One engine to rule them all: Towards executing declarative processes with CEP

Leo Poss[1,*], Stefan Schönig[1]

[1]University of Regensburg, 93040 Regensburg, Germany

Abstract

We present a novel tool that directly executes MP-Declare process models on standard Complex Event Processing (CEP) engines, removing the need for middleware. By tightly integrating Business Process Management (BPM) and CEP, it supports flexible, declarative process execution over high-frequency IoT data. Its core innovation is a multi-level event abstraction framework that translates declarative constraints into optimized, executable CEP queries. This demonstrates how a unified engine paradigm simplifies architecture, significantly reduces latency, and enables the real-time enactment of event-driven processes, effectively bridging the abstraction gap between low-level events and high-level business logic and offering a scalable, responsive solution for process-aware systems.

Keywords

Declarative Process Execution, Complex Event Processing, Business Process Management, Event-Driven Architecture

1. Introduction

Business Process Management (BPM) plays a central role in how modern organizations create and deliver value. Increasingly, BPM must integrate with real-time data streams from the Internet of Things (IoT) and leverage capabilities of Complex Event Processing (CEP) to remain adaptive and responsive [1]. Event-Driven Architectures (EDA) promise to address this need by enabling systems to respond dynamically to external data. However, the high-frequency and low-level semantics of IoT events pose a major challenge: bridging the abstraction gap between raw sensor data and high-level business activities [2]. Current BPM solutions often rely on intermediate middleware layers to pre-process and transform event data before it can be used in process logic [3]. This architectural fragmentation leads to increased complexity, higher latency, and missed opportunities to leverage CEP's strengths, such as pattern detection for process execution. Particularly underutilized is the potential of CEP to directly enforce constraints from declarative process models, which offer flexibility and robustness in dynamic environments [4] but remain difficult to operationalize. This demonstration presents a novel tool that unifies these two paradigms: It enables the *direct execution of declarative process models*, specifically MP-Declare, on a standard CEP engine. The tool introduces a multi-level event abstraction framework, enabling the expression and execution of constraint activation, monitoring, and fulfillment as CEP queries. This approach eliminates the need for separate BPM systems or preprocessing pipelines, simplifying architecture and enabling true event-driven process execution over high-frequency data. This demo paper showcases an implemented tool and illustrates its use through technical validation and an industrial use case. It makes a concrete contribution to the BPM field by offering a scalable, flexible, and executable architecture for real-time, event-driven process management.

Cooperative Information Systems – Early Research Achievement and Demos, 2025

*Corresponding author.

leo.poss@ur.de (L. Poss); stefan.schoenig@ur.de (S. Schönig)

0009-0009-9221-8704 (L. Poss); 0000-0002-7666-4482 (S. Schönig)

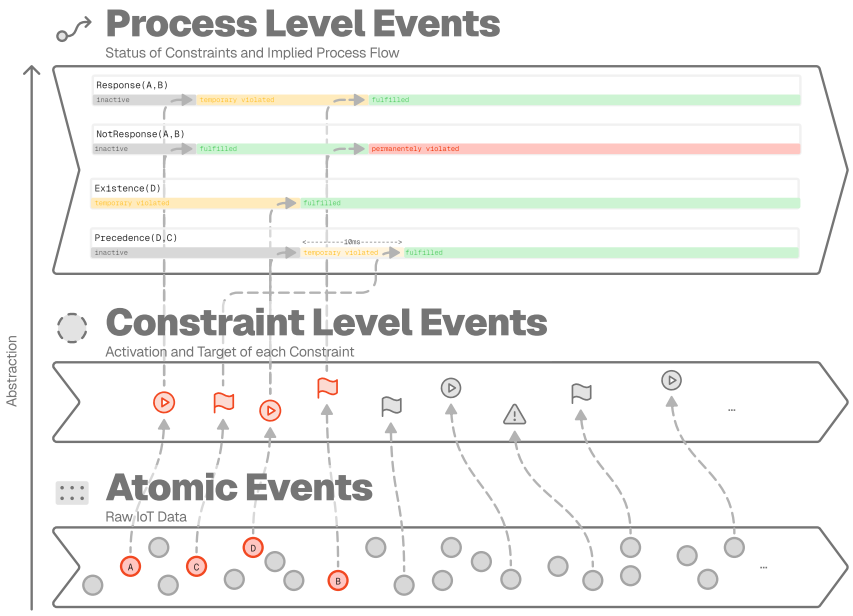

Figure 1: Overview of different abstraction layers and events

2. Innovation and Key Features

This demonstration introduces a novel tool that operationalizes declarative process models directly on a standard CEP engine. The key innovation lies in its ability to evaluate MP-Declare constraints over high-frequency data streams without relying on middleware or external BPM engines. This addresses a long-standing challenge in BPM research: closing the semantic and architectural gap between low-level IoT events and high-level process logic. The tool's architecture is based on a three-layer abstraction model that transforms raw events into semantically enriched process states (cf. Figure 1).

