24th International Conference on Business Information Systems (BIS 2021)
Smart Infrastructures
https://doi.org/10.52825/bis.v1i.51

Published: 02 July 2021

Execution of Multi-Perspective Declarative Process Models using Complex Event Processing

Niklas Ruhkamp[1], Stefan Schönig[1][https://orcid.org/0000-0002-7666-4482]

[1] Institute for Management Information Systems, University Regensburg, Germany

Abstract. The Internet of Things (IoT) enables continuous monitoring of phenomena based on sensing devices as well as analytics opportunities in smart environments. Complex Event Processing (CEP) comprises a set of techniques for making sense of the behavior of a monitored system by deriving higher level knowledge from lower level system events. Business Process Management (BPM) attempts to model processes and ensures that executed processes conform with a predefined sequence. In IoT scenarios frequently a large number of events has to be analyzed in real-time to allow an instant response. While BPM reaches its limits in such situations, CEP is able to analyze and process high volume streams of data in real-time. The evaluation and execution of rules and models of both paradigms are currently based on separate formalisms and are frequently implemented in heterogeneous systems. The presented paper integrates both domains by proposing an execution approach for multi-perspective declarative process process models completely based on CEP. The efficiency of the combined paradigms is validated in an implemented demonstration with simulated and real-life sensor data.

Keywords: Process Execution, Complex Event Processing, MP-Declare, Event-driven systems

The world is increasingly linked through a large number of connected devices, typically embedded in electrical/electronical components and equipped with sensors and actuators, that enable sensing, (re-)acting, collecting and exchanging data via various communication networks including the Internet of Things (IoT). As such, it enables continuous monitoring of phenomena based on sensing devices (wearable devices, beacons, smartphones, machine sensors, etc.) as well as analytics opportunities in smart environments (smart homes, connected cars, smart logistics, Industry 4.0, etc.) [1]. Event processing focuses on capturing and processing events in real-time, for detecting changes or trends indicating opportunities or problems. *Complex Event Processing* (CEP) comprises a set of techniques for making sense of the behavior of a monitored system by deriving higher level knowledge from lower level system events. CEP and *Business Process Management* (BPM) have traditionally been considered very separate from each other [2], [3]. BPM attempts to model processes and ensures that executed processes conform with a predefined sequence. In addition, BPM offers methods to analyze business processes and to search for potential improvements [4]. In complex situations a large number of events has to be analyzed in real-time to allow an instant response. While BPM reaches its limits in such situations, CEP is able to analyze and process high volume streams of data in real-time. To implement novel scenarios in the area of the IoT, e.g., Industry 4.0 and Smart Home scenarios, a combination of these two domains is becoming an active field of research under the term event-driven BPM [5]. Both domains provide their own advantages, which can complement each other [2]. Consider, e.g. an Industry 4.0 environment that produces a large amount

of raw data. Using CEP, this data can be processed efficiently. In the context of predictive maintenance the failure of a machine can be predicted long time in advance. The combination of CEP and BPM would not only allow to detect such a pattern using CEP but also to define how human operators have to react to such a failure prediction based on activities defined in the underlying process model.

The evaluation and execution of rules and models of both paradigms, however, are currently based on separate formalisms and are frequently implemented in heterogeneous systems. First attempts to combine both paradigms have already been done and are mentioned in related work. An integrated execution approach combining both worlds in one engine is still missing. We fill this research gap and integrate both domains by proposing an execution approach for business process models completely based on CEP. For this purpose, the multi-perspective extension of the declarative process modeling language Declare [6], MP-Declare [7] is used for mapping predefined constraints to CEP-queries, which are then executed by a CEP-engine. We implemented[1] our approach based on the *Esper*[2] CEP-engine and the corresponding Esper Query Language (EQL). Additionally, screencasts and videos demonstrate the real-life application of the approach using sensor data provided via MQTT[3].

The remainder of this paper is structured as follows: Section 1 describes fundamentals and related work. In Section 2.1 we introduce our approach to execute MP-Declare constraints using CEP queries. Section 2.2 describes the implementation of the integrated exeuction engine. The approach is validated with simulated and real data in Section 3 and Section 4 concludes the paper.

1 Background and Related Work

Next, we describe event-driven systems, basics of multi-perspective declarative process modelling and give an overview of related approaches.

