NEEL: The Nested Complex Event Language for Real-Time Event Analytics

Mo Liu[1], Elke A. Rundensteiner,[1] Dan Dougherty[1], Chetan Gupta[2], Song Wang[2], Ismail Ari[3], and Abhay Mehta[2]

[1] Worcester Polytechnic Institute, USA, `(liumo|rundenst|dd)@wpi.edu`,
[2] HP Labs, USA, `(chetan.gupta|songw|abhay.mehta)@hp.com`
[3] Ozyegin University, Turkey, `Ismail.Ari@ozyegin.edu.tr`

Abstract. Complex event processing (CEP) over event streams has become increasingly important for real-time applications ranging from health care, supply chain management to business intelligence. These monitoring applications submit complex event queries to track sequences of events that match a given pattern. As these systems mature the need for increasingly complex nested sequence query support arises, while the state-of-art CEP systems mostly support the execution of only flat sequence queries. In this paper, we introduce our nested CEP query language *NEEL* for expressing nested queries composed of sequence, negation, AND and OR operators. Thereafter, we also define its formal semantics. Subtle issues with negation and predicates within the nested sequence context are discussed. An E-Analytics system for processing nested CEP queries expressed in the *NEEL* language has been developed. Lastly, we demonstrate the utility of this technology by describing a case study of a real-world application in health care.

Key words: Nested Query, CEP, Syntax, Semantics

1 Introduction

Complex event processing (CEP) has become increasingly important in modern applications, ranging from online financial feeds, supply chain management for RFID tracking to real-time business intelligence [1, 2]. There is a strong demand for CEP technology that can be applied to process enormous volumes of sequential data streams for online operational decision making as demonstrated by several sample application scenarios below. CEP must be able to support sophisticated pattern matching on real time event streams including the arbitrary nesting of sequence operators and the flexible use of negation in such nested sequences. The need for such sophisticated CEP technology is motivated via several example applications next.
Motivating Example 1. In the web business application context, the query Q_1 = *SEQ(Create-profile c, Update-profile u, NOT (Answer-Email ae OR Answer-Phone ap, ae.uid = c.uid = u.uid, ap.uid = c.uid = u.uid))* detects customers not answering an email or a phone call after creating and then updating online

profiles within a specified time. This query could be used for checking customer inactivity with the goal to understand customer behavior, to subsequently delete unused customer profiles as well as to adopt marketing strategies to retain customer interest. Efficient execution of such complex nested CEP queries is critical for assuring real-time responsiveness and for staying competitive in an increasingly fast paced business world.

Motivating Example 2. Another example of applications in need of complex nested event queries are organizations that need to track the status of their inventory. Consider tracking inventory within a hospital setting. For instance, reporting contaminated medical equipments within the daily workflow [3, 4, 5]. Let us assume that the tools for medical operations are RFID-tagged. The system monitors the histories of the equipment (such as, records of surgical usage, of washing, sharpening and disinfection). When a healthcare worker puts a box of surgical tools into a surgical table equipped with RFID readers, the computer would display approximate warnings such as “This tool must be disposed”. A query Q_2 = *SEQ (Recycle r, Washing w, NOT SEQ(Sharpening s, Disinfection d, Checking c, s.id = d.id = c.id), Operating o, r.id = w.id = o.id, o.ins-type = “surgery”)* expresses this critical condition that after being recycled and washed, a surgery tool is being put back into use without first being sharpened, disinfected and then checked for quality assurance. Such complex sequence queries contain complex negation specifying the non-occurrence of composite event instances, such as negating the composite event of sharpened, disinfected and checked subsequences.

Motivating Example 3. In a supply chain management scenario, suppliers may need to monitor the transport of RFID tagged medical goods. It is important to neither break the cold chain of pharmaceuticals nor to overrun the expiration of any of these goods. During transportation and temporary storage, pharmaceuticals can be exposed to environmental conditions that may damage the goods. The temperature in the cooling trucks of the carrier may exceed the allowable limits, leading to spoilage. It is thus critical to monitor all ongoing transports in real time for tracking patterns of “safe” and “unsafe” transport. The pattern query Q_3 = *SEQ(Distribution-Center dc, OR(Hospital h, NursingHome n, h.id = d.id, n.id = d.id), DrugStore d, dc.id = d.id, d.temperature* $<$ *5° Celsius)* is submitted together with the required conditions to track the paths of each truck to its destination. Such pattern query is executed continually during daily operations to reliably identify possible violators or locations of violation. With suitable technology, decision makers reduce the risk of loss of products, danger to health of our communities, and even potential lawsuits. In this scenario, CEP technology is poised to help cut production costs and to increase the quality of goods for human consumption and health.

Beyond these three motivating examples, numerous other real-time monitoring scenarios are emerging for supply chain management (SCM), financial systems, sensor networks, and e-commerce. What is common across these scenarios is a need to query large volumes of sequence data in real-time. However, each scenario has its own characteristic in terms of streaming data volumes, query latency goals and event query complexities. We believe enabling complex nested CEP queries would allow us to efficiently correlate trends and detect anomalies on sequence data for real-time business intelligence.