Declarative constraints, defined in linear temporal logic (LTL_f), are decomposed into activation-target pairs and mapped to CEP queries. The tool supports a broad range of MP-Declare templates, including forward-, backward-, and hybrid-looking constraints such as Response, Precedence, and RespondedExistence [4]. For this, we classify them by lifecycle behavior (e.g., runtime monitoring, instant fulfillment, unrecoverable violation) and support data conditions directly in CEP queries using MP-Declare payloads. In summary, this tool introduces the first fully integrated execution environment for declarative BPM that: *(i)* Unifies event processing and process execution in a single CEP engine, *(ii)* Provides native support for MP-Declare semantics through reusable query patterns, *(iii)* Operates without middleware, enabling true real-time responsiveness in IoT-rich scenarios. This approach provides a significant architectural simplification for process-aware information systems, demonstrating the practical applicability of declarative BPM in high-frequency, event-driven environments. It is essential to note that by unifying these paradigms, our tool deliberately sacrifices the comprehensive feature sets of traditional BPM suites (e.g., advanced graphical modelers, user management) in favor of a lightweight, highly responsive, and streamlined execution environment that focuses solely on declarative logic.

3. Implementation

CEP engines differ fundamentally in their approaches to event and query definition: Apache Flink uses functional APIs with embedded pattern definitions. At the same time, Esper's SQL-like EPL provides a clear separation between pattern logic (the *what*) and execution mechanics (the *how*), enabling dynamic query construction and deployment. While the presented examples primarily use pattern matching to detect event sequences, the full potential of the CEP engine can be further realized by integrating advanced concepts for event detection. For instance, aggregations could enforce cardinality-based

```
INSERT INTO constraintStatus
SELECT id, 'Resp' as name, 'ACTIVATION' as type
FROM GenericEvent WHERE eventType = 'OrderReceived'

INSERT INTO constraintStatus
SELECT id, 'Resp' as name, 'TARGET' as type
FROM GenericEvent WHERE eventType = 'PaymentProcessed'

SELECT a.id, a.name, a.type
FROM PATTERN [every a=constraintStatus(type='ACTIVATION', name='Resp') -> b=constraintStatus(type='TARGET',
↪  name='Resp')]
```

Listing 1: Query to detect RESPONSE activation and target and insert fulfillment `constraintStatus` event.

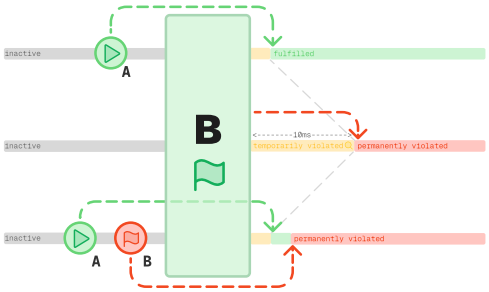

Figure 2: Rule detection of ALTERNATEPRECEDENCE

constraints (e.g., requiring three approval events), while joins could correlate data from different event streams before evaluating a constraint. This enables the definition of more expressive, context-aware declarative processes directly within the unified architecture.

Exemplary constraints Before describing the implementation, we present two exemplary constraints and their implementation specifics. Next to runtime constraints, the simplest constraint is detecting RESPONSE(A, B), which requires detecting the *activation* A and the *target* B (*Atomic Events*, see Figure 1). Based on these two events (*Constraint Level Events*), we can derive the current status of the constraint (*Process Level Event*). As soon as we detect the *activation*, the constraint is 'temporarily violated.' If we detect an *activation* followed by a *target*, we can set it to 'fulfilled,' as the response constraint is defined in Declare. Taking the example from above RESPONSE(*OrderReceived*, *PaymentProcessed*), this means detecting an event of type *OrderReceived* that triggers an event with type *activation* and a unique constraint identifier, as well as a similar event with role *target* shown in Listing 1. Additionally, we listen for these two events at a higher abstraction level, where we update the constraint state to 'temporarily violated' if we detect the *activation* and to 'fulfilled' if we detect the complex event described by *activation* followed by a corresponding *target*.

A complex constraint like ALTERNATEPRECEDENCE(*ShippingLabelGenerated*, *AuthorizationGranted*) could be part of the same ordering process, where a return approval allows the generation of exactly one return shipping label. If the customer loses the label and needs another one, triggering the "generate label" event again after the initial approval is disallowed. In addition to detecting *activation* and *target*, we must also detect a possible violation and update the constraint status from the middle layer accordingly. To detect precedence, after the *activation* is detected and the constraint is set to 'temporarily violated,' we must retrospectively look for a possible *target* that has already occurred, which is done by reversing the events in Listing 1. As this would also apply to future events, we need to manually create an event that violates this query if the precedence is not detected within a short timeframe. These possibilities are illustrated in Figure 2: if we detect the *target*, the constraint is 'temporarily violated,' as we are unsure whether we already detected a matching *target* or the constraint must be set 'permanently violated.' If we have detected a *target*, the constraint is 'fulfilled'; if not, it is 'permanently violated.' Finally, if we detect another *target* event in between, the constraint is also set to 'permanently violated.'