1.1 Event-Driven Systems

Processes in our everyday life and business related procedures are influenced and triggered by various events. The processing, interpretation and reaction to such events is therefore an important part of how companies work. The three basic steps of event-driven systems are: *(i) Sense:* The starting point is the recognition of relevant information or facts by sources like sensors. This information is interpreted as events and reflects a relevant part of the state of reality. For event-driven systems the events must be recognized immediately at the time of their occurrence to guarantee real-time processing. Otherwise, the value of the information decreases. *(ii) Process:* In this step, the analysis of the detected events is performed. Events are aggregated, correlated, abstracted, classified, or if necessary discarded. Here, we seek for patterns in and between the event streams, which express certain relationships and dependencies between the events. *(iii) Respond:* If a pattern is recognized, the system can react with a corresponding action, e.g., warnings or the triggering of a business activity. The generation of new events is also a possible reaction, as for instance the generation of complex events on a higher level of abstraction.

As soon as data from event sources like sensors, network data, or news tickers arrive, they are processed by a CEP engine using predefined rules to detect patterns and derive complex events. This process can be repeated on several levels of abstraction. Subsequently, predefined actions are triggered or the obtained information is transferred to other systems, e.g., databases, message channels, or information systems. This way, processes are not only monitored but additionally automatic actions can be triggered [5].

[1] Available at `https://github.com/NiklasRuhkamp/MP-Declare-To-CEP`

[2] EsperTech - Esper Documentation: `https://www.espertech.com/esper/&sc=SUR`

[3] `https://vimeo.com/512049878`

1.2 Multi-Perspective Declarative Process Modelling

Declarative process modelling approaches like Declare [6] have proven to be suitable means to capture activities and agent interactions within flexible environments additionally involving real world objects [1]. A central shortcoming of the process modeling language Declare is the fact that constraints only apply to activities while other perspectives such as time and data perspectives are ignored. In real world scenarios like IoT applications these are important factors that must be considered to model realistic processes. To include these perspectives, Declare has been extended by a multi-perspective version called *MP-Declare* [7].

Semantics of Declare are extended by further conditions, which refer to the payload of events and must be *fulfilled* to satisfy a constraint namely the *activation condition*, the *correlation condition* and the *target condition*. These additional conditions will be demonstrated using the example of template *Response (A, B)* in the context of the IoT. With standard Declare, the constraint would be formulated as *Response (machine failure, order maintenance)*. Machine failure is the *activation*, while order maintenance is the *target*. This constraint defines that if a machine is down, the maintenance department must be informed at some point in the future to initiate the maintenance. With MP-Declare semantics additional conditions can now be added. The *activation condition* φ_a, in this case, is the fact that the machine is a production-critical one, whose failure would cause a standstill of production within minutes. If this condition does not occur while machine failure does, the constraint is not activated. This is formally written as $A \wedge \varphi_a(x)$ which means that when action *A* occurs, the condition φ_a must be true for *x*. The *correlation condition* φ_c is already part of the *target* and addresses the payload of both, event *A* and event *B*. Therefore, φ_c must be valid when the *target* arrives. It is formulated as $B \wedge \varphi_a(x,y)$. In this context, an example would be that if a machine from certain vendor is down, the maintenance department of the same vendor must be informed and not the one of another vendor. Additionally, there is the *target condition* φ_t which only refers to the payload of event *B* and is written as $\varphi_t(y)$. Depending on the use case a time period can also be defined in which the rule must be fulfilled [8]. The complete constraint would therefore be described as: if a machine essential for ongoing production is down, maintenance has to be initiated by the same vendor, which also has to be available for maintenance within the defined time frame. This process would ensure that the machine is repaired by the corresponding maintenance right after a problem is detected with minimal consequences for the production process.

