

A Scalable Framework to Detect Interval Temporal Patterns in Data Streaming

Zhongqing Chen, Feng Liu, Hongyu Jia, and Liang Zhang^(✉)

School of Computer Science
Fudan University, Shanghai, China
{21210240135, 21210240246, 21212010017, lzhang}@fudan.edu.cn

Abstract. Complex event processing (CEP) is vital for real-time analysis of data streaming where interval temporal pattern detection plays a fundamental role due to the rich semantics embedded. Traditional CEP systems rely on sequential pattern matching, which is inefficient in capturing the interaction behaviors, e.g., the Allen interval algebra, among data from multiple sources, which paralyzes complex queries over data streaming. In this paper, we propose a novel solution to the problem of interval temporal pattern detection for streaming data. It casts binary event pairs with interval relationships into regular expressions and automata to reach low-latency recognition. Any complex patterns are decomposed into multiple event pairs, detected *independently*, and then assembled *seamlessly* using natural joins to achieve accurate detection results. Experimental results demonstrate that the proposed method outperforms current state-of-the-art approaches when handling complex queries in terms of efficiency improvements. This is due to the excellence of its nearly quasilinear growth in time and quasilinear space complexity over exponential complexity in traditional approaches. These advancements contribute significantly to the efficiency of CEP systems in recognizing complex events with interval temporal relationships.

Keywords: complex event processing (CEP) · interval temporal pattern detection · data streaming · low-latency

1 Introduction

Real-time decision-making in dynamic environments, such as maritime situational awareness [1] and fraud detection [4], relies heavily on Complex Event Processing (CEP). CEP systems allow organizations to analyze vast streams of data, to deliver insights critical for timely actions. However, conventional CEP facilities pay attention to capability to describe and process temporal relationships among event, particularly interval events characterized by durations, which is fail to react to complex scenarios where most events occur over a period of time rather than at a single instant. This limitation hinders their ability to fully capture events that are overlap or relate to one another over time. Consequently, it is essential for CEP systems to handle events that occur over time and to describe their relationships, such as those defined by Allen’s interval relations [3].

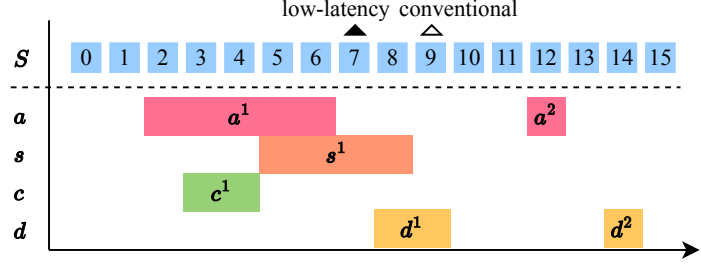


Fig. 1: Points events S vs meaningful interval events a^i , s^i , c^i , and d^i . The later aggregate contiguous point events that satisfy specific predicates accelerating a , speeding s , lane-changing c , and decelerating d . The interval events (a^1 , s^1 , c^1 , d^1) and their temporal relationship exemplify an aggressive driving pattern *Cutting in front of another driver and then slowing down*.

For example, a traffic monitoring system continuously receives data from connected vehicles to alert drivers of nearby threats, such as aggressively driven cars¹. According to the American Automobile Association, aggressive driving behaviors such as *cutting in front of another driver and then slowing down* depicted as Figure 1 pose significant risks on the road. In this scenario, behaviors are represented by interval events: a (accelerating), s (speeding), c (lane-changing), and d (decelerating). The temporal relationships between these interval events could be *naturally* described with Allen’s interval relations [3].

Sequential pattern matching [1] methods require representing various temporal combinations of interval events, which incurs huge complexity in processing. For instance, consider an extreme scenario where both lane changing and decelerating intervals occur within a speeding interval—a *during* relation in Allen’s Temporal Relations Algebra [3]. According to Allen, there are 13 possible relationships between these intervals. Representing the interactions among these three events using sequential pattern matching would require constructing 13 separate query statements. In practice, involving relations beyond *before*, *after*, *equals* introduces multiple possibilities. Additionally, as the number of interval dimensions increases, the complexity grows nearly exponentially, making it extremely difficult to represent using sequential pattern matching methods.

TPStream [8], a state-of-the-art method, detects multiple sets of interval event pairs characterized by Allen interval relations with low latency. However, complex patterns cause total paralysis since it matches nearly all interval events within the window. This results in exponential complexity as the problem size increases, thereby limiting its scalability in real-time streaming data scenarios.

To address these challenges, we develop a novel interval temporal pattern detection solution for streaming data. Technically, it decomposes complex patterns into multiple binary event pairs, which could be transformed into regular

¹ <https://exchange.aaa.com/safety/driving-advice/aggressive-driving>

expressions and automata to enable low-latency recognition. By detecting these binary event pairs *independently* (whose result is obtained in constant time), and seamlessly assembling them using natural joins (demonstrated as quasilinear time complexity with respect to the problem size), the approach ensures accurate and efficient pattern detection. Although the overall space complexity is higher than that of *TPStream*, it remains quasilinear, which significantly enhances the scalability of CEP systems in real-time streaming scenarios.

The main contributions of this paper are two folds,

- Proposing a novel framework for transforming predicate-based interval event pairs into regular expressions and automata. This enables efficient encoding of temporal relationships, achieving low-latency detection for complex interval patterns in streaming data.
- Developing a scalable approach to support above framework, not only reduces time complexity from exponential to nearly quasilinear but also achieves nearly quasilinear space complexity, significantly improving the efficiency of handling complex queries in real-time streaming systems.

The remainder of this paper is organized as follows. Section 2 reviews related work to set a foundation for the study. Section 3 describes the proposed framework and analyzes its computational complexity. Section 4 presents experimental results and discusses their implications. Finally, Section 5 concludes the paper and outlines future research directions.

2 Related Work

This section reviews existing approaches in traditional CEP and Allen temporal relations, identifies their limitations in low-latency interval event recognition, and position our method.

Traditional CEP Alevizos [1] introduces predicates in pattern definitions and encodes them as symbolic regular expressions and finite automata for event recognition. Event detection is achieved through state transitions that depend on whether the current event satisfies the predicate. However, this framework is limited to recognizing point events and patterns formed by the sequential connections of their Kleene closures. Drawing inspiration from this approach, the proposed framework encodes a pair of predicates that satisfy a specific Allen interval relation [3] as symbolic regular expressions and finite automata, enabling constant-time, low-latency recognition of pairs of interval events. Though StreamInsight [2], ZStream [11], and Cayuga [7] also explore time intervals in CEP, their work confines interval event relationships to the basic relations of *before*, *after*, and *equals* that can be derived through predicate conjunctions and yet cover other Allen interval relations, which thus limiting their expressiveness and applicability in more complex scenarios.