While nested queries are commonly supplied by SQL engines over static databases, the state-of-art CEP literature [1, 2, 6] does not support such nested queries. While the Cayuga system [2] mentions composable queries, they assume the negation filter is only applied to a single primitive event type within the SEQ pattern. Our objective instead is to allow the specification of negation within any level of the nested query as demonstrated in the scenarios above. While CEDR [6] allows the application of negation over composite event types, authors didn't provide a clear syntax nor formal semantics for the specification of such nested pattern queries. In addition, existing CEP systems [1, 2, 6] don't provide query semantics concerning the handling of predicates in nested CEP queries. Design of a clean syntax and semantics for nested CEP queries is a delicate task. In this work, we address this gap by carefully designing the syntax, semantics and algebraic plan for complex nested sequence queries. Preliminary experiments with processing nested query plans expressed in the *NEEL* language are reported in [16]. Last but not least, we describe a case study of applying this technology to health care applications.

Organization of Paper: The rest of the paper is organized as follows: Section 2 introduces the event model. Section 3 presents our proposal of a nested query language *NEEL*. Sections 4 defines the nested query semantics of *NEEL*. Section 5 discusses our E-Analytics system. Sections 6 describes the case study. Section 7 discusses related work, while Section 8 concludes the paper.

2 The NEEL Event Model

An event instance is an occurrence of interest in a system which can be either primitive or composite as further introduced below. *A primitive event instance* denoted by a lower-case letter (e.g.,'e') is the smallest, atomic occurrence of interest in a system. e_i.ts and e_i.te denote the start and the end timestamp of an event instance e, respectively, with $e_i.\text{ts} \leq e_i.\text{te}$. For a primitive event instance e_i, $e_i.\text{ts} = e_i.\text{te}$. For simplicity, we use the subscript i attached to a primitive instance e to denote the timestamp i. *A composite event instance* is composed of constituent primitive event instances $e = < e_1, e_2, ..., e_n >$. A composite event instance e occurs over an interval. The start and end timestamps of e are equal to e.ts = $\min\{e_i.\text{ts} \mid \forall\ e_i \in \text{e}\}$ and e.te = $\max\{e_i.\text{te} \mid \forall\ e_i \in \text{e}\}$, respectively.

An *event type* is denoted by a capital letter, say E_i. An event type E_i describes a set of attributes that the event instances of this type share. An event type can be either a primitive or a composite event type [7]. *Primitive event types* are pre-defined in the application domain of interest. *Composite event types* are aggregated event types created by combining other primitive and/or composite event types to form an application specific type. $e_i \in E_j$ denotes that e_i is an instance of the event type E_j. We use e_i.type to denote the type E_j of e_i. Suppose

one of the attributes of E_j is attrj and $e_i \in E_j$, we use e_i.attrj to denote e_i's value for that attribute attrj.
Event History H is an ordered set of primitive event instances. Details of event history can be found in Section 4. *Cross product (×) of event histories* for A[H] = $\{a_1, a_2, ..., a_n\}$ and B[H] = $\{b_1, b_2, ..., b_m\}$ is A[H] × B[H] = { $\sum_{1 \leq i \leq n; 1 \leq j \leq m}$ $\{a_i, b_j\}$ with $a_i \in$ A[H] and $b_j \in$ B[H]}.

3 NEEL: The Nested Complex Event Query Language

We now introduce the *NEEL*[1] query language for specifying complex event pattern queries. *NEEL* supports the nesting of AND, OR, Negation and SEQ operators at any level of nesting.

<**Query**>::= PATTERN <event-expression>
WITHIN <window>
[RETURN <set of primitive events>]
E_v = <event-expression> <var>
<event-expression> ::=
SEQ((E_v $\mid$! (E_v, [<q>]))*, E_v, (E_v $\mid$! (E_v, [<q>]))*, [<q>])
$\mid$ AND((E_v, (E_v $\mid$! (E_v, [<q>]))*, [<q>])
$\mid$ OR((E_v)$^+$, [<q>])
$\mid$ (<primitive-event>, [<q>])
<primitive-event> ::= E_1 $\mid$ E_2 $\mid$ ...
<var> ::= event variable e_i
<q>::= (<elemqual>)*
<elemqual> ::= <var>.attr <op> <var>.attr $\mid$
<var>.attr <op> constant
<op> ::= $<$ $\mid$ $>$ $\mid$ $\leq$ $\mid$ $\geq$ $\mid$ $=$ $\mid$ $!=$
<window>::= time duration w

Table 1. NEEL Query Language

The BNF syntax for *NEEL* is shown in Table 1. In *NEEL*, the PATTERN clause retrieves event instances specified in the event expression from the input stream. The qualification in the PATTERN clause further filters event instances by evaluating predicates applied to event attributes. The WITHIN clause specifies a time period within which all the events of interest must occur in order to be considered a match. In our language, the time period is expressed as a sliding window, though other window types could be easily plugged in. A set of histories is returned with each history equal to one query match, i.e., the set of event instances that together form a valid match of the query specification. Clearly,

[1] NEEL stands for **Ne**sted Complex **E**vent Query **L**anguage

additional transformation of each match could be plugged in to the RETURN clause.

Operators in the PATTERN clause. SEQ in the PATTERN clause specifies a sequence indicating the particular order in which the event instances of interest should occur. The components of the sequence are the occurrences and non-occurrences of events [1]. Any component of SEQ including at the start or the end of the pattern can be negated using "!". **AND** also specifies events occurrences and non-occurrences but their order does not matter. **OR** operator specifies disjunction of events.