1.3 Related Work

The use of a CEP engine to execute multi-perspective declarative process models is yet not well studied. The paper by Soffer et al. [2] is one of the most important sources for the combination of CEP and BPM in general, and specifically for concepts of integrating BPM and CEP. The paper provides a state-of-the-art review of current research of the symbiosis of CEP and BPM and describes challenges and opportunities. Janiesch et al. [9] are dealing with the combination of IoT and BPM in a research and practitioner's point of view. A general framework for event-driven BPM is presented in [10]. Li et al. [11] provide a translation of the block-structured part of Business Process Execution Language (BPEL) into events in order to be able to execute them using CEP. Although BPEL is not a modeling language, but rather an execution language, there is a mapping between BPMN and BPEL [12]. Cicekli and Cicekli [13] use an imperative process modeling language called control-flow-graph, which works with basic control flow patterns. To execute them and increase their expressiveness, they provide corresponding event rules. Another attempt to realize event-driven systems is done by RuleML [14] using rule detection to trigger processes. Hens et al. [15] use imperative languages such as BPMN and YAWL and divide them into small chunks. The start and completion of these are then processed by a CEP-engine. However, this cannot be seen as a complete execution on the CEP-engine since the individual chunks are still handled by the process engine. In the work of Daum et al. [16], they investigate the integration of BPM and CEP. However, they investigate how process models can be supported or extended through CEP, but not the execution of these models on a

CEP engine. Another approach of integrating external events into business models is done by Mandal et al. [17] combining a process engine and a event engine in a heterogeneous system. There exist also first approaches for the execution of declarative rules using CEP. Jergler et al. [18] propose a version of the Guard-Stage-Milestone model (GSM) based on CEP to specify life cycle processes of business artifacts. This model contains Event-Condition-Action rules (ECA-rules), which can be executed by a CEP engine. Soffer [19] also suggests an ECA-based execution of declarative models. Approaches to detect declarative process models from event-logs were devised, such as in [20]. According to [21], some works use process stream mining to do so. In [22], Schönig et al. introduce that SQL can be used to derive multi-perspective declarative models from logs. In fact, these approaches are using static event logs and do not process the data as a stream. Therefore, the latency between occurrence of a constraint and its detection is not acceptable for time critical cases. Another problem could be the storage requirements, if processing the data in real-time is not possible. Burattin et al. [23] propose a framework for the discovery of declarative process models. They combine algorithms for stream mining and algorithms for the online discovery of Declare models to get a real-time representation of the process. Another attempt in this direction is proposed by [24] using Hoeffding trees. A similar approach is presented in [25] by examining event streams to process models using prefix-alignments.

In the context of Big Data traditional approaches may reach their limits due to the high data volume. Therefore, a mapping to CEP, which is geared towards Big Data, could perform more efficiently. Wu et al. [26] are using SASE [27] over streams of RFID-events in order to detect matching patterns and to feed an external monitoring application. Another work dealing with monitoring in combination with CEP is introduced by [28]. In this approach CEP is used in combination with Business Activity Monitoring (BAM) to monitor cloud BPM.

None of these approaches deals with the execution of multi-perspective declarative models itself. A first attempt was made in [29]. In this work, MP-Declare constraints are transformed into the modelling language Alloy and executed afterwards. In summary, the execution of complete multi-perspective declarative process models via CEP is still unexplored. The motivation of the work at hand is to study and implement a solution which integrates MP-Declare process models entirely into CEP and thereby bridging the gap between these two paradigms.

2 Execution of MP-Declare Models using CEP

This chapter explains how multi-perspective declarative MP-Declare models can be executed on top of a CEP-engine. Therefore, we introduce the concept of how MP-Declare models can be mapped to CEP queries.

2.1 Concept

MP-Declare constraints essentially consist of two components. The *activation* of the constraint is on the left-hand-side of the constraint. As soon as it occurs, the constraint is *activated*. In addition to this, the *activation condition* must also be *fulfilled*. On the right-hand-side is the *target* including the *correlation condition* and the *target condition*, which must also occur to *fulfill* the constraint. A time period within the *target* must occur can be specified additionally. Therefore, a concept is needed for applying CEP rules to examine a stream of data for the *fulfillment* or *violation* of MP-Declare constraints and thus to execute entire MP-Declare process models by means of a CEP-engine.

CEP is able to recognize patterns in an event stream and react to them. Furthermore, incoming low-level event streams can be filtered by CEP and, if necessary, depending on the application, a new stream of events can be generated including the relevant data. This feature is now used to apply MP-Declare constraints to incoming event streams. The basic idea here is to detect the left-hand-side of the constraint - the *activation* - using CEP. As soon as event of

