likewise, the intermediate cancellation event cannot be captured, because it is a manual interaction between the driver and the pick-up location worker. However, indirectly, the happening of both can be derived whenever another container is requested. Altogether, we are able to observe at least one state transition for the tasks `Drive to container location`, `Get assigned container`, and `Mount container`, the start and end events as well as the gateway by recognizing events of the event object types specified in the lower part of Fig. 2. Task `Check container` and the attached intermediate cancellation event cannot be observed.

4 Query Generation

Monitoring the execution of a process instance requires CEP queries to recognize state transitions of its given process model. Next, we introduce the algorithm enabling the automatic generation of such queries from a process model with attached Process Event Monitoring Points (PEMPs) as introduced above. This algorithm comprises three main steps which will be explained in detail in the following sections. First, a RPST is created from a given process model (see Section 4.1). The RPST is based on the edges of a process model, but for the actual query generation, we require the nodes. Therefore, in a second step, we transform the RPST into a *component tree* representing the structure of the process model based on nodes (see Section 4.2). Finally, we utilize this component tree and automatically create the CEP queries for the given process model (see Section 4.3).

Fig. 3 illustrates the first two steps of this algorithm for the process model introduced in Section 3 and again shown in Fig. 3a. Fig. 3b depicts its graph used to construct the RPST given in Fig. 3c. The resulting component tree is presented in Fig. 3d and its transformation (resp. step three) to CEP queries in Section 4.3. The implementation for this particular case is given in Section 4.4.

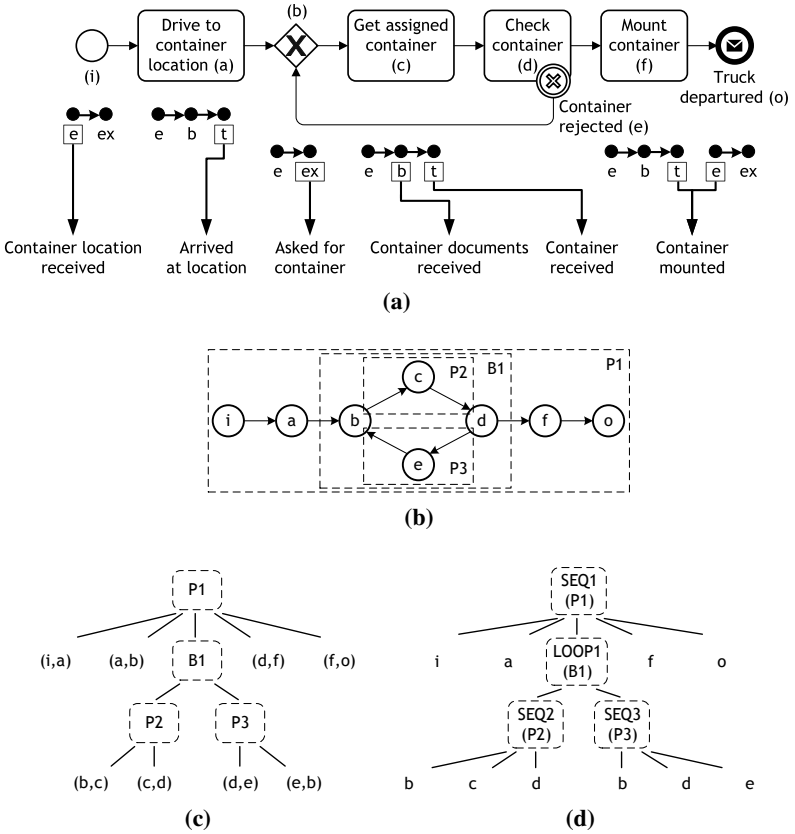

Fig. 3. Scenario as (a) BPMN model, (b) graph, (c) RPST, and (d) component tree

4.1 Creation of the RPST

In order to generate queries from a process model, it is necessary to split the model into smaller parts (sub-graphs). In particular, we use the RPST that decomposes a model respectively the graph into a hierarchy of single-entry / single-exit (SESE) blocks. These SESE blocks have special characteristics as they are canonical fragments. A fragment is canonical if its contained nodes do not overlap with the nodes of another fragment, i.e., the nodes of canonical fragments can either interleave or be disjoint. A formal definition is provided in [17]. Thus, the RPST is a tree which contains the canonical fragments of a graph as tree nodes and the edges between nodes of the graph as tree leaves (see Fig. 3c). Additionally, this decomposition provides special characteristics for the canonical fragments which are derived from the so called triconnected components. Note that each canonical fragment is also a triconnected component each of which being either a bond, a polygon, or a rigid (see [17]).