We now explain step by step the proposed *NEEL* language using the earlier RFID-based hospital tool management scenario. Again, RFID tags are assumed to be either embedded in or attached to surgical knives, clamps, scissors, etc. Sensors transmit the events performed on the equipment in an input event stream of event types washing, sharpening, disinfection, etc. For example, the query Q_4 below detects activity related to surgical knife management. The PATTERN clause contains a SEQ construct that specifies a sequence consisting of a Recycling, a Washing instance followed by the occurrence of an Operating event instance.

```
Q4 = PATTERN SEQ(Recycle r, Washing w, Operating o)
```

Nested expressions and variable scope. If E_1, E_2 ,..., E_n are event expressions, an application of SEQ, AND and OR over these event expressions is again an event expression [7]. In other words, nesting of AND, OR and SEQ operators is supported. An event expression exp_i can be used as an **inner** component to construct an **outer** expression exp_j. The operator construct optionally also includes an event variable (<var>). The benefits of using such an event variable are that it is (1) more concise to refer to an event expression in a predicate, (2) easier for the user to interpret predicates, (3) and avoids ambiguity if the same expression occurs twice, e.g., $Washing\ w_1$, $Washing\ w_2$. The event variable in an outer expression exp_j is visible within the outer expression exp_j as well as within the scope of any of its own nested inner expressions exp_i. For example, the PATTERN clause of the query Q_5 extends the query Q_4 by nesting the occurrence of a sub-sequence consisting of a Sharpening, a Disinfection and a Checking instance within the outer sequence of a Recycling, a Washing and an Operating event instance. Event instances "r", "w" and "o" declared in the outer SEQ expression are visible both in the outer and inner SEQ operators. Event instances "s", "d", "c" declared in the inner SEQ expression are visible only within this inner SEQ operator.

```
Q5 = PATTERN SEQ(Recycle r, Washing w,
                 SEQ(Sharpening s, Disinfection d, Checking c),
                 Operating o)
     WITHIN 2 hours
```

Window Constraints. We currently work with simple sliding windows, though other window models could be adopted in the future. The window constraint in the WITHIN clause imposes a time duration constraint on all instances involved in a match of the query. For a nested event expression, the same window clause w is applied to all nested subexpressions as a constraint. However, this window constraint w will be further restricted implicitly in each nested subexpression based on its context within its outer expression. For example, the window for the query Q_5 is 2 hours. The window for the subexpression SEQ(Sharpening s, Disinfection d, Checking c) is bounded by the timestamps of events w and o, namely by the interval [w.te, o.ts]. Explicit time windows for the inner SEQ can also be supported in the future without violating the window constraints of the outer nested sequences.

Predicate specification. The optional qualification in the PATTERN clause, denoted by *qual*, contains one or more predicates. Predicates only referring to events in the local expression exp_i (**simple predicates**) are specified directly inside exp_i. Predicates referring to event instances both from an outer and an inner expression are **correlated predicates**. They must be placed with the innermost expression where a variable used in the expression is declared. For example, in Q_6 the correlated predicate "s.id=d.id=c.id=o.id" referring to both inner ("s", "d" and "c") and outer ("o") events must be placed within the inner SEQ operator where any of the variables are defined. The simple predicate "o.ins-type = surgery" is placed with the outer SEQ operator where the variable "op" is declared. Predicates across the OR arguments are not allowed as only one of the OR arguments will match at a time. Correlated predicates involving two sibling expressions are not allowed since the event instances in one expression are not visible within the scope of the other expression.

```
Q6 = PATTERN SEQ(Recycle r, Washing w,
             SEQ(Sharpening s, Disinfection d, Checking c,
                 s.id=d.id=c.id=o.id),
             Operating o, o.ins-type="surgery")
```

Q_7 below is not a valid query as the subexpression SEQ(Washing d, Sharpening s) contains a correlated predicate (w.id = c.id) referring to the Checking event c which is not within the scope of this predicate because it is declared in a sibling SEQ operator. Similarly, the event w is not within its proper scope, yet is referred to, in the subexpression SEQ(Disinfection d, Checking c).

```
Q7 = PATTERN SEQ(Recycle r,
             SEQ(Washing w, Sharpening s, w.id = c.id),
             SEQ(Disinfection d, Checking c, d.id = w.id))
```

Negation. The symbol "!" before an event expression E_i expresses the non-occurrence of E_i and indicates that E_i is not allowed to appear in the specified position. If there is a ! (Negation) symbol before an event expression, we now say that the event expression marked by ! is a **negative event expression**, otherwise, it is a **positive event expression**. At least one positive event expression must exist in SEQ and AND operators. Event instances that satisfy the positive event expressions of a query with no events existing in the input stream satisfying the negative event expressions in the specified positions are said to be a **valid match**. For example, the query Q_8 specifies the non-occurrence of Washing events anywhere between Recycle and Operating events.

```
Q8 = PATTERN SEQ(Recycle r, ! Washing w, Operating o)
```