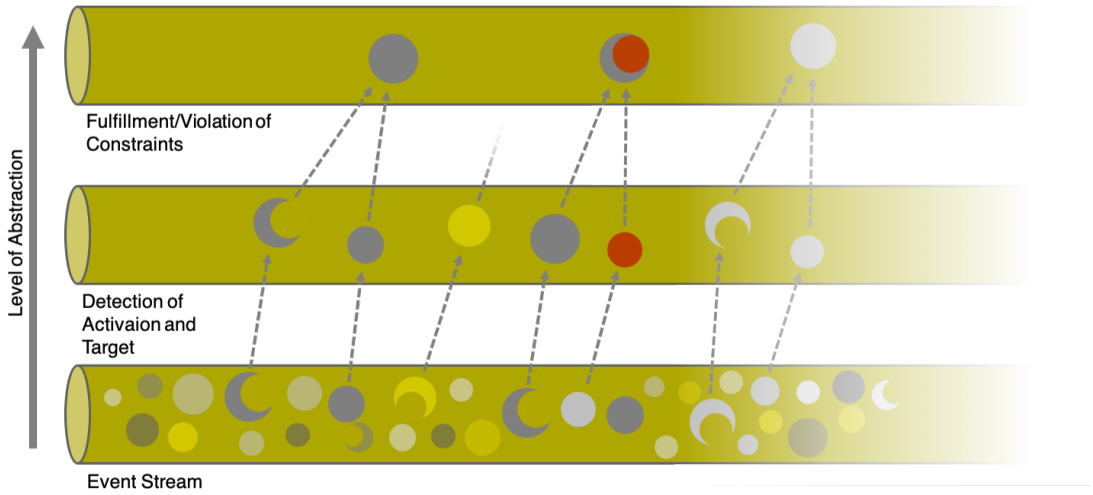

Figure 1. Recognition of Activation and Target in Event Streams of different Levels of Abstraction

a new process instance occur in the event stream and an *activation* of a constraint has been detected, the right-hand-side of the constraint - the *target* - must arrive. Thus, CEP must be able to store all events of process instances where the *activation* of a constraint occurred, and then to process the event stream to see if the *target* occurs as well. In this case, this instance *fulfills* the constraint. If the *target* is not found in the event stream, this process instance *violates* the rule or can not *fulfill* the rule so far.

This procedure is illustrated in Figure 1. On the lowest level of abstraction the incoming event streams can be seen. These are completely unordered, unfiltered and from different data sources. The CEP-engine accesses this stream and examines it for incoming *activations*. The engine has to store all *activated* constraints until they are *fulfilled* or *violated*. To implement this concept, streams of different abstraction levels are used as shown in Figure 1. *Activations* are stored on an intermediate level of abstraction. On this level also *targets* and *violations* are represented. Once an *activation* occurs, the CEP-engine searches for the corresponding patterns in the event stream to find the *target*. As soon as it arrives, the *target* is also added to the higher-level stream. The intermediate level stream represents all available *activations* and *targets*. Finally, at the highest abstraction level stream, the CEP-engine checks whether the constraints have been *fulfilled* or *violated*. For this purpose, it examines all *activations* and checks if the intermediate stream contains the matching *targets* or a *violation*. Additionally, the CEP engine examines the payload of the events to determine whether required conditions are met. As soon as a constraint can be detected as *fulfilled* or *violated*, it is added to the highest-level stream. As shown as the dark grey event in Figure 1, both the *activation* and the *target* are in the intermediate stream and hence the constraint is *fulfilled*. As shown as the green event, only the *activation* occurs without the matching *target*. The constraint is not yet *fulfilled* but might be in the future. This enables CEP to examine the event stream for CEP constraints and thus execute the entire MP-Declare models in form of a set of constraints using the CEP engine.

Besides the time perspective (whether the *target* has to occur before or after the *activation*), the constraints can be divided into three categories. The first group is constraints that can only be *fulfilled* or are not yet *fulfilled* but not directly *violated*. These consist of an *activation* and a *target*. The time period in which the *target* has to appear is potentially infinite.

The second group of constraints are those which can be *fulfilled* or can be *violated* directly. The *target B* of *alternate response*, for example, must occur without other instances of event *B* in between. Therefore, the constraint is *violated*, as soon as another *B* occurs between *A* and *B*.

The negating constraints, which are marked by the prefix *"not"* belong to the third category. These are *violated* as soon as the *activation* of a constraint is followed by the corresponding *target*, which according to the constraint must not occur. As soon as both sides of a *"not"* constraint occur, this rule is *violated*.