Allen Temporal Relations in CEP ISEQ [9] is a stream processing system that supports Allen interval relations. But it requires interval events as input. This requirement significantly impacts the system’s efficiency, particularly when dealing with real-time event streams. In Constrast, TPStream [8] supports the recognition of multiple pairs of interval events under all Allen relations and achieves low-latency performance. That is most close to our approach except its complexity. As the problem size increases, TPStream’s computational complexity grows exponentially, which can limit its scalability. Another concern regarding correctness and false positives remains to be fully addressed in TPStream ². Similarly, RTECA [10] could build upon Allen temporal relations within the RTEC framework [5], but it currently lacks support for detecting patterns under constraints that involve multiple pairs of interval events. This limitation restricts its applicability in more sophisticated event pattern recognition tasks.

In summary, there is an urgent need a scalable framework to detect interval temporal patterns with low latency in data streaming.

3 The Scalable Framework to Detect Interval Temporal Patterns in Data Streaming

Inspired by Allen’s interval temporal relations [3] and the symbolic representation of complex events through automata [1, 6], we present a framework that introduces a novel transformation of predicate-defined interval event pairs into symbolic regular expressions. Low-latency detection is then achieved through symbolically equivalent automata derived from these regular expressions. The detection outcomes are subsequently aggregated to produce the final results.

3.1 Related Concepts

Query Language TPStream [8] proposes a SQL-based query language. We follow the same interface as it but propose a more efficient implementation to ensure the stability of its upper level software. As an example, consider the query statement for detecting aggressive driving behaviors shown in Example 1:

Example 1. Query Statement for Detecting Aggressive Driving Behaviors

```

FROM    Carsensors
DEFINE  a AS accel > 8,  -- Acceleration(m/s^2)
          s AS speed > 70, -- Speeding threshold(mph)
          d AS accel < -9  -- Hard braking(m/s^2)
          c AS (lateralAccel > 0.5) OR (lateralAccel < -0.5)
              -- Horizontal speed(m/s)

PATTERN a meets;overlaps;starts;during s
          AND a meets;overlaps;finished_by;contains c

```

² We are currently in communication with the authors of TPStream to discuss these challenges.

```

    AND s contains;overlaps;finished_by c
    AND c overlaps;meets;before d
    AND s contains;finished_by;overlaps;meets d
    AND a before d
  WITHIN 5 minutes
  RETURN FIRST(s.timestamp) AS startTime,
         AVG(s.speed) AS avgSpeed

```

Here, the DEFINE clause defines some Predicate-based interval events based on four predicates. The PATTERN clause connects six pairs of interval events using AND, referred to as mPiePair in this paper, where each pair includes n Allen temporal relations. The WITHIN clause defines the time window for query, and the query results are returned in the form of aggregate functions. \square

Minterms of two predicates To describe the conditions of interval events more precisely, Boolean algebra is used. For any predicates φ and ψ defined on the domain \mathcal{D} , we use Boolean combinations, such as $accel > 8$ or $(lateralAccel > 0.5) \vee (lateralAccel < -0.5)$. As shown in [6], the Boolean conjunction of two predicates φ, ψ generate four minterms:

$$N = \{\varphi \wedge \psi, \neg\varphi \wedge \psi, \varphi \wedge \neg\psi, \neg\varphi \wedge \neg\psi\}.$$

These minterms represent all possible logical relationships the two predicates φ and ψ can produce. Their key properties are:

- **Mutual Exclusivity:** No two minterms overlap, ensuring distinct interpretations.
- **Completeness:** The union of all minterms covers the entire domain \mathcal{D} , partitioning it into disjoint subsets.

Thus, by using minterms, the domain \mathcal{D} is partitioned into four non-overlapping logical states, enabling precise and efficient analysis of interval events.

3.2 Low-Latency Detection of Predicate-Based Interval Event Pairs

This component begins with fundamental point events and streaming data, gradually introducing interval events, predicates, and temporal relations. It systematically constructs a pair of predicate-based interval events from bottom to up that satisfies a temporal interval relation and then converts it into symbolic automata to enable low-latency detection.

Data Stream A data stream is composed of continuously point events, each characterized by a timestamp and attributes. Consecutive point events can be aggregated into interval events to capture meaningful patterns.

Definition 1 (Point Event). *A point event is defined as an entity characterized by a timestamp and a payload.*

The timestamp t represents the moment when the event occurs, while the payload p is a set of attributes defined over the domain \mathcal{D} , expressed as $p = \{a_1, a_2, \dots, a_k\}$, where each $a_i \in \mathcal{D}$. A point event can thus be represented as:

$$pe = (t, p)$$

Definition 2 (Stream). *A stream is a sequence of point events ordered by ascending timestamps, where each point event is defined over a domain, and the timestamps are strictly increasing.*

Let S denote the stream, and pe represent point events. Each point event $pe.p$ is defined over the domain \mathcal{D} , and the timestamps satisfy $pe^i.t < pe^{i+1}.t$ for all i . The stream can be represented as:

$$S = pe^i \cdot pe^{i+1} \cdot \dots$$

Point events occurring continuously over a period can be aggregated into an interval event. Point events can also be viewed as special interval events with a unit length. The specific definition of an interval event is as follows.

Definition 3 (Interval Event). *An interval event consists of a series of consecutive point events and can be modeled as a contiguous subsequence of a stream.*

Let ie denote an interval event composed of consecutive point events. The length of the interval is at least one, i.e., $|ie| \geq 1$. An interval event is:

$$ie = pe^i \cdot pe^{i+1} \cdot \dots \cdot pe^j, \quad \text{where } j \geq i$$

The start time of the interval event is $pe^i.t$ and the end time is $pe^{j+1}.t$.