If several adjacent event types are marked by ! in a SEQ operator such as in Q_9 below, the query requires the non-existence of **any** $Washing$ and *Sharpening* events between our matched pair of *Recycle* and *Operating* event instances. In other words, SEQ(*Recycle* r, ! $Washing$ w, ! *Sharpening* s, *Operating* o) is equivalent to SEQ(*Recycle* r, ! *Sharpening* s, ! $Washing$ w, *Operating* o) as no ordering constraint holds between $Washing$ and *Sharpening* events. Events of either types can't exist in the location between our *Recycle* and *Operating* matches.

```
Q9 = PATTERN SEQ(Recycle r, ! Washing w, ! Sharpening s, Operating o)
```

Scoping of Negation. Pattern matching involving negation is different from matching on positive event types. Let us consider query Q_4 with only positive event types. It looks for the existence of Recycle, Washing and Operating events in the proper order. But Q_8 is different. We do not look for three instances, the first matching Recycle, the second matching ! Washing, and the third matching Operating. If we were to follow this interpretation, for event history H = $\{r_1, w_2, s_3, o_4\}$, we would return $\{r_1,\ o_4\}$, since s_3 would match "! Washing". However, $\{r_1,\ o_4\}$ should not be returned in this case because $w_2 \in$ Washing exists between r_1 and o_4. Clearly, the role of the "!" Washing in this context is different from the role of positive event types in the same position. Mainly, the "!" in a SEQ operator has a "for all" semantics and not an "exists" semantics. Put differently, in Q_8, "! Washing" doesn't mean matching one particular ! Washing instance between Recycle and Operating events. Rather, the query requires the non-existence of Washing events anywhere in the input stream between the matched Recycle and Operating events.

Nested Negation. A negative event expression exp_i can be used as an inner expression to filter out the construction of other outer event expression exp_j. For example, in Q_{10} the negative event type "Disinfection" is a sub-component of the negative event expression SEQ(*Sharpening* s, ! $Disinfection$ d, *Checking* c). The later in turn is a sub-component of the outermost SEQ expression of Q_{10}. Q_{10} states that $< r, w, o >$ is a valid match if either no *Sharpening* and *Checking* event pairs exist in the input stream between our *Washing* w and *Operating* o events in the outer match $< r, w, o >$, or otherwise if they do exist, then disinfection events must also exist between all *Sharpening* and *Checking* event pairs.

```
Q10 = PATTERN SEQ(Recycle r, Washing w,
             ! SEQ(Sharpening s, ! Disinfection d, Checking c),
             Operating o)
```

Predicates with Negation. Consider the query Q_{11} below.

```
Q11 = PATTERN SEQ(Recycle r, ! Washing w, Operating o,
             r.attr1 + w.attr1 = o.attr1)
```

Assume the history H = { r_1, o_5} and r_1.attr1 = 1 and o_5.attr1 = 1. Here we assume that no negative *Washing* events exist in this history. Should query Q_{11} return {r_1, o_5}? The question is how do we decide whether the condition is true or false since there is no value for a Washing event to participate in the predicate?

One answer might be that the predicate (r.attr1 + w.attr1 = o.attr1) will be treated as true whenever it refers to attributes of negative events (like w.attr1 above). This would lead to awkward semantics because the logic of the predicate will be unexpected. For example, if P is any formula and w is the "excluded" event, then (P ∨ (w.attr1 !=w.attr1)) will evaluate to true. This logic would clearly not be sensible. Or, we could have the above predicate evaluate to false. {r_1, o_5} would not be returned in this case as a result. It is also unexpected as no Washing events exist so no sequence results of the outer positive event expression should be filtered. Instead, we adopt a third strategy of interpreting nesting similar in spirit of our interpretation described above for the ! symbol in the primitive case.

```
Q12 = PATTERN SEQ(Recycle r, Washing w,
              ! SEQ(Sharpening s, Disinfection d, Checking c),
                Operating o)
      WITHIN  1 hours

Q13 = PATTERN SEQ(Recycle r, Washing w,
              ! SEQ(Sharpening s, Disinfection d, Checking c,
                    s.id=d.id=c.id=o.id),
                Operating o, r.id=w.id=o.id)
      WITHIN  1 hours
```

For this, we now propose that **the way we write predicates influences its meaning and thus its results.** Simple predicates involving negative event types are placed with negative event types. We require that all predicates referring to only positive events are stated separately, as they refer to instances that must exist. And we require all predicates involving negative event types are stated separately with the negative event types. During pattern matching, we first match events of the positive event expression and their predicates. If and only if we find a match, then we check events for the non-existence of instances to match the negative event expression and thus we then check their associated predicates. Assume the history H = $\{r_1, w_2, s_3, d_4, c_5, o_6\}$ and r_1.id = w_2.id = o_6.id = 1, s_3.id = 2, d_4.id = 3 and c_5.id = 4. Assume one user requires all event instances in Q_{12} have the same id. If the user put the condition "r.id = w.id= o.id = s.id = d.id = c.id" in the end of the outer SEQ in Q_{12}, no sequence results for the positive event expression SEQ(Recycle r, Washing w, Operating o) are constructed as the predicate is not satisfied. However, when the user represents the predicate as "r.id=w.id=o.id" associated with the outer SEQ expression, and "s.id=d.id=c.id=o.id" associated with the inner SEQ expression as shown in Q_{13}, during Q_{13} pattern evaluation, we first construct the outer sequence $< r_1, w_2, o_6 >$ with r_1.id = w_2.id = o_6.id. Then we check between w_2 and o_6 if one or more matches for the inner expression a SEQ(Sharpening s, Disinfection d, Checking c) sequence exists with “s.id=d.id=c.id=o_6.id”. $< r_1, w_2, o_6 >$ is a match for Q_{13} as no such inner sequence is found.

```
Q14 = PATTERN SEQ(Recycle r, Washing w,
                  ! SEQ(Sharpening s, Disinfection d,
                        ! (Checking c, c.id = d.id),
                        s.id=d.id=o.id),
                  Operating o, r.id=w.id=o.id)
      WITHIN  1 hours
```