For constraints of the first category, however, the constraint is *fulfilled* if both sides occur and are not yet *fulfilled* when the *activation* occurs but not (yet) the *target*. In brief, the engine waits for the *fulfillment* of activated constraints. The procedure for the second category is different. The CEP engine does not have to wait to see whether the constraint is *fulfilled* at some point in the future. The engine does not only search for the *fulfillment* but also for *violations* of activated constraints by looking for occurring of the opposite of the constraints.

To highlight this behavior, Figure 1 illustrates the detection of a constraint of the second category, using the example of *Chain Response* defined as: if *A* occurs, *B* must occur next, which in turn means no other events must occur in between. In the event stream the matching *target* occurs after *activation*, but not immediately after its *activation*. The next event after the *activation* is the event marked in red. As a result the CEP-engine can mark the rule as *violated*. Using multiple streams at different levels of abstraction, MP-Declare models can entirely be mapped to CEP rules and executed using CEP-engine.

2.2 Implementation

For the execution of MP-Declare models by means of CEP, the CEP-Engine Esper is used in this work. The CEP-engine works like an inverted database. In a conventional database, the data sets are fixed, and the data is accessed dynamically using a query language. However, with a CEP-engine the query rules are known from the beginning, and the data is then loaded in the form of an event stream and checked for the rules in real time. Incoming event streams can be read via input adapters and connectors. These streams are characterized by very high volume, fast emergence of new data and occurrence in real-time. Esper also provides access to historical data in memory to provide querying possibility on historical events. The engine manages the registered statements, examines the streams for these statements and stores the results in the form of Plain Old Java Objects (POJO). These are then available for downstream systems via the output adapter.

2.2.1 Transforming MP-Declare to EQL

Esper provides its own SQL-like Event Processing Language (EPL) ccalled Event Query Language (EQL). Here, queries essentially consist of SELECT-, FROM- and WHERE-constructs. EQL-queries are used to examine the event-stream for incoming *activations*, *targets* and *violations*. The EQL-query to detect, for instance, the target of a *Response* constraint in the context of machine failures and the order of a maintenance looks as follows:

```
SELECT a.id, b.company FROM PATTERN[
every a = MachineFailureEvent(productionCritical = true)
-> b= OrderMaintenanceEvent(available = true)]
WHERE a.manufacturer = b.company
```

The *PATTERN* defines that all cases, in which a production critical machine fails, *followed by* a maintenance order for which the company must be available at that time, are to be considered. The term *"every"* defines that the query should not be made only once, but all instances which *fulfill* this pattern should be returned. The WHERE clause checks the *correlation condition*. In case of backward-looking constraints like *Precedence* the EQL-query looks similar to the query of *Response*. The difference is that the *target* is on the left-hand-side and the *activation* is on the right-hand-side. Therefore, the engine is storing all potential *targets* of a backward-looking constraint as long as the corresponding *activation* follows and the constraint can finally be detected as *fulfilled*. As depicted in Figure 1 the engine has to search for *violations* in some

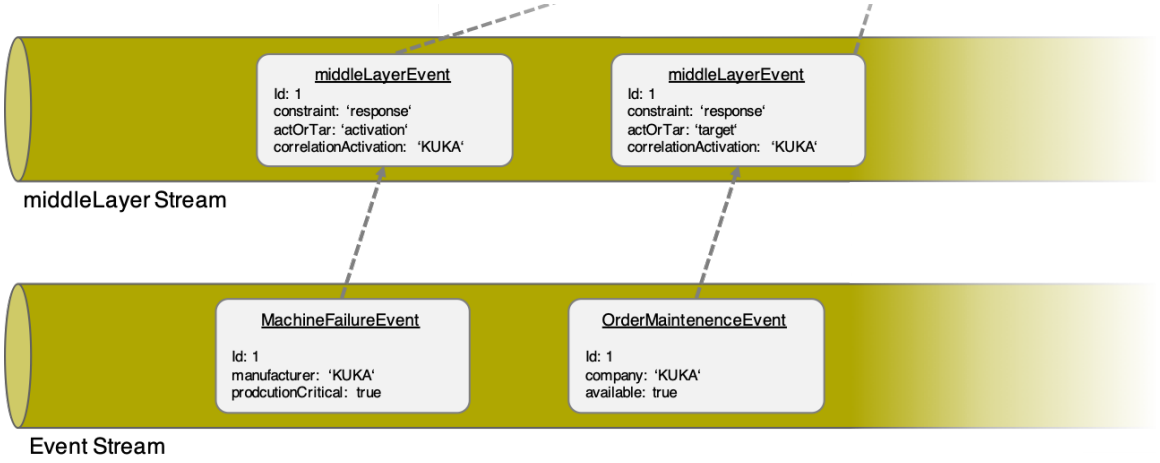

Figure 2. Example of events in the middleLayer-stream to fulfill Response

cases, too.