Example 2. As shown in Figure 1, S consists of 15 point events from car sensors, including vehicle ID, with *instant* speed, acceleration, lateral velocity, timestamp and other attributes. Predicates a , s , c , and d are defined on these attributes, and the colored blocks in the figure represent interval events that satisfy the corresponding predicates. Specifically, a^1 is an interval event with a duration of 5, starting at timestamp $a^1.ts = 2$ and ending at $a^1.te = 7$. Additionally, d^2 is an interval event of length 1, corresponding to point event pe^{14} . \square

Predicate-based Interval Event (PIE) Predicates identify point events in a data stream that satisfy specific conditions. we can define PIE as below,

Definition 4 (Predicate-Based Interval Event (PIE)). *A Predicate-Based Interval Event (PIE) is the longest interval event in which all internal point events satisfy the predicate φ . Specifically, for an interval event $ie = pe^i \cdot \dots \cdot pe^j$ where $i \leq j$, pie_φ is defined if and only if the following conditions are met:*

$$\begin{aligned} & \forall k \in [i, j], \quad pe^k \models \varphi, \\ & \wedge (pe^{i-1} \not\models \varphi), \\ & \wedge (pe^{j+1} \not\models \varphi). \end{aligned}$$

Here, $pe^k \models \varphi$ indicates that the point event pe^k satisfies the predicate φ , while pe^{i-1} and pe^{j+1} are the immediate predecessor and successor point events of pe^i and pe^j , respectively.

As shown in Figure 1, within stream S , only the interval events a^1 and a^2 satisfy pie_a . Although the interval event covered by c^1 also meets pie_a , it is not the longest subsequence and therefore does not qualify under pie_a .

Assuming the data stream is ordered and stable, and that a PIE represents the longest matching subsequence, PIEs based on the same predicate are time-ordered, non-overlapping, and have a minimum length of 1. Consequently, the start and end times of pie_φ are strictly ordered and unique, making the start time $pie_\varphi.ts$ a sufficient and unique identifier for each PIE.

Predicate-Based Interval Event Pair (PiePair) Allen's interval temporal relations [3] can be used to describe the relationships between interval events, comprising 13 unique temporal relations, as shown by \mathcal{A} in Table 1. Each temporal relation $r \in \mathcal{A}$ has a corresponding inverse relation $r^{-1} \in \mathcal{A}$, such that for any two interval events ie_i and ie_j , $r(ie_i, ie_j)$ is equivalent to $r^{-1}(ie_j, ie_i)$. For example, in Figure 1, a^1 and s^1 satisfy the *overlaps* relation, denoted as $overlaps(a^1, s^1)$.

While Allen's relations provide a comprehensive framework, they lack the semantic precision to represent "immediate succession." The *before* and *after* relations, for instance, continue to match subsequent intervals beyond the immediate pair, leading to imprecise semantics and redundant matches. To address this limitation, two additional temporal interval relations, *followed-by* and *follows*, are introduced. These relations focus on immediate pairs, ensuring semantic clarity and reducing redundant matches. Beyond addressing the semantic gap, these relations are highly compatible with the proposed method, enabling efficient identification. Moreover, in system implementation, *followed-by* and *follows* are more resource-efficient than *before* and *after* and can act as effective substitutes under suitable conditions.

Definition 5 (Precise Temporal Interval Relations). *Precise temporal interval relations are a set of 13 temporal relationships, denoted as \mathcal{P} , which includes followed-by, follows, and 11 other relations from Allen's interval temporal relations [3], excluding before and after. The relation followed-by($pie_\varphi^i, pie_\psi^j$) holds if:*

- pie_φ^i ends before pie_ψ^j starts ($pie_\varphi^i.te < pie_\psi^j.ts$);
- No intermediate interval event of pie_φ occurs before pie_ψ^j ;
- No intermediate interval event of pie_ψ occurs after pie_φ^i .

The relation *follows* can be defined through predicate transformation of the *followed-by* relation and all relations are detailed in Table 1.

As listed in Table 1, any two interval events ie_i and ie_j , if they satisfy a precise temporal interval relation $r \in \mathcal{P}$, this relationship can be expressed as $r(ie_i, ie_j)$.

For example, in Figure 1, there exists (c^1, d^1) that satisfies $\text{followed-by}(c^1, d^1)$ and also satisfies $\text{before}(c^1, d^1)$, but (c^1, d^2) only satisfies before relation.

Definition 6 (Interval Event Pair (IEP)). *An Interval Event Pair (IEP) is defined as a triplet (ie_i, ie_j, r) , where ie_i and ie_j are distinct interval events, and r is the temporal relation between the interval events (ie_i, ie_j) , satisfying $r(ie_i, ie_j)$, where $r \in \mathcal{P} \cup \mathcal{A}$. As shown in Table 1.*

Definition 7 (Predicate-Based Interval Event Pair (PiePair)). *A Predicate-Based Interval Event Pair (PiePair) represents two PIEs, pie_φ and pie_ψ , based on different predicates, along with a specific temporal interval relation r , where $r \in \mathcal{P} \cup \mathcal{A}$. It is expressed as $r(pie_\varphi, pie_\psi)$.*

Similar to a PIE, a PiePair is an abstract representation. Although it specifies a temporal interval relation, it corresponds to multiple concrete interval event pairs (IEPs) in the data stream. For example, in Figure 1, the interval event pair that satisfies the PiePair $\text{overlaps}(pie_a, pie_s)$ is (a^1, s^1) , expressed in IEP as $(a^1, s^1, \text{overlaps})$. This represents the accelerating vehicle, transitioning from non-speeding to speeding, and remaining speeding after acceleration stops.

Example 3. The temporal interval relation followed-by captures the closest adjacency within the before relation. In Figure 1, the IEPs satisfying PiePair $\text{before}(pie_c, pie_d)$ include $(c^1, d^1, \text{before})$ and $(c^1, d^2, \text{before})$, but only an IEP $(c^1, d^1, \text{followed-by})$ satisfies $\text{followed-by}(pie_c, pie_d)$. Using temporal relation before can cause a lane-changing interval c^1 to repeatedly match with d^1 and subsequent deceleration intervals, increasing computational overhead. Thus, replacing before with followed-by in certain queries can significantly reduce this burden. \square

Transfer PiePair into SRE and Automata Another component transforms PiePairs with precise temporal relations into their corresponding symbolic regular expressions (SREs), enabling efficient representation and processing of interval event pairs in streaming data.

Based on the previously defined PIE pie_φ , it satisfies the Symbolic Regular Expression (SRE) $E = \varphi^+$ [6], meaning $pie_\varphi \in \mathcal{L}(\varphi^+)$. Similarly, a PiePair corresponds to an SRE. For instance, $\text{overlaps}(pie_\varphi, pie_\psi)$ in the stream S maps to $\mathcal{L}((\varphi \wedge \neg\psi)^+ \cdot (\varphi \wedge \psi)^+ \cdot (\neg\varphi \wedge \psi)^+)$.