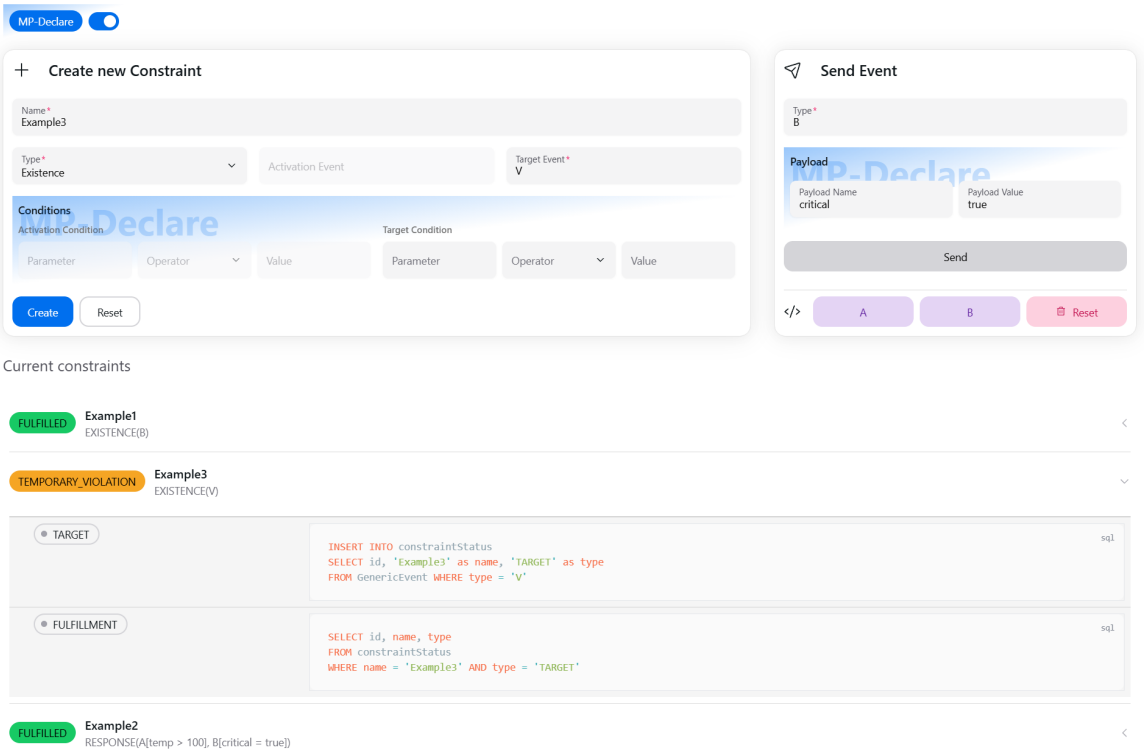

Figure 3: Implementation with creation of constraints, sending events with MP-Declare, and the current constraint states and event queries

Implementation For executing declarative process models and high-frequency data, possibly from IoT devices, we present an implementation based on standard web technologies (Quarkus[1], Next.js[2]) and Esper[3] as a CEP engine that includes the queries and constraints mentioned above. Using Esper enables us to easily add new queries at runtime and to simplify the creation of string-based constraints and the corresponding queries. The implementation[4] consists of two main components: the frontend, which is required for user interaction, and the backend, which handles constraints and the actual execution of the model. The frontend as shown in Figure 3 contains three different sections: one for creating new constraints, i.e., modeling the process, top left, and injecting or sending events (in practice, this would include both, actors finishing their respective tasks, and high-frequency IoT data), top right[5]. Here, we can toggle MP-Declare components for activation, target conditions, and event payloads. This is followed by an overview of currently deployed constraints and their events. For example, for Response(A, B[`critical = true`]) this shows EPL queries for detecting the *activation* and *target* for the constraint (inserting a new event of higher abstraction or semantic meaning into `constraintStatus`), as well as updating the status to 'temporary violation' and 'fulfillment' (querying complex events from `constraintStatus`). The backend provides RESTful endpoints for creating and querying constraints, as well as for receiving and processing events using the embedded CEP Engine.