When a negative event type is nested in another negative component such as the *Checking* event in Q_{14}, the user is required to put predicate requirements for the *Checking* event directly with the *Checking* event type. Predicates referring to *Sharpening* and *Disinfection* events but not involving the *Checking* event are specified in the end of the inner SEQ operator as shown in Q_{14}. During Q_{14} pattern evaluation, we first construct outer <r, w, o> sequences with r.id=w.id=o.id. Then we check between w and o pairs if one or more matches for the inner expression a SEQ(Sharpening s, Disinfection d, ! Checking c) sequence exists with “s.id=d.id=o.id” and “c.id = d.id”. If not, <r, w, o> sequences are matches for Q_{14}. If yes, the query filters this intermediate match.

4 Formal Semantics of NEEL

We now define the operator semantics using the notion of event histories. Below, we define the set of operators that *NEEL* supports in the PATTERN clause of a query and the semantics of the expressions that they form. Below E_i represents an event expression of either a primitive or composite event type.

For closure, the input and output data types are the same. The cross product of event histories A[H] and B[H] is a power set over H denoted by A[H] $\times$ B[H]. This in turn could be the input of another operator which then would generate a power set over H again by working on one event history at a time (pow(H) $\rightarrow$ pow(H)).

Definition 1. *Assume the window size for a nested event expression is w. For sliding window semantics, at any time t, we apply a query to the window constrained event history H_w = H[ts, te] with te := t and ts := t-w such that:*

$$H_w = \{e | e \in H \wedge (ts \leq e.ts \leq e.te \leq te)\}. \tag{1}$$

Definition 2. $E_i[H_w]$ *selects events of event type E_i from H_w.*

$$E_i[H_w] \ = \ \{\{e\} | e \in H_w \wedge e.type = E_i\}. \tag{2}$$

Definition 3. *Union of event histories. $H_1 \cup H_2 = \{ \ e_i \mid e_i \in H_1 \vee e_i \in H_2 \ \}$. Duplicates of e_i that appear in both histories H_1 and H_2 are removed from the result set.*

The notation $\overrightarrow{e_{1,n}}$ denotes an ordered sequence of event instances e_1, e_2, ... , e_n such that for all pairs (e_i, e_j) with i < j in the sequence, e_i.ts $\leq$ e_i.te < e_j.ts $\leq$ e_j.te holds. The notation $\widehat{e_{1,n}}$ denotes a set of event instances $\{e_1, e_2, ..., e_i, ..., e_n\}$ without any ordering constraints. The notation $\uplus E_{1,n}$ denotes the cross product (defined in Section 2) of event histories. Namely, $\uplus E_{1,n}[H_w] = E_1[H_w] \times E_2[H_w]$ $\times$ ... $E_i[H_w] \times \ldots \times E_n[H_w]$. We use the notation $\mathcal{P}_{1,j}(\widehat{e_{1,n}})$ to refer to predicates P_1, ..., P_j on events $\{e_1, \ldots, e_n\}$. Namely, $\mathcal{P}_{1\ldots j}(\widehat{e_{1,n}}) = P_1(\widehat{e_{1,n}})$,..., $P_j(\widehat{e_{1,n}})$.

Definition 4. *SEQ specifies a particular order in which the event instances of interest* e_1, e_2 ,..., e_n = $\overrightarrow{e_{1,n}}$ *should occur.*

$$\begin{aligned} &SEQ(E_1\ e_1, E_2\ e_2, ..., E_i\ e_i, \ldots, E_n\ e_n, \mathcal{P}_{1\ldots m}(\widehat{e_{1,n}}))[H_w] \\ &\quad = \{\widehat{e_{1,n}} | (\overrightarrow{e_{1,n}} \in \uplus E_{1,n}[H_w]) \wedge \mathcal{P}_{1\ldots m}(\widehat{e_{1,n}})\}. \end{aligned} \tag{3}$$

Example 1. Given SEQ(Recycle r, Washing w) and $H = \{r_1, w_2, w_3\}$, *SEQ(Recycle r, Washing w)[H] generates 2 histories:* $\{r_1, w_2\}$ *and* $\{r_1, w_3\}$.