To analyze PiePairs more effectively, we introduce a mapping function that translates the original data stream S into a predicate-based stream $S_{\varphi, \psi}$ composed of symbolic representations of minterms.

Definition 8 (Minterm-Based Mapping Function). *Given a pair of predicates $\varphi, \psi \in \Psi$ defined over the domain \mathcal{D} , a bijective mapping function $\mathcal{F}_{\varphi, \psi}$ is defined to map a point event $pe = (t, p)$ to a symbol from the alphabet $\Sigma = \{O, I, Z, E\}$:*

$$\mathcal{F}_{\varphi,\psi}(pe) = \begin{cases} O & \text{if } pe.p \models (\neg\varphi \wedge \neg\psi), \\ I & \text{if } pe.p \models (\neg\varphi \wedge \psi), \\ Z & \text{if } pe.p \models (\varphi \wedge \neg\psi), \\ E & \text{if } pe.p \models (\varphi \wedge \psi), \end{cases}$$

Here, O , I , Z , and E represent the four possible Boolean states of the predicates φ and ψ .

Definition 9 (Predicate-Based Stream). *The predicate-based stream $S_{\varphi,\psi}$ is obtained by applying $\mathcal{F}_{\varphi,\psi}$ to each point event in S , forming a sequence of symbols:*

$$S_{\varphi,\psi} = \mathcal{F}_{\varphi,\psi}(pe^1) \cdot \mathcal{F}_{\varphi,\psi}(pe^2) \cdot \dots$$

The PiePair $\text{overlaps}(pie_\varphi, pie_\psi)$ in the stream S maps to a subsequence in $S_{\varphi,\psi}$ that satisfies the symbolic regular expression:

$$\mathcal{L}((\varphi \wedge \neg\psi)^+ \cdot (\varphi \wedge \psi)^+ \cdot (\neg\varphi \wedge \psi)^+),$$

where each segment corresponds to a specific phase of the temporal relation overlaps:

- $(\varphi \wedge \neg\psi)^+$: The period where only pie_φ holds.
- $(\varphi \wedge \psi)^+$: The overlapping period where pie_φ and pie_ψ hold simultaneously.
- $(\neg\varphi \wedge \psi)^+$: The period where only pie_ψ holds.








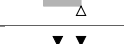
Example 4. In Figure 2, the mapping function $\mathcal{F}_{a,s}$ is based on the minterms of predicates a (accelerating) and s (speeding). Here, $S_{a,s}$ is the stream obtained by mapping S through $\mathcal{F}_{a,s}$. Point events indicating the vehicle is accelerating and speeding are mapped to the alphabet E , while those indicating the vehicle is accelerating but not speeding are mapped to Z , and so on. \square

Table 1 specifies corresponding relationships, and $E(r(pie_\varphi, pie_\psi))$ lists the SREs for the PiePair $r(pie_\varphi, pie_\psi)$, which satisfy the relationships defined in Proposition 1. However, no corresponding SREs are provided for the relations *before* and *after* because these relations can continuously match for a given PIE, resulting in an SRE with a Kleene closure with wildcards. This structure prevents a one-to-one correspondence with the PiePair, violating Proposition 1. As a result, automaton-based detection is unsuitable for these relations, and we instead use endpoints to derive them while still enabling low-latency detection.

Proposition 1. *A PiePair is defined on the original data stream S , and symbolic regular expressions (SREs) are defined on the corresponding predicate-based stream $S_{\varphi,\psi}$, derived by applying the mapping function $\mathcal{F}_{\varphi,\psi}$. The relationship between PiePairs and SREs, based on precise temporal interval relations, is as follows:*

1. *If a sequence $sie \in \mathcal{L}(E(r(pie_\varphi, pie_\psi)))$ exists in $S_{\varphi,\psi}$, then the PiePair $r(pie_\varphi^i, pie_\psi^j)$ exists in S .*

Table 1: Temporal Interval Relations: Visualization, Types, and Corresponding SRE for Predicates φ and ψ

r	r^{-1}	$E(r(\text{pie}_\varphi, \text{pie}_\psi))$	Visualization	$r, r^{-1} \in$
followed-by	follows	$\varphi^+ \cdot (\neg\varphi \wedge \neg\psi)^+ \cdot \psi^+$		\mathcal{P}
meets	met-by	$(\varphi \wedge \neg\psi)^+ \cdot (\neg\varphi \wedge \psi)^+$		\mathcal{A}, \mathcal{P}
overlaps	overlapped-by	$(\varphi \wedge \neg\psi)^+ \cdot (\varphi \wedge \psi)^+ \cdot (\neg\varphi \wedge \psi)^+$		\mathcal{A}, \mathcal{P}
starts	started-by	$(\neg\varphi \wedge \neg\psi)^+ \cdot (\varphi \wedge \psi)^+ \cdot (\neg\varphi \wedge \psi)^+$		\mathcal{A}, \mathcal{P}
during	contains	$(\neg\varphi \wedge \psi)^+ \cdot (\varphi \wedge \psi)^+ \cdot (\neg\varphi \wedge \psi)^+$		\mathcal{A}, \mathcal{P}
finishes	finished-by	$(\neg\varphi \wedge \psi)^+ \cdot (\varphi \wedge \psi)^+ \cdot (\neg\varphi \wedge \neg\psi)^+$		\mathcal{A}, \mathcal{P}
equals	equals	$(\neg\varphi \wedge \neg\psi)^+ \cdot (\varphi \wedge \psi)^+ \cdot (\neg\varphi \wedge \neg\psi)^+$		\mathcal{A}, \mathcal{P}
before	after	/		\mathcal{A}

r^{-1} : The inverse relation of r , and $r(A, B)$ is equivalent to $r^{-1}(B, A)$.

\mathcal{A} : Allen's interval temporal relations [3].

\mathcal{P} : Precise temporal interval relations proposed in this paper.

Solid triangle marks *trigger time*: The earliest time when the relation r is satisfied.

Hollow triangle marks *completion time*: The final moment when the entire PiePair ends.

2. Conversely, if the PiePair $r(\text{pie}_\varphi^i, \text{pie}_\psi^j)$ exists in S , then a sequence $\text{sie} \in \mathcal{L}(E(r(\text{pie}_\varphi, \text{pie}_\psi)))$ must exist in $S_{\varphi, \psi}$.

Proof. The detailed proof of this proposition can be found in Appendix A.