2.2.2 Constraint Builder

The most important component is the *ConstraintBuilder*. In order to execute MP-Declare rules using CEP, these must be translated into EQL-queries for the engine to be able to process them. The *ConstraintsBuilder* is composed of a for-loop, which iterates over all added constraints in the *constraintAndConditionList*, handed over by the *ConstraintScreen*. After all required contents are assigned, an if-else-statement is used to search for the suitable constraint type since each constraint type has different requirements.

2.2.3 middleLayer-Stream

As mentioned before (cf. Section 2.1) and shown in Figure 1 an additional stream is used, called *"middleLayer"* and is necessary to handle *activations*, *targets* and *violations* on a higher level of abstraction. This stream has four properties: The first one (integer *"id"*) is used to store the identifier of each event in the *middleLayer*-stream. This enables the engine to handle the different instances of processes. If the process does not have multiple instances, the identifier remains at its default value null. The second property (string *"constraint"*) is storing the type of the constraint. As soon as the user defines a specific name for a constraint, this name will be used instead. This allows handling several constraints of the same type. For example: if two different constraints of the type *Response* were defined, it might happen, that a *target* of the second constraint is assigned to the *activation* of the first one. In this case the first *Response* would mistakenly be *fulfilled*. To prevent this from happening, unique names need to be set. To recognize whether an event of the *middleLayer*-stream is an *activation*, a *target* or a *violation*, the third property (string *"actOrTar"*) is used. The last property (string *"correlationActivation"* solves a similar problem as the second one and is only used if the constraint includes a *correlation condition*. It ensures that, for example, the activating *MachineFailureEvent* of a "KUKA"-machine, can only be followed by an *OrderMaintenanceEvent* for which the company is also "KUKA" to solve the *Response* constraint. An illustration of the *middleLayer-Stream* is shown in Figure 2.

2.2.4 Assignment of Process Instances

It is sometimes important to process events context-dependent. For example, if a machine fails and needs to be repaired, it is important to check the event stream of the maintenance orders for an order that was placed for that particular machine and not for another one, to ensure the maintenance of the correct machine. By using the *CREATE CONTEXT* clause, Esper divides events into context partitions.