Definition 5. *Equation 4 defines the SEQ operator with negation.*

$$\begin{aligned} &SEQ(E_1\ e_1, ..., E_i\ e_i, !(E_{i+1}\ e_{i+1}, P_{i+1}(\widehat{e_{1,n}})), E_{i+2}\ e_{i+2}, ..., E_n\ e_n, \mathcal{P}_{1\ldots i}(\widehat{e_{1,n}}), \mathcal{P}_{i+2\ldots m}(\widehat{e_{1,n}}))[H_w] \\ &\quad = \{\{e_1, ..., e_i, e_{i+2}, ..., e_n\} | (\{e_1, ..., e_i, e_{i+2}, ..., e_n\} \in (\uplus E_{1,i}[H_w] \times \uplus E_{i+2,n}[H_w])) \wedge \\ &\quad (\overrightarrow{e_{1,i}} \wedge \overrightarrow{e_{i+2,n}} \wedge e_i.te < e_{i+2}.ts) \wedge \mathcal{P}_{1\ldots i}(\widehat{e_{1,n}}) \wedge \mathcal{P}_{i+2\ldots m}(\widehat{e_{1,n}}) \wedge (\neg \exists e_{i+1} \in E_{i+1}[H_w] \\ &\quad where\ (e_i.te < e_{i+1}.ts < e_{i+1}.te < e_{i+2}.ts) \wedge \mathcal{P}_{i+1}(\widehat{e_{1,n}}))\}. \end{aligned} \tag{4}$$

Equation 4 defines the SEQ operator with negation in the middle of a list of event types. P_{i+1} involves predicates referring at least once to an instance of type E_{i+1}. In Equation 4, events $\{e_1, ..., e_i, e_{i+2}, ..., e_n\}$ of the positive event expression satisfying the associated predicates are first constructed. We then check the non-existence of E_{i+1} instances satisfying the predicate P_{i+1} with timestamps between e_i and e_{i+2} events.

Negation could equally exist at the start or the end of the SEQ operator. If negation exists at the start, the non-existence left time bound should be $e_n.te - w$. Similarly, if negation exists at the end, the non-existence right time bound should be $e_1.ts + w$. If negations are specified at both the start and the end of the SEQ operator, we need to bound them conservatively into both directions simultaneously from the leftmost and the rightmost positive components. Multiple negations could exist in the SEQ operator. For example, SEQ(E_1 e_1 ,..., E_i e_i, ! (E_{i+1} e_{i+1}, $P_{i+1}(\widehat{e_{1,n}})$), ! ($E_{i+2}$ e_{i+2}, $P_{i+2}(\widehat{e_{1,n}})$), ..., ! ($E_{i+j}$ e_{i+j}, $P_{i+j}(\widehat{e_{1,n}})$), E_{i+j+1} e_{i+j+1} ,..., E_n e_n, $\mathcal{P}_{1\ldots m}(\widehat{e_{1,n}})$). It requires the non-existence of E_{i+1}, E_{i+2}, ...and E_{i+j} event instances between e_i and e_{i+j+1} with those qualifications.

.

Definition 6. *AND operator computes the cross product of the input events.*

$$AND(E_1\ e_1, E_2\ e_2, ...E_n\ e_n, \mathcal{P}_{1...m}(\widehat{e_{1,n}}))[H_w] \\ = \{\widehat{e_{1,n}} | (\widehat{e_{1,n}} \in \uplus E_{1,n}[H_w]) \wedge \mathcal{P}_{1...m}(\widehat{e_{1,n}})\}. \quad (5)$$

Definition 7. *Equation 6 defines the AND operator with negation.*

$$AND(E_1\ e_1, ..., E_i\ e_i, !(E_{i+1}\ e_{i+1}, P_{i+1}(\widehat{e_{1,n}})), E_{i+2}\ e_{i+2}, ..., E_n\ e_n, \mathcal{P}_{1...i}(\widehat{e_{1,n}}), \mathcal{P}_{i+2...m}(\widehat{e_{1,n}}))[H_w] \\ = \{\{e_1, ..., e_i, e_{i+2}, ..., e_n\} | (\{e_1, ..., e_i, e_{i+2}, ..., e_n\} \in (\uplus E_{1,i}[H_w] \times \uplus E_{i+2,n}[H_w])) \wedge \\ \mathcal{P}_{1...i}(\widehat{e_{1,n}}) \wedge \mathcal{P}_{i+2...m}(\widehat{e_{1,n}}) \wedge \neg\exists e_{i+1} where\ (e_{i+1} \in E_{i+1}[H_w] \wedge \mathcal{P}_{i+1}(\widehat{e_{1,n}}))\}. \quad (6)$$

Again, negative event expressions just like positive ones could be composed of SEQ, AND and OR operators.

Definition 8. *Formally, the set-operator OR is defined as follows. Predicates across the OR arguments are not allowed as arguments are independent, i.e., only one of the instances will constitute the result history for each match.*

$$OR(E_1\ e_1\ , ..., E_n\ e_n, \mathcal{P}_{1...m}(\widehat{e_{1,n}})[H_w] = (E_1[H_w], P_1(e_1)) \cup ... \cup \quad (E_n[H_w], P_n(e_n)). \quad (7)$$

Example 2. Assume the query Q_{15} = OR(*Checking*, *Sharpening*, *Checking*.id > 10, *Sharpening*.id > 15)[H] and the event history $H_w = \{c_1, c_2, c_6, s_8\}$ where c_1.id = 5, c_2.id = 20, c_6.id = 2 and s_8.id = 25. Then Q_{16} = (*Checking*[H_w], *Checking*.id > 10) $\cup$ (*Sharpening*[H_w], *Sharpening*.id > 15) = $\{\{c_2\}, \{s_8\}\}$.