4.2 Transformation of a RPST to a Component Tree

In the second step, the RPST is transformed into a node-oriented tree, the so called **component tree**. This tree contains a component for every canonical fragment and its leaves are build from the edges of this fragment. For example, Fig. 3d shows that for every canonical fragment (P1, B1, P2 and P3) from the RPST shown in Fig. 3c a component is created in the component tree. The edges of a graph contained in the RPST such as (b, c) and (c, d) for the P2-fragment are then split up into its contained notes b, c and d. These nodes represent the XOR-gateway as well as the `Get assigned container` and the `check container` activity.

Afterwards, we assign a type to every component of the tree. The component types characterize the behavior of the component and are distinguished into *AND*, *XOR*, *Sequence*, *Loop* and *SubProcess*. The assumption of block-structuredness enables an easy mapping of polygons to sequences and bonds to the other mentioned component types. AND, XOR, Sequence, and Loop represent the control flow structure with AND- or XOR-Gateways, while SubProcess indicates an entire subprocess as component. Corresponding to that, the scenario process model contains three components of type Sequence (SEQ1, SEQ2, SEQ3) and one component of type Loop (LOOP1) (see Fig. 3d).

4.3 Query Generation from the Component Tree

Finally, we generate event processing queries from component trees to monitor the execution of a process model. A query is generated for every node in the component tree. Depending on the component types, we generate different types of queries. In addition, we consider the PEMPs of a process model to create queries for state transitions. These *state transition queries* allow to monitor the life cycles of observable process nodes.

A CEP query can be written in any EPL, e.g., Esper [3]. Using an EPL allows to query an *event stream* and use patterns as part of the query to define particular ordering relations among the events and its event types respectively. For demonstration, we used the *Esper Query Language*. The CEP query pattern for the component types and state transitions of BPMN in Esper are summarized in Table 4.3, where $et_1..et_{11} \in \mathcal{ET}$ are the event types that are expected. These event types can be defined in a PEMP. As a subprocess is a specific type of an activity that can contain several other flow nodes and control flows, it is transformed to CEP queries as complement of the other patterns. For example, the process shown in Fig. 2 could be seen as subprocess part of a complete transportation chain including planning and invoicing.

All the queries are ordered and nested according to the fact that they can depend on each other. The triggering of one query can expedite the progress of other queries which are on a higher hierarchy level. While the sequence in which the queries are called is derived from the process-flow, the hierarchy is derived from structure of the component tree. For instance, in Fig. 3d the query for LOOP1 depends on SEQ2 that itself depends on the state transition query for node c. Examples for the implementation of these queries is given in Section 4.4.

Table 1. Esper patterns for query generation

BPMN pattern	Esper pattern
Loop	FROM PATTERN [(EVERY S4=et_1) UNTIL EVERY S5=et_2]
Sequence	FROM PATTERN [(EVERY S0=et_3 → EVERY S1=et_4)]
XOR	FROM PATTERN [(EVERY S0=et_5 OR EVERY S1=et_6)]
AND	FROM PATTERN [(EVERY S0=et_7 AND EVERY S1=et_8)]
State transition	FROM PATTERN [(EVERY S0=et_9 → EVERY S1=et_{10} → EVERY S2=et_{11})][2]

In addition, we support intermediate timer events and intermediate cancel events. An intermediate timer event has a duration from which we generate a *timer query* that waits for the specified time and signals the expiration of the timer duration. Intermediate cancel events are attached to activities and subprocesses. The cancel events can indicate the abortion of the node the cancel event is attached to. Thereby, it is possible to monitor models with expected runtime exceptions.

4.4 Implementation

A process can be monitored based on its process model. We have implemented a service in our Event Processing Platform (EPP) [4, 11][3] which generates the component tree from a process model that includes PEMPs. We provide two combinations for importing process models and defining its PEMPs. First, the business user may import BPMN-specific models and directly adapt the PEMP definitions for specific nodes in the user interface of the EPP. Second, we defined an BPMN extension[4] with which life cycles and PEMPs can be attached to a node in a BPMN model used to derive CEP queries. Thus, process models used in the EPP are specified in the BPMN-conform XML format and include the representation of state transitions of activities, gateways, and events using PEMPs. Finally, we take these annotated models and generate the CEP queries.