SREs serve as a bridge between the temporal relations in the original data stream S and finite automata, enabling efficient detection of PiePairs. By converting SREs into deterministic symbolic finite automata (DSFA), and further into deterministic classical finite automata (DFA), we leverage state transitions for low-latency detection.

Lemma 1. For every symbolic regular expression E , there exists a symbolic finite automaton M such that $\mathcal{L}(E) = \mathcal{L}(M)$ [1].

The predicate-based stream $S_{\varphi, \psi}$ is defined over the alphabet $\Sigma = \{O, I, Z, E\}$, representing all possible minterm combinations of predicates φ and ψ . Using Lemma 1, we convert $E(r(\text{pie}_\varphi, \text{pie}_\psi))$ into a DSFA M_s^r , which is further transformed into a DFA M_c^r using Lemma 2.

Lemma 2. For each deterministic symbolic finite automaton (DSFA) M_s , there exists a deterministic classical finite automaton (DFA) M_c such that $\mathcal{L}(M_c) = \mathcal{L}(M_s)$ [1].

The trigger moment for low-latency detection is defined as the earliest moment when M_c^r transitions to a terminal state, indicating that the temporal relation r is satisfied.

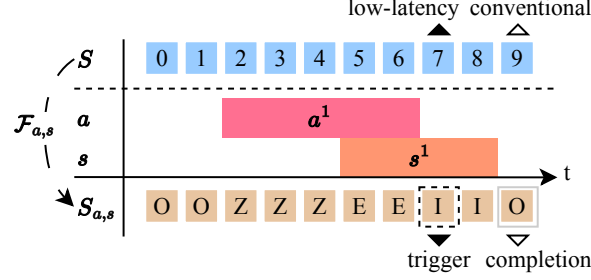


Fig. 2: S represents a data stream of point events, where predicates a and s correspond to accelerating and speeding behaviors, respectively. $\mathcal{F}_{a,s}$ is a bijective mapping function that maps the minterms of predicates a, s , i.e., $\{\neg a \wedge \neg s, \neg a \wedge s, a \wedge \neg s, a \wedge s\}$, to the alphabet $\{O, I, Z, E\}$. $\mathcal{F}_{a,s}$ maps S into $S_{a,s}$. Interval events a^1 and s^1 satisfy $\text{overlaps}(pie_a, pie_s)$.

PiePair Detection with Low Latency Low-latency detection refers to identifying temporal relations between interval events at the earliest possible moment, without waiting for all events to fully complete. This approach significantly reduces detection delays and is ideal for real-time streaming data applications.

The PiePair detection process is based on Precise Temporal Interval Relations. For relations like *before* and *after*, low-latency detection is achieved by comparing endpoints. Due to space limitations, the detailed process for these relations is not discussed here.

Traditional methods [9] require both interval events to fully complete before their temporal relations can be detected, resulting in unnecessary delays. In contrast, the proposed method achieves low-latency detection using M_c^r , which precisely models the temporal relations between predicates and detects relations by transitioning to a terminating state. For example, in Figure 2, the low-latency detection time for the acceleration interval a^1 and speeding interval s^1 occurs at $t = 7$, compared to $t = 9$ with traditional methods when the PiePair is fully completed. In TPStream [8], the detection time is denoted by t_3 .

The trigger moment for low-latency detection occurs when the automaton M_c^r transitions from a non-terminating state to a terminating state—a transition defined as the **trigger condition**.

Example 5. In Figure 3, the automaton for $\text{overlaps}(pie_\varphi, pie_\psi)$ transitions from state S_2 to S_3 , satisfying the trigger condition. This corresponds to the automa-

ton for $\text{overlaps}(pie_a, pie_s)$ making a state transition upon receiving the point event at $t = 7$ in Figure 2. \square

For certain temporal relations, such as finishes, finished-by, and equals, the PiePair is completed exactly at the trigger moment. For other relations, one end-point remains undefined until the PiePair is fully completed, which occurs when the automaton transitions based on the **completion condition**. The completion condition depends on the temporal relation and the received alphabet.

For relations like followed-by, overlaps, starts, during, and meets, the completion condition is met when the automaton, while in the terminating state, receives the alphabet $\{O, Z\}$. For relations such as follows, overlapped-by, started-by, contains, and met-by, the completion condition is met when the automaton receives $\{O, I\}$ in the terminating state.

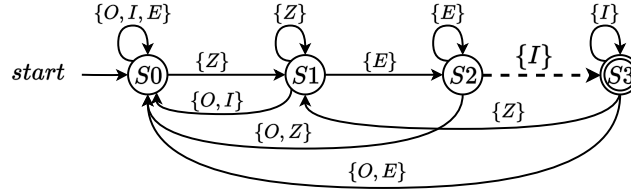


Fig. 3: The automaton derived from $\text{overlaps}(pie_\varphi, pie_\psi)$ employs transition symbols corresponding to the alphabet $\{O, I, Z, E\}$, which are the minterms of predicates φ and ψ , i.e., $\{\neg\varphi \wedge \neg\psi, \neg\varphi \wedge \psi, \varphi \wedge \neg\psi, \varphi \wedge \psi\}$. Dashed transitions represent trigger conditions.

Algorithm 1: detectPiePair

Input : Current point event $pe(t, p)$, precise temporal interval relation r in PiePair, shared result queue Q , and automaton M corresponding to $r(pie_\varphi, pie_\psi)$.

Output: Q : Updated queue

- 1 **Initialization** $endPoint[4]$: Buffer for recording endpoint events
- 2 $M.lastState \leftarrow M.state$
- 3 $M.state \leftarrow transition(M.state, pe.p)$
- 4 **if** $M.lastState \neq M.state$ **then**
- 5 $endPoint \leftarrow record(r, M.lastState, M.state)$;
- 6 **if** $isTriggered(r, M.lastState, M.state)$ **then**
- 7 $iep \leftarrow CreateIEP(r, endPoint)$;
- 8 $Q.enqueue(iep)$;
- 9 **if** $isCompleted(r, M.lastState, M.state)$ **then**
- 10 $Q.update(endPoint)$

Algorithm 1 outlines the runtime low-latency detection process for a PiePair $r(pie_\varphi, pie_\psi)$. Before execution, the symbolic automaton M is initialized with predicates φ, ψ , and the temporal relation r , while the array *endPoint* is set up to record the start and end times of PIEs. As each point event pe is received in order, M updates its state based on pe 's attributes, recording the previous state as $M.lastState$. Each state transition triggers the recording of relevant time points, such as the start or end times of PIEs, depending on the transition states and the temporal relation.