5 E-Analytics System

E-Analytic Architecture. Figure 1 shows the overall architecture of our E-Analytics system. Input adaptors read event streams from different devices and of different formats. Queries are first compiled into query plans, then optimized and lastly submitted to the query executor for processing. The execution engine will instantiate the query plan by instantiating the corresponding physical operators in the query plan. Thereafter, execution of the query is activated which then will continuously consume the input event stream and produce complex events that match the query pattern. The resulting streams will then be fed continuously to the monitoring business applications. These applications can either have a Graphical User Interface (GUI) for visual analytics or can be console-based. The system will help track the critical conditions for scenarios described in Section 1. A more detailed case study of this technology can also be seen in Section 6.

Query Compiler. A query expressed by *NEEL* is translated into a default nested algebraic query plan composed of the following algebraic operators: Window Sequence (*WinSeq*), Window Or (*WinOr*) and Window And (*WinAnd*). The same window w is applied to all operator nodes. During query transformation,

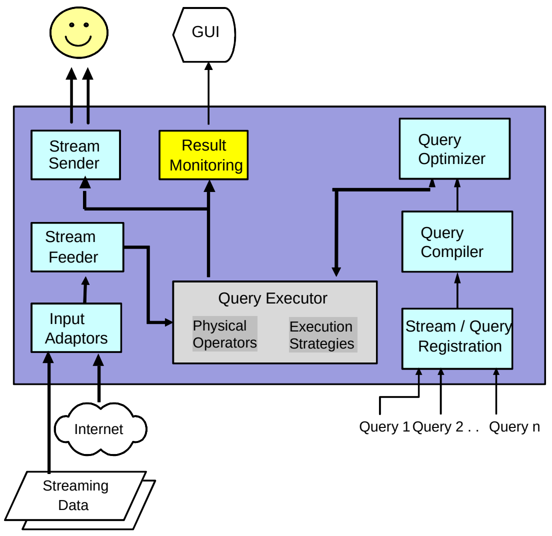

Fig. 1. E-Analytic Processing System

each expression in the event pattern is mapped to one operator node in the query plan. *WinSeq* first extracts all matches to the positive components specified in the query, and then filters out events based on negative components as specified in the query. *WinOr* returns an event *e* if *e* matches one of the event expressions specified in the *WinOr* operator. *WinAnd* computes the cross product of events of its component event expressions. The operator node for an outer expression is placed on top of the operator nodes of its inner expressions. For queries expressed by *NEEL*, predicates are placed in their proper position in nested event expressions as discussed in Section 3. A leaf node, labeled by a primitive event type E, selects instances of the primitive event type E from the input event stream S and passes them to their parent nodes.

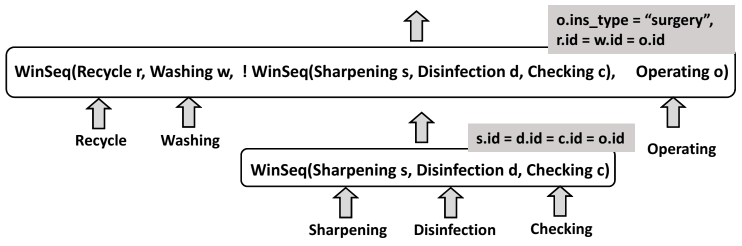
o.ins_type = "surgery",
r.id = w.id = o.id
WinSeq(Recycle r, Washing w, ! WinSeq(Sharpening s, Disinfection d, Checking c), Operating o)
Recycle
Washing
s.id = d.id = c.id = o.id
Operating
WinSeq(Sharpening s, Disinfection d, Checking c)
Sharpening
Disinfection
Checking

Fig. 2. Basic Query Plan

Example 3. Figure 2 shows the basic query plan for the sample query SEQ(*Recycle* r, *Washing* w, ! SEQ(*Sharpening* s, *Disinfection* d, *Checking* c, s.id = d.id = c.id = o.id), *Operating* o, r.id = w.id = o.id, o.ins-type = "surgery"). The two

SEQ operators are transformed into two WinSeq operator nodes in the query plan. The simple predicate o.ins-type = "surgery" is attached to the topmost WinSeq operator node containing the Operating event type. The correlated predicate is attached to the inner WinSeq operator node.

Query Executor. Following the principle of nested iteration for SQL queries [12, 13], we apply the iterative execution strategy to queries expressed by *NEEL*. The outer query is evaluated first followed by its inner sub-queries. The results of the positive inner queries are passed up and joined with the results of the outer query. For every outer partial query result, a constrained window is passed down for processing each of its children sub-queries. These sub-queries compute results involving events within the constraint window. Qualified result sequences of the inner operators are passed up to the parent operator. The outer operator then joins its own local results with those from its positive sub-queries. The outer sequence result is filtered if the result set of any of its negative sub-queries is not empty. We apply iterative execution until a final result sequence is produced by the root operator. Finally, the process repeats when the outer query consumes the next instance e. A more detailed description with preliminary experimental evaluation of the nested execution strategy can be found in [16].
Query Optimization. With precise semantics in place, we now have laid a solid foundation for developing optimization strategies for E-Analytics. For instance, in [16], selective caching of intermediate results is introduced as technique for optimizing iterative execution. In addition, interval-driven cache expansion and interval-driven cache reduction are proposed in [16].