In our EPP, each query must be written and registered before the events can be captured and processed. The EPP registers each CEP query in Esper [3] via listeners. These listeners get informed if the query matches observed events with the specified conditions defined by the query. Based on the patterns given in Table 4.3, Listing 1.1 shows the four queries in the Esper query language for our scenario process model in Section 3. These queries are derived from the component tree as described in Sections 4.2 and 4.3. As not every activity is observable the derived queries are restricted to those events that are observable.

[2] This query depends on the transitions that are observable for a node. In our case, only enable, begin, and terminate are observable.

[3] Downloads, tutorials, and further information can be found at: `http://bpt.hpi.uni-potsdam.de/Public/EPP`

[4] Due to page limitations, we could not include this definition in the paper but provide it on our website at `http://bpt.hpi.uni-potsdam.de/Public/EPP#BPMN_Extension`.

Listing 1.1. Monitoring queries using Esper

```
St1:
SELECT *
FROM PATTERN [(EVERY S0=ContainerDocsReceived -> EVERY S1=ContainerReceived)]
WHERE SetUtils.isIntersectionNotEmpty({S0.ProcessInstances, S1.
    ProcessInstances})

Seq2:
SELECT *
FROM PATTERN [(EVERY S2=AskedForContainer -> EVERY S3=St1)]
WHERE SetUtils.isIntersectionNotEmpty({S2.ProcessInstances, S3.
    ProcessInstances})

Loop1:
SELECT *
FROM PATTERN [(EVERY S4=Seq2) UNTIL EVERY S5=MountContainer]
WHERE SetUtils.isIntersectionNotEmpty({S4.ProcessInstances, S5.
    ProcessInstances})

Seq1:
SELECT *
FROM PATTERN [(EVERY S6=ContainerLocReceived -> EVERY S7=ArrivedAtLocation ->
    EVERY S8=Loop1)]
WHERE SetUtils.isIntersectionNotEmpty({S6.ProcessInstances, S7.ProcessInstances
    , S8.ProcessInstances})
```

At first, the *St1* query monitors the sequence of two monitoring points with the event types of the `Get assigned container` activity. While the definition of the sequential ordering of the events for this query is enclosed in `PATTERN[...]` in the `FROM`-clause, the `WHERE` clause checks whether the events from both event types have occurred for the same process instance. In the *Seq2* query monitors whether the gateway event is followed by the previously defined query (resp. `Get assigned container` activity). Since activity `Check container` as well as the intermediate event `Container rejected` are not observable, the *Loop1* query can only check if *Seq2* is followed by events belonging to the *Mount container* activity. Finally, we can use the *Seq1* to check occurrences of events for process instances executing the process shown in Fig. 2.

It is possible to decouple the query creation from the monitoring part into a separate service module. In this vein, it is possible to generate queries that are independent from a specific EPL. Depending on the EPL used, it might require adaptations. In all cases, it is required to represent the dependencies between the queries and enable the checking of events through the graph of queries. For example, the termination of the *Loop1* query is required for the complete observation of the *Seq1* query and thus the whole process execution.

5 Application

In this section, we exemplarily address three areas of BPM the presented approach can be applied to: (i) monitoring of business process progress, (ii) monitoring of process model deviations, and (iii) calculation of Key Performance Indicators (KPIs).

(i) Through the usage of our approach, it is possible to correlate events in a CEP engine to the nodes of a process model. Therewith, the monitoring of a single process execution can be established. As per the introduced framework, the life cycles for single

process nodes and for the components of the process model are observable. Thus, a very detailed status of the execution progress for process instances can be presented. Recognizing an event at a PEMP will predict the actual state of a process execution and its performed activities. For example, when we see the events *Container location received* and *Arrived at location*, we can infer that the activity `Drive to container location` was fully performed.

(ii) Further, based on this monitoring information, deviations from the process model during runtime can be determined. In comparison to the approaches of [18] or [1], we do not create queries to detect deviations, but search for deviations on the basis of the execution status of a process instance. The usage of the component tree allows to determine order relations between the process nodes, which are similar to the order relations defined in [18]. Every time a query is triggered in our approach, a special monitoring component looks for execution deviations. By doing so, it is possible to detect nodes which should be exclusive but were observed together in the same process instance, nodes which should be in a strict sequential order but were monitored deviant to this order, nodes which should be present but were absent during runtime of the process instance, nodes which should occur only once but happened more often, and all execution deviations for nodes which are contained in a loop.