4. Evaluation and Use Case

The initial technical evaluation of our tool focuses on two critical dimensions: functional correctness and integrity of the abstraction layer. Each constraint type was evaluated against formal definitions from literature to ensure semantic fidelity and functionality [4]. This involved defining test cases

[1]https://quarkus.io/
[2]https://nextjs.org/
[3]https://www.espertech.com/esper/
[4]Repository with installation instructions: https://github.com/LeoPoss/quarkusCEP/tree/base
[5]Guided screencast of the tool: https://youtu.be/2qUTwzLKCxM.

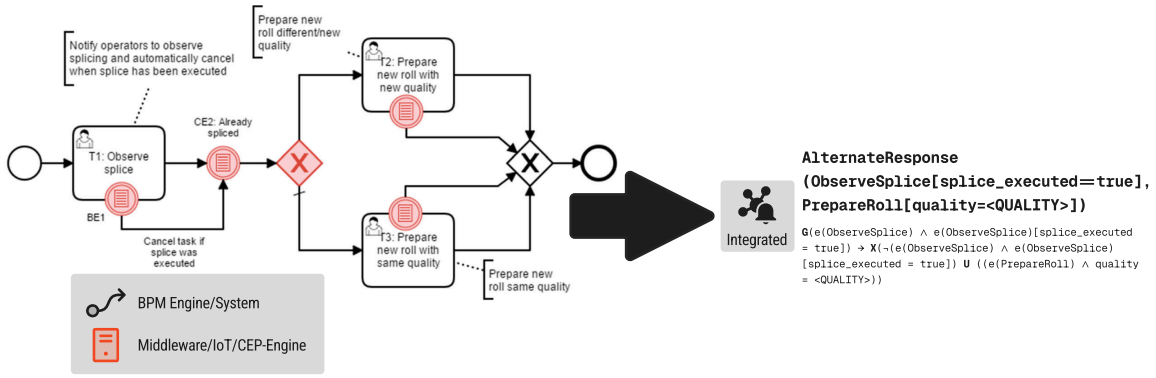

Figure 4: Process snippet [2] and resulting MP-declare constraint

with synthetic event sequences and verifying that the CEP engine produced the expected outcomes. We verified the coherence and integrity of the information flow across the three abstraction layers, particularly in the propagation of events between layers (Atomic, Constraint Level, Process Level) and the management of constraint states throughout process execution.

This technical validation provides evidence that the synergistic integration is coherent by demonstrating that declarative constraints can be transformed into event-based representations while preserving semantic integrity across temporal relationship categories and constraint state types.

Use Case Application: Industry Process Our tool solves real-world process management challenges by seamlessly integrating declarative constraints and event-based monitoring. We do this by migrating a real-world process model from [2] to our approach using a single constraint. The imperative process model and the resulting declarative version are shown in Figure 4. The version presented in [2] consists of two systems: handling and processing external data (middleware), as seen in many related works, and preparing the required data and the engine itself. The process involves gluing together paper to create corrugated paper, the raw material used to produce cardboard boxes. To ensure continuous splicing, the next roll must be prepared at the end of each previous roll. After observing the splice from raw machine data, with no remaining meters on the roll, either a roll of the same quality must be manually prepared or another one. Using declarative process modeling and the execution on our CEP-based engine, we can simplify it into a single system and a single AlternateResponse constraint that includes the attribute conditions for activation. Compared to the imperative approach, which comprises eleven elements and requires two different systems for execution, our approach relies on a single system, and the declarative paradigm enables us to reduce it to a single constraint. Compared with the current approach, directly integrating low-level data events and eliminating the need for separate middleware improves consistency and reliability by reducing potential bottlenecks, simplifying execution, and streamlining process management. While a full quantitative benchmark was beyond the scope of this demonstration, the use case showed positive qualitative evidence of the tool's utility and efficacy. Acknowledging this, future work should focus on a comprehensive performance evaluation. Most issues concerning scalability and performance are primarily dependent on the implementation, not the core concept. The performance of CEP engines, such as Esper, provides a strong foundation, with older benchmarks already demonstrating high throughput. For a robust industrial deployment, this architecture would be enhanced by integrating event-sourcing tools such as Apache Kafka for ingestion and a load balancer to ensure high availability, thereby enabling rigorous testing under real-world conditions.

Declaration on Generative AI

During the preparation of this work, the authors used Grammarly for grammar and spelling checks.

References

[1] O. Etzion, Event processing in action, Manning, 2011.

[2] S. Schönig, L. Ackermann, S. Jablonski, et al., IoT meets BPM: a bidirectional communication architecture for IoT-aware process execution, Softw. Syst. Model. 19 (2020) 1443–1459.

[3] P. Soffer, A. Hinze, A. Koschmider, et al., From event streams to process models and back: Challenges and opportunities, Inform. Syst. 81 (2019) 181–200.

[4] C. Di Ciccio, M. Montali, Declarative Process Specifications: Reasoning, Discovery, Monitoring, Springer International Publishing, 2022, pp. 108–152.