6 Case Study of Health Care

We are collaborating with medical staff in the University of Massachusetts Memorial Medical Center in the context of a project entitled "Development and Testing of an Electronic Infection Control Reminder System for Healthcare Workers" [17]. We aim to tackle the major concerns in healthcare today of the spread of human infectious diseases in hospital settings. It has been established by the medical community that one of the simplest yet most effective methods for prevention of hospital-acquired infections is to have healthcare workers cleanse their hands and follow other precautions (such as wearing masks for H1N1, gowns, etc) before and after they see patients. Unfortunately, compliance for hand hygiene even in the best practicing hospitals in the country is below acceptable levels (75% to 80% in the best case) and methods of enforcement are minimal to non-existent.

Thus, our objective is to overcome this problem by building a Hospital Infection Control System (HICS) that continuously tracks healthcare workers throughout their workday for hygiene compliance and for paths of exposure to different diseases and unhygienic conditions. Our system would alert healthcare workers at the appropriate moments, for instance, if they are about to enter an operation room without first performing the required disinfection procedure. Such events are detected in real time by our system using queries such as Q_{17}.

```
Q17 = PATTERN SEQ(!(Sanitize-Area s, s.wID = o.wID),
                    Enter-Operating-Room o)
              WITHIN 1 minutes)
```

Similarly, in intensive care units we need the capability to conduct path analysis determining who went into an operating room, left for a break or to visit a different patient, and returned without washing and drying hands. Such logic could be expressed in *NEEL* by query Q_{18}. On finding matches, officials may then need to find out which operating rooms and/or patients are potentially at risk to undertake the needed actions to confront and remind the healthcare workers.

```
Q18 = PATTERN
      SEQ(Operating Room o1,
          OR(Break Room br, Patient Room pr,
             br.wID = o2.wID, pr.wID = o2.wID),
          ! SEQ(Washing w, Drying d, w.wID = d.wID = o2.wID),
          Operating Room o2, o1.wID = o2.wID)
      WITHIN 1 minutes
```

If a significant number of violations occur at a certain room, the supervising staff may want to review all violation patterns related to this room and deduce the potential causes of this phenomenon. Or, the supervising staff may want to identify the violator causing such an abnormal violation pattern who possibly may be a young physician or intern not well versed in required safety regulations or simply a worker neglecting usual precautions distracted by an overly busy schedule. Clearly, real-time analysis to not only track sequences of events in real time, but also to analyze their frequency relative to prior behavior of the same health care work, prior patterns by the overall staff within this intense CPU unit also across the overall care facility may be time critical, potentially, mitigating risks before they spread.

As an example of the type of services our *HyReminder* system provides, consider our real-time monitoring console in Figure 3 that displays the current hygiene compliance state of every HCW for the head nurse to supervise. The map-based monitoring window displays each worker as a moving object in the intense care unit map. Real-time statistics about the hand hygiene violations can be accessed and filtered by specifying conditions in the "view control" panel.

7 Related Work

Most event processing systems, such as SNOOP [20], do not support scope. The Cayuga [2] query language is a simple mapping of the algebra operators into a SQL like syntax, similar in spirit to the complex event language in SASE [1]. In Cayuga [2] and SASE [1], scope is expressed respectively by a duration predicate and a window clause. *NEEL* adopts basic query constructs similar to the ones in SASE [1] for expressing a flat query. Negation is not treated as an operator in *NEEL*. Instead, negation is scoped within the context of other algebra

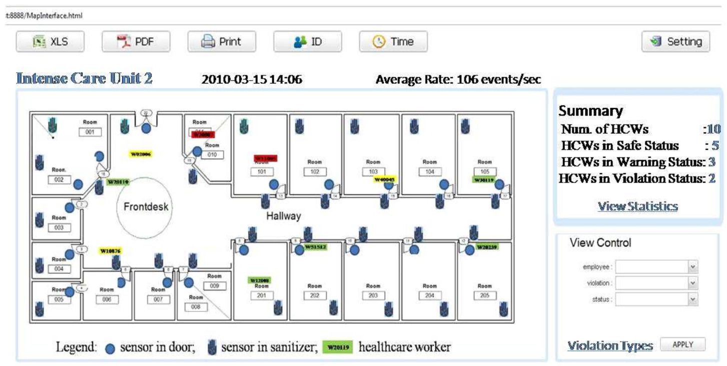

Fig. 3. Real-time Hand Hygiene Monitoring

operations to act as filter of positive matches, namely, within SEQ and AND operators. As compared to previous event languages such as *ODE* [19], SASE provides a compact CEP query language which is easy to read. For instance, "*relative*(*deposit*, ! *before interest*) *and withdraw*" is used to express the pattern "deposit followed eventually by withdraw with no intervening interest" in *ODE* [19]. Using *SASE*, the pattern can be simply expressed as PATTERN SEQ(deposit, ! interest, withdraw). However, *SASE* [1] has several limitations. It only supports SEQ and $SEQ_{WITHOUT}$ operators which allow you to express flat sequence queries. In addition, it only allows queries to transform events from primitive types to complex types, but has not looked at transforming from complex types to (even more) complex types. Put differently, *SASE* doesn't support the nesting of complex operators. This is our key focus.

While CEDR [