However, the detection of execution deviations for nodes that are part of a loop is limited, because exclusiveness, order, missing, or duplicate violations cannot be distinguished with certainty. For example, assuming activity `Check container` is observable allows to monitor both activities contained in the loop. In case, the trace *A,B,C,D,E,G,G,F*[5] is observed, a loop-deviation is detected. The deviation is monitored, because event *F*, indicating the container was mounted, was monitored only once, whereas the event *G*, indicating the performance of `Check container`, was observed twice. Thus, it is possible that the events *C,D,E* for the XOR join and the activity `Get assigned container` are missing for the second loop iteration or that the second `Check container` activity instance represented by the second event *G* is a duplicate.

(iii) Besides the application of our approach for process monitoring, we can utilize the events relating to a particular PEMP to measure KPIs. We refer to the definition of a KPI as stated in [19]. A KPI is linked to a business goal it is contributing to and has a name and a data type. The KPI definition includes an algorithm that describes how to measure the KPI, a target value, and upper and lower target margins. For KPI measurement, the particular PEMPs can be used in the corresponding algorithm. As described in Section 3, the terminal operator has the business goal to ensure a certain customer satisfaction that is influenced by the duration the drivers need to spend at the terminal to mount a container, for instance. Therefore, a KPI is defined that measures the time between the truck driver getting the information about the location of the assigned container to be mounted (start point of KPI measure) and the truck departure (end point of KPI measure).

[5] In accordance to Fig. 2: A is an event of the Event Object Type (EOT) *Container location received*, B is an event of EOT *Arrived at location*, C is an event of EOT *Asked for container*, D is an event of EOT *Container documents received*, E is an event of EOT *Container received*, F is an event of EOT *Container mounted*, and G is the newly introduced event of EOT *Container checked* for monitoring `Check container`.

Referring to the scenario described in Section 3, one can see that the KPI may be influenced by the loop. In case a container is rejected, because of an identified damage for instance, the start point of the KPI measure is passed again. This challenges the measurement of the KPI, because it has to be decided whether the KPI measurement is still valid (start point of the KPI measure is still the first occurrence of the event captured at the beginning of activity `Get assigned container`) or the start point of the KPI measure needs to be reset. This constellation is not trivial to handle, as we cannot observe the entrance into the loop cycle explicitly, because the activity `Check container` nor the event of rejecting the container can be observed. Thus, we cannot differentiate whether the loop was intended as described in the process model and the KPI measure needs to be reset, because the assignment of the container is not in the responsibility of the terminal, or there was another execution of activity `Get assigned container` by mistake or any other reason and the KPI measurement needs to be kept.

6 Related Work

Barros et al. [2] present a set of patterns describing relations and dependencies of events in business processes that have to be captured in process models to observe the overall process context. Their assessment of the modeling languages BPMN and Business Process Execution Language (BPEL) resulted in their language proposal called Business Event Modeling Notation (BEMN) [7], a graphical language for modeling composite events in business processes. BEMN allows to define event rules, e.g., specific combinations of events, that are to be used in stand-alone diagrams or as integration into BPMN. Similarly, Kunz et al. [12] introduce an approach to enhance the creation of CEP queries. In particular, the approach presents how EPL statements can graphically be represented by BPMN elements. In this way, the authors provide a means to model CEP queries with a better usability for business users. Both modeling approaches, in [7] and [12], focus on the representation of CEP in business processes. Complementary, our approach includes a standard-conform extension of BPMN with which we are able to automatically derive CEP queries from process models and not only check the process-flow but also life cycle transitions of nodes via events.

In [1], the authors introduce techniques to automatically generate Esper queries by taking a choreography model as a formalization of the process, however, without including the life cycles of nodes or basing the approach on a specific modeling language. Similar, Weidlich et al. [18] take BPMN models as basis to create EPL statements to monitor process violations only. Both approaches presume complete and structured event logs. Thus, they are not suited for processes that include non-observable events. In our approach the process model must be annotated with PEMPs that bind events to state transitions of BPMN elements as described in [11] first.

In the context of BPM, Dahanayake et al. [5] give an overview of Business Activity Monitoring (BAM) and introduce a four class-categorization of BAM systems all basing on events. Therefore, the approach presented in this paper can be applied to enable BAM techniques and methods to provide valuable monitoring results by using the produced extracted events as input.