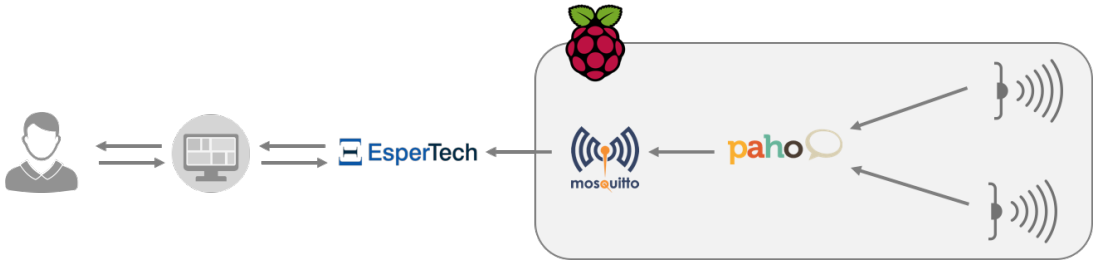

Figure 3. Environment to simulate a MQTT-Stream

3 Evaluation

We extensively evaluated our approach in terms of functionality and performance. In order to validate the presented approach events were first sent to the CEP-engine from the Java-environment in a predefined sequence. The simulation of the sample events is implemented on a new thread. During the validation, 84 different iterations were made. Each constraint was tested in seven different variations. The tool was able to detect all *activations*, *violations* and *fulfillments* of each constraint prooving the funtionality and correctness. In addition, the latency between the occurrence of an event and the assignment of a *violation* or *fulfillment* was measured, to check the performance in terms of efficiency of the tool. The gap between the occurrence of an event and the detection of the constraint, is used as a measure of latency. According to the result of this measurement, the latency was only 1 msec. If the event does not only affect one but several different *activations*, the latency increases. The maximum latency in this measurement was 4 msec. On average, the latency was 1.91 msec. A longer period of time is to be expected for constraints like *Precedence*, where the *violation* can not be detected directly, which is therefore not included in the latency validation. The engine is waiting for 1 second, if the *activation* is followed by the detection of a *target* within this time. If not, the constraint is *violated*. Therefore, the highest possible latency for detection is 1 second. The results of the validation show that our tool is efficient at the detection of MP-Declare constraints while achieving a 100 percent success rate. Additionally, we evaluated our approach with a real-life approach. Here, an IoT setup was simulated as illustrated in Figure 3. An ultrasonic distance sensor and a push-button matrix are connected to a Raspberry Pi 4B. The sensor data was sent via MQTT (Message Queuing Telemetry Transport) to the tool, running on another machine. MQTT is a lightweight, publish/subscribe messaging transport protocol, often used in the context of IoT. The sensors are connected to the GPIO-board of the Raspberry Pi. The Raspberry Pi is also used as an MQTT message broker, using Eclipse Mosquitto. The data of the sensors are read with Python and published to the MQTT broker, using Paho. A screencast and video demonstrating the application and functionality are available online.[4]. It has proven that constraints involving real-life sensor data can be sent to the engine via MQTT and are processed correctly by the engine.

4 Conclusion and Future Work

Our approach presents an efficient, scalable and reliable approach to examine a stream for MP-Declare constraints in real-time, using complex event processing. The CEP-engine Esper has been implemented and adapted such that it is able to detect *activations*, *fulfillments* and *violations*. For this purpose, besides the low-level event streams, different streams of a higher level of abstraction are used to store *activations* and to listen for following *targets* or *violations*. This way, MP-Declare constraints are executable even for unstructured high-volume streams of events. The graphical user interface offers an intuitive and easy way to formulate MP-Declare constraints. The user has full flexibility in adding new constraints. A screencast

[4] https://vimeo.com/512049878

available online[5] demonstrates the functionality of the tool. After the process has been started, the engine is giving real-time feedback to the user. This work provides an implementation of the discussed approach, to successfully and quickly execute MP-Declare process models. The tool can also easily be extended by predefined actions, which should apply as soon as a constraint is *violated*. Therefore, the system can easily be implemented to automatically detect whether predefined restrictions have been *violated* and to initiate necessary reactions. The validation has proven that this approach is highly reliable while achieving low latency. As future work, another validation in a more realistic environment is recommended. Since CEP is designed for Big Data and implemented to process huge amounts of data in real-time, it is expected that such integration is going to be successful. The integration into the context of IoT-environments like Industry 4.0 should demonstrate, that even multiple large streams can successfully be integrated and examined.

References

[1] G. Meroni, C. D. Ciccio, and J. Mendling, "An artifact-driven approach to monitor business processes through real-world objects," in *ICSOC*, vol. 10601, 2017, pp. 297–313.

[2] P. Soffer *et al.*, "From event streams to process models and back: Challenges and opportunities," Information Systems, vol. 81.2019, pp. 181–200, 2019.

[3] S. Schönig, L. Ackermann, S. Jablonski, and A. Ermer, "Iot meets BPM: a bidirectional communication architecture for iot-aware process execution," *Softw. Syst. Model.*, vol. 19, no. 6, pp. 1443–1459, 2020.

[4] D. M. Goetz, "Integration of business process management and complex event processing," 2010.

[5] R. Bruns and J. Dunkel, *Complex Event Processing*. 2015, ISBN: 978-3-658-09898-8.

[6] W. M. P. van der Aalst, M. Pesic, and H. Schonenberg, "Declarative workflows: Balancing between flexibility and support," *Comput. Sci. Res. Dev.*, vol. 23, no. 2, pp. 99–113, 2009.

[7] A. Burattin *et al.*, "Conformance checking based on multi-perspective declarative process models," *Expert Syst. Appl.*, vol. 65, pp. 194–211, 2016.

[8] H. van der Aa, K. J. Balder, F. M. Maggi, and A. Nolte, "Say it in your own words: Defining declarative process models using speech recognition," in *BPM Forum*, vol. 392, 2020, pp. 51–67.