In lines 6-8, the algorithm evaluates the trigger condition for low-latency detection. Each temporal relation has a predefined trigger condition that specifies the automaton transition required for detection. When the trigger condition is met, the algorithm performs low-latency detection by creating a partially detected PiePair iep and appending it to the detection queue Q , even if the final endpoint is not yet determined. As shown in Example 5, the temporal relation *overlaps* is detected at $t = 7$. Since the relation *overlaps* is not yet completed at the trigger moment, the iep stored in Q is $((2, 7), (5, \text{null}), \text{overlaps})$, where the second endpoint remains undefined. After this round, the iep will be used to match the full pattern immediately, which involves multiple relations within a PiePair and will be discussed in section 3.3.

In lines 9-10, the algorithm checks the completion condition to finalize the detected PiePair. Once satisfied, it updates Q with the fully completed PiePair. For the relation *overlaps* in Figure 2, completion occurs when the automaton receives the alphabet $\{O, Z\}$, at which point $pie_s.te = 9$ is recorded and used to update the corresponding PIEs with the same start time in queue Q .

This process ensures that low-latency detection is performed as early as possible while maintaining correctness through subsequent completion checks. Furthermore, regardless the completion condition is satisfied, the algorithm continues to process streaming data by performing state transitions and adding or updating iep in the detection queue Q , thereby maintaining continuous and efficient detection without interruption.

3.3 Full Pattern Matching with Join Operations

In this section, we discuss the complete process of full pattern matching with join operations, focusing on the detection of event pairs under multiple temporal relations and the merging of these pairs using natural joins.

Detecting PiePairs under Multiple Temporal Relations The method described above enables the detection of a PiePair with a specific temporal interval relation. However, it is important to consider that a pair of interval events can exhibit multiple possible relationships. For example, in the context of aggressive driving behavior discussed in Example 1, the temporal relation between a and s could be *meets*, *overlaps*, *starts*, or *during*. To simplify the representation of multiple potential temporal relationships between a pair of PIEs, we define the multi-relationship interval event pair (mPiePair) based on predicates, as follows.

Definition 10 (Predicate-Based Interval Event Pair under Multiple Relations (mPiePair)). A Predicate-Based Interval Event Pair under Multiple Relations (*mPiePair*) represents a pair of PIEs that satisfies one of several specific temporal relations. If a pair of PIEs, pie_φ and pie_ψ , satisfies a temporal relation r , where $r \in C$ and $|C| > 0$, then the *mPiePair* can be expressed as $C(pie_\varphi, pie_\psi)$. Furthermore, $C(pie_\varphi, pie_\psi) \iff \exists r \in C, r(pie_\varphi, pie_\psi)$.

For example, in the case of aggressive driving behavior shown in Figure 1, (a^1, s^1) satisfies $overlaps(a^1, s^1)$ and also satisfies the *mPiePair* $\{meets, overlaps, starts, during\}(pie_a, pie_s)$.

To detect an *mPiePair* $C(pie_\varphi, pie_\psi)$, it can be transformed into detecting $|C|$ individual PiePairs, $r(pie_\varphi, pie_\psi)$, for each $r \in C$, as described in Section 3.2 and the full detection process is described in Algorithm 2.

Algorithm 2: detectMPiePair

Input : Current point event $pe(t, p)$, *mPiePair* mpp to be detected, set of automata $MSet$ for the *mPiePair*, shared result queue Q , and queue Qba for temporal relations *before* and *after*.
Output: Updated shared result queue Q and Qba .
1 **foreach** Automaton $M \in MSet$ **do**
2 | **detectPiePair**($pe, Q, M.r, M$);
3 **end**
4 **if** *before* $\in mpp.C$ **or** *after* $\in mpp.C$ **then**
5 | **deriveBefAftRels**(Q, Qba);

The core idea of Algorithm 2 is to detect PiePairs, which involves calls to the function *detectPiePair*, as shown in lines 1-2. It is important to note that *mPiePairs* share a common result queue since different temporal relations cannot be triggered simultaneously for the same pair of PIEs. That is, at any given time, only one PiePair can be recognized for each pair of PIEs. As a result, multiple relations for the same PIE pair can be stored together and even detected in parallel.

After the *mPiePair* detection, the algorithm processes the *before* and *after* relations between PIEs (lines 4-5). This involves storing interval events within the window and performing comparisons and derivations on the endpoints. For simplicity, the details of this process are omitted here, but the basic idea is to efficiently capture and derive temporal relations.

Merging Results through Natural Joins In this section, we describe how the results detected by *mPiePairs* can be merged using natural joins to obtain the correct detection results.

The results detected by *mPiePairs* contain four endpoints and a temporal relation. Each result can be viewed as a record, where the two starting endpoints serve as the keys for the corresponding interval events. If multiple *mPiePairs*

involve the same interval events, the corresponding start times must be the same. Therefore, natural joins can be used to merge the detection results of the mPiePairs, ensuring the correct final result is obtained.

To formalize this process, we introduce the following theorem that asserts the equivalence between the detection result of a Pattern query and the result obtained by performing a natural join on the detected mPiePairs.

Theorem 1. *The detection result of a Pattern query is equivalent to the result obtained by detecting mPiePairs and then performing a natural join on the results. More formally:*

The Pattern query consists of n mPiePairs, represented by the set $MPP = \{C_i(pie_{\varphi_i}, pie_{\psi_i}) \mid 0 < i < n\}$, connected by the logical operator AND. The query also includes all the m PIEs in the set $\mathcal{X} = \{pie_i \mid 0 < i < m\}$, where m is the total number of PIEs involved in the query. Let $\mathcal{Y} = \{ie_i \mid 0 < i < m\}$ represent the a corresponding detection result. Then:

- **Sufficiency:** *If the result \mathcal{Y} satisfies the Pattern query, then it must also satisfy the detection result of the mPiePairs followed by natural joins.*
- **Necessity:** *Conversely, if the result \mathcal{Y} satisfies the mPiePair detection followed by the natural join, it must also satisfy the Pattern query.*

Proof. The detailed proof of this theorem can be found in Appendix B.

The detection results of multiple mPiePairs can be merged through multiple natural joins. In this approach, we simplify the process by transforming the multiple natural joins into a sequence of pairwise natural joins, which are performed one after another to obtain the final result.