7 Conclusion

We combined BPM with CEP to allow model-driven monitoring of business process executions in semi-automated environments. In essence, we can decompose a process model via a graph representation into a RPST, which we then transform into a component tree, which in turn is the basis to derive CEP queries determining the status of an execution. The constructs of a process model being considered for query generation are specified by the stakeholder by attaching PEMPs to nodes of the process model. This allows to specify the activities, events, and decisions to be observed in a process model to especially receive information about happenings the stakeholder is interested in and lowers the effort for creating those perticular queries manually. Further, the specification of PEMPs to nodes of a process model is implemented in an EPP allowing business users to do so without the need to know the technical specialties. In future work, we will apply this approach to process monitoring and analysis tasks in general, e.g., runtime or process cost analysis. Analyzing process event occurrences is another application area the approach can contribute to.

References

1. Baouab, A., Perrin, O., Godart, C.: An Optimized Derivation of Event Queries to Monitor Choreography Violations. In: Liu, C., Ludwig, H., Toumani, F., Yu, Q. (eds.) ICSOC 2012. LNCS, vol. 7636, pp. 222–236. Springer, Heidelberg (2012)
2. Barros, A., Decker, G., Grosskopf, A.: Complex events in business processes. In: Abramowicz, W. (ed.) BIS 2007. LNCS, vol. 4439, pp. 29–40. Springer, Heidelberg (2007)
3. Bernhardt, T., Vasseur, A.: Esper: Event stream processing and correlation. O'Reilly Media (2007), published at `http://onjava.com/`
4. Bülow, S., Backmann, M., Herzberg, N., Hille, T., Meyer, A., Ulm, B., Wong, T.Y., Weske, M.: Monitoring of Business Processes with Complex Event Processing. In: BPM Workshops. Springer (2013) (accepted for publication)
5. Dahanayake, A., Welke, R., Cavalheiro, G.: Improving the Understanding of BAM Technology for Real-time Decision Support. IJBIS 7(1), 1–26 (2011)
6. Daum, M., Götz, M., Domaschka, J.: Integrating CEP and BPM: how CEP realizes functional requirements of BPM applications (industry article). In: DEBS, pp. 157–166 (2012)
7. Decker, G., Grosskopf, A., Barros, A.: A graphical notation for modeling complex events in business processes. In: EDOC, pp. 27–36. IEEE (2007)
8. EsperTech: Esper - Complex Event Processing, `http://esper.codehaus.org` (as of May 2013)
9. Etzion, O., Niblett, P.: Event Processing in Action. Manning Publications Co. (2011)
10. Herzberg, N., Kunze, M., Rogge-Solti, A.: Towards Process Evaluation in Non-automated Process Execution Environments. In: Services and Their Composition, ZEUS (2012)
11. Herzberg, N., Meyer, A., Weske, M.: An Event Processing Platform for Business Process Management. In: EDOC. IEEE (2013) (accepted for publication)
12. Kunz, S., Fickinger, T., Prescher, J., Spengler, K.: Managing Complex Event Processes with Business Process Modeling Notation. In: Mendling, J., Weidlich, M., Weske, M. (eds.) BPMN 2010. LNBIP, vol. 67, pp. 78–90. Springer, Heidelberg (2010)
13. Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley (2002)
14. OMG: Business Process Model and Notation (BPMN), Version 2.0 (2011)

15. Rozsnyai, S., Lakshmanan, G.T., Muthusamy, V., Khalaf, R., Duftler, M.J.: Business Process Insight: An Approach and Platform for the Discovery and Analysis of End-to-End Business Processes. In: 2012 Annual of the SRII Global Conference (SRII), pp. 80–89. IEEE (2012)
16. van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M.: Business process management: A survey. In: van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M. (eds.) BPM 2003. LNCS, vol. 2678, pp. 1–12. Springer, Heidelberg (2003)
17. Vanhatalo, J., Völzer, H., Koehler, J.: The Refined Process Structure Tree. Data & Knowledge Engineering 68(9), 793–818 (2009)
18. Weidlich, M., Ziekow, H., Mendling, J., Günther, O., Weske, M., Desai, N.: Event-Based Monitoring of Process Execution Violations. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 182–198. Springer, Heidelberg (2011)
19. Weske, M.: Business Process Management: Concepts, Languages, Architectures, 2nd edn. Springer (2012)