[9] C. Janiesch *et al.*, "The internet of things meets business process management: A manifesto," *IEEE Systems, Man, and Cybernetics Magazine*, vol. 6, no. 4, pp. 34–44, 2020. DOI: 10.1109/MSMC.2020.3003135.

[10] R. von Ammon, C. Emmersberger, T. Greiner, F. Springer, and C. Wolff, "Event-driven business process management," in *Fast Abstract, Second International Conference on Distributed Event-Based Systems, DEBS 2008, Rom, Juli 2008*, 2008. [Online]. Available: https://epub.uni-regensburg.de/6829/.

[11] G. Li, V. Muthusamy, and H.-A. Jacobsen, "A distributed service-oriented architecture for business process execution," *ACM Trans. Web*, vol. 4, no. 1, 2010.

[12] M. Weidlich, G. Decker, A. Großkopf, and M. Weske, "BPEL to BPMN: the myth of a straight-forward mapping," in *OTM Confederated International Conferences*, vol. 5331, 2008, pp. 265–282.

[13] "Formalizing the specification and execution of workflows using the event calculus," *Information Sciences*, vol. 176, no. 15, pp. 2227–2267, 2006.

[5] https://vimeo.com/512048601

[14] A. Paschke, "The reaction ruleml classification of the event / action / state processing and reasoning space," *CoRR*, vol. abs/cs/0611047, 2006. arXiv: cs/0611047. [Online]. Available: http://arxiv.org/abs/cs/0611047.

[15] P. Hens, M. Snoeck, G. Poels, and M. D. Backer, "Process fragmentation, distribution and execution using an event-based interaction scheme," *J. Syst. Softw.*, vol. 89, pp. 170–192, 2014.

[16] M. Daum, M. Götz, and J. Domaschka, "Integrating cep and bpm: How cep realizes functional requirements of bpm applications (industry article)," in *International Conference on Distributed Event-Based Systems*, 2012, pp. 157–166.

[17] S. Mandal, M. Hewelt, and M. Weske, "A framework for integrating real-world events and business processes in an iot environment," in *OTM 2017 Conferences*, 2017, pp. 194–212.

[18] M. Jergler, H.-A. Jacobsen, M. Sadoghi, R. Hull, and R. Vaculın, "Safe distribution and parallel execution of data-centric workflows over the publish/subscribe abstraction," in *ICDE*, 2016, pp. 1498–1499.

[19] P. Soffer, "A state-based intention driven declarative process model," *International Journal of Information System Modeling and Design*, vol. 4, pp. 44–64, 2013.

[20] F. M. Maggi, M. Montali, and U. Bhat, "Compliance monitoring of multi-perspective declarative process models," in *EDOC*, 2019, pp. 151–160.

[21] W. M. P. van der Aalst, *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. 2011, ISBN: 3642193447.

[22] S. Schönig, C. D. Ciccio, F. M. Maggi, and J. Mendling, "Discovery of multi-perspective declarative process models," in *ICSOC*, vol. 9936, 2016, pp. 87–103.

[23] A. Burattin, M. Cimitile, F. M. Maggi, and A. Sperduti, "Online discovery of declarative process models from event streams," *Transactions on Services Computing*, vol. 8, no. 6, pp. 833–846, 2015.

[24] N. Navarin, M. Cambiaso, A. Burattin, F. M. Maggi, L. Oneto, and A. Sperduti, "Towards online discovery of data-aware declarative process models from event streams," in *IJCNN*, 2020, pp. 1–8.

[25] S. J. van Zelst, A. Bolt, M. Hassani, B. F. van Dongen, and W. M. P. van der Aalst, "Online conformance checking: Relating event streams to process models using prefix-alignments," *Int. J. Data Sci. Anal.*, vol. 8, no. 3, pp. 269–284, 2019.

[26] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams," in *International Conference on Management of Data6*, 2006, pp. 407–418.

[27] University of Massachusetts, *SASE Home*, http://sase.cs.umass.edu&sc=SUR, Online; accessed 20 December 2020.

[28] J. N. Martínez Garro, P. Bazán, and F. J. Díaz, "Using bam and cep for process monitoring in cloud bpm," *Computer Science and Technology*, vol. 16, no. 01, 2016.

[29] L. Ackermann, S. Schönig, S. Petter, N. Schützenmeier, and S. Jablonski, "Execution of multi-perspective declarative process models," in *OTM Conferences*, vol. 11230, 2018, pp. 154–172.