In streaming data scenarios, to accelerate the speed of merging results, this approach implements hash indexing and performs incremental joins after each detection round, instead of re-scanning and joining all intermediate results in the window. This significantly improves the join speed. A detailed complexity analysis of this approach is delayed to Section 3.4.

Full Pattern Matching Process The previous sections have described how to detect multiple temporal relationships that may exist between a pair of predicates, and how to obtain the final pattern matching result through natural join operations based on the detection results. The following will present a detailed explanation of the entire process using Algorithm 3.

First, in line 1, the query statement q is parsed to obtain a set of mPiePairs $mppSet$, the window size W , and mappings from each mPiePair to the result queue Q , the result queue for **before** and **after** relations Qba , as well as the corresponding set of automata for each mPiePair. The point events in the streaming data are continuously received. Before processing new data for detection, expired events beyond the window size are cleared (lines 2-3). As shown in lines 4-9, for each received point event, the mPiePairs are processed sequentially. In lines 6-8, the corresponding automaton set, result queue Q , and the result queues for the

Algorithm 3: executeQuery

Input : Query statement q , streaming data to be detected S

```

1 ( $mppSet, W, mppToQ, mppToQba, mppToAutoSet$ )  $\leftarrow$   $parse(q)$ 
2 while  $S.hasNext()$  do
3   clearExpiredEvents( $W$ )
4    $pe \leftarrow S.getNext()$ 
5   foreach  $mpp \in mppSet$  do
6      $MSet \leftarrow mppToAutoSet[mpp]$ 
7      $Q \leftarrow mppToQ[mpp]$ 
8      $Qba \leftarrow mppToQba[mpp]$ 
9     detectMPiePair( $pe, mpp, MSet, Q, Qba$ )
10   $res \leftarrow$  (mergeByJoins( $mppToQ, mppToQba, mppSet$ ))
11   $output(res)$ 

```

before and after relations Qba are retrieved. Line 9 then calls Algorithm 2 to perform the mPiePair detection, updating the Q and Qba queues. After processing all mPiePairs, the results are merged through a series of sequential binary natural joins to produce the final output, as shown in lines 10-11.

3.4 Complexity

The time and space complexities of the entire method, i.e., Algorithm 3, are analyzed below. This analysis considers the presence or absence of the before and after relations.

Time Complexity

Without before/after relations In each round, the detection of a single PiePair, as described in Algorithm 1, takes constant time. In Algorithm 2, each mPiePair contains k_i relations ($k_i \leq 13$), where k_i represents the number of temporal relations. The detection of each mPiePair requires $O(k_i)$ time. Given that Algorithm 3 processes all n mPiePairs, the detection time complexity per round becomes $O(nk_i)$. To merge the results from the n mPiePairs, $n-1$ binary natural joins are performed. Utilizing hash indexing on the result queue and incremental joins, and given that each mPiePair produces at most one result per round, the time for performing a single binary join operation is constant. Consequently, the overall time complexity for the result merging process is $O(n-1)$.

With before/after relations When the before or after relations are involved, an additional derivation step is required in Algorithm 2. This step depends on both the window size W and the event density within the stream, with a worst-case time complexity of $O(W)$ and a best-case time complexity of $O(1)$. Therefore, the detection time complexity becomes $O(n(k_i + W))$. During the natural join process, the inclusion of before and after relations in the mPiePair detection results

increases the complexity to $O(W)$. Consequently, the overall time complexity for the join process becomes $O(W(n-1))$.

Space Complexity

Without before/after relations During detection, since the mPiePairs share a common result queue, the n mPiePairs require n queues of average length W , resulting in a space complexity of $O(nW)$. In the natural join process, intermediate results and indices must be stored. Each intermediate result may add a new column, with a length of W , leading to an additional space complexity of $O\left(W \sum_{i=3}^{n+2} i\right) = O(n^2W)$. In the extreme case where every result can be joined, the space complexity may reach $O(W^n)$. However, such extreme cases are typically rare due to practical constraints on event densities and query statements.

With before/after relations When the before and after relations are present, they continuously include prior interval events within the time window W . As a result, each mPiePair requires an average queue length of W^2 for the Qba queues. Therefore, the total space complexity for these relations is $O(nW^2 + nW)$. Similarly, intermediate results now have a length of W^2 , which requires additional space of $O\left(W^2 \sum_{i=3}^{n+2} i\right) = O(n^2W^2)$. As discussed earlier, in the worst case where every result can be joined, the space complexity may reach $O(W^{2n})$.

In summary, under typical conditions, the complexities can be categorized into two scenarios. *Without* the before/after relations, the time complexity per round is $O(n)$, and the space complexity is $O(n^2W)$. *With* the before/after relations, the time complexity per round becomes $O(nW)$, accounting for the additional derivation and join steps, while the space complexity increases to $O(n^2W^2)$. These estimates reflect typical scenarios, excluding extreme cases where complexities may be higher.

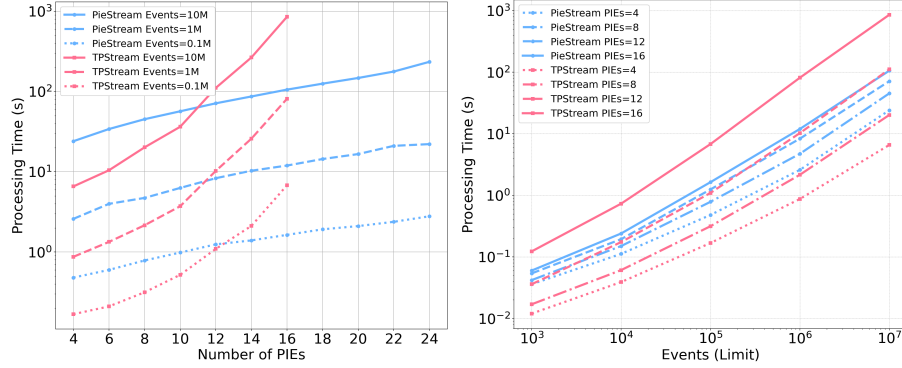
4 Experiment

In this section, we evaluate the proposed method by comparing it with TPStream [8], the current state-of-the-art solution for this problem. The primary focus of the comparison is on processing time under conditions of large-scale datasets and complex queries. The experimental data used in this study is synthesized from TPStream. Below, we first outline the experimental setup and then describe the experiment conducted.

All experiments were conducted on a Dell G7 7590 equipped with an Intel Core i7-9750H processor (12 cores), 32GB of RAM, and running Ubuntu 24.04.1. The system utilized Java version 1.8 and was restricted to a maximum of 16GB of memory usage during the experiments.

TPStream is available as open source³, and the proposed method, PieStream, is also open sourced⁴. The following experiments were conducted using the same synthetic dataset and query statements. A sequence of n PIEs forms $n - 1$ mPiePairs, with each mPiePair containing 6 same Allen temporal relations. The number of PIEs denotes the scale of the problem, with its linear growth reflecting an increase in complexity.

Processing Time The experiment used a window size of 10,000, with the fastest event input rate. Different numbers of events (ranging from 10^3 to 10^7) were input into PieStream and TPStream. Multiple experiments were conducted by varying the problem scale (i.e., adjusting PIEs from 4 to 18) to evaluate the total processing time.



(a) Effect of PIE quantity on processing time. (b) Effect of processed event count on processing time.

Fig. 4: Processing time as a function of PIEs and event count with a window size of 10,000.

Figure 4a illustrates the relationship between processing time and problem size. For PieStream, the processing time increases quasilinearly with the problem size, whereas TPStream [8] exhibits an exponential growth in processing time, causing the TPStream method to fail execution when the number of PIEs exceeds 16. This exponential growth is attributed to TPStream's low-latency algorithm, which directly matches all unfinished and completed interval events within the window. Consequently, the algorithmic complexity escalates sharply as the problem size increases.

Although PieStream's processing time also grows with the problem size, the rate of increase is significantly slower. Because for small problem sizes, TPStream's direct matching is faster, while PieStream incurs additional overhead

³ <http://uni-marburg.de/oaCPk>

⁴ <https://github.com/cyann7/pieP>

from index construction and result management. However, as problem size increases, TPStream’s matching time grows significantly, whereas PieStream’s indexing and result management reduce redundant computations, resulting in better performance.

The primary challenge for PieStream in achieving linear scalability lies in multi-way natural joins, which can generate massive result sets when dealing with large volumes of data. The time overhead associated with generating these result sets, constructing indexes, and removing expired results from previous joins collectively contributes to the increased processing time.

Figure 4b shows that processing time increases linearly with the number of events processed. When the number of PIEs is below 12, TPStream outperforms PieStream in terms of processing time. However, once the number of PIEs exceeds 12, PieStream’s processing time becomes lower than that of TPStream, with the performance gap widening as the problem size grows.

Furthermore, the blue line representing PieStream in Figure 4b is more concentrated, indicating that PieStream maintains relatively stable performance across different problem sizes. This observation further supports the assertion that PieStream’s time complexity scales quasilinearly with the problem size.

5 Conclusion and Future work

To leverage CEP to complex interaction behaviors and queries over data streaming. We proposed a novel framework that casts binary event pairs with interval relationships into regular expressions and automata to reach low-latency recognition. As a result, any complex patterns could be decomposed into multiple event pairs, detected *independently*, and then assembled *seamlessly* using natural joins to achieve accurate detection results. Analysis and Experimental results demonstrate that the proposed method outperforms current state-of-the-art approaches when handling complex queries in terms of efficiency improvements. We believe that these advancements contribute significantly to the efficiency of CEP systems in recognizing complex events with interval temporal relationships.

One weak spot of the proposed framework is that it fails to capture events with two identical predicates, which is our next step to study.

Acknowledgements This work is supported by Projects of International Cooperation and Exchanges NSFC-DFG (Grant No. 62061136006).

References

1. Alevizos, E., Artikis, A., Paliouras, G.: Complex event forecasting with prediction suffix trees. *The VLDB Journal* **31**(1), 157–180 (2022)
2. Ali, M.H., Gereia, C., Raman, B.S., Sezgin, B., Tarnavski, T., Verona, T., Wang, P., Zabback, P., Ananthanarayan, A., Kirilov, A., et al.: Microsoft cep server and online behavioral targeting. *Proceedings of the VLDB Endowment* **2**(2), 1558–1561 (2009)

3. Allen, J.F.: Maintaining knowledge about temporal intervals. *Communications of the ACM* **26**(11), 832–843 (1983)
4. Artikis, A., Katzouris, N., Correia, I., Baber, C., Morar, N., Skarbovsky, I., Fournier, F., Paliouras, G.: A prototype for credit card fraud management: Industry paper. In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. pp. 249–260. DEBS '17, Association for Computing Machinery (2017). <https://doi.org/10.1145/3093742.3093912>, <https://doi.org/10.1145/3093742.3093912>
5. Artikis, A., Sergot, M., Paliouras, G.: An Event Calculus for Event Recognition **27**(4), 895–908 (2015). <https://doi.org/10.1109/TKDE.2014.2356476>, <https://ieeexplore.ieee.org/document/6895142>
6. D’Antoni, L., Veanes, M.: The power of symbolic automata and transducers. In: Majumdar, R., Kunčák, V. (eds.) *Computer Aided Verification*. pp. 47–67. Springer International Publishing, Cham (2017)
7. Demers, A.J., Gehrke, J., Panda, B., Riedewald, M., Sharma, V., White, W.M., et al.: Cayuga: A general purpose event monitoring system. In: *Cidr*. vol. 7, pp. 412–422 (2007)
8. Körber, M., Glombiewski, N., Morgen, A., Seeger, B.: TPStream: Low-latency and high-throughput temporal pattern matching on event streams **39**(2), 361–412 (2021). <https://doi.org/10.1007/s10619-019-07272-z>, <https://link.springer.com/10.1007/s10619-019-07272-z>
9. Li, M., Mani, M., Rundensteiner, E.A., Lin, T.: Complex event pattern detection over streams with interval-based temporal semantics. In: *Proceedings of the 5th ACM International Conference on Distributed Event-Based System*. pp. 291–302. ACM (2011). <https://doi.org/10.1145/2002259.2002297>, <https://dl.acm.org/doi/10.1145/2002259.2002297>
10. Mantenoglou, P., Kelesis, D., Artikis, A.: Complex Event Recognition with Allen Relations. In: *Proceedings of the Twentieth International Conference on Principles of Knowledge Representation and Reasoning*. pp. 502–511. International Joint Conferences on Artificial Intelligence Organization (2023). <https://doi.org/10.24963/kr.2023/49>, <https://proceedings.kr.org/2023/49>
11. Mei, Y., Madden, S.: Zstream: a cost-based query processor for adaptively detecting composite events. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. pp. 193–206 (2009)

A Proof of Proposition 1

Due to space constraints, the proof is omitted but available upon request.

B Proof of Theorem 1

Due to space constraints, the proof is omitted but available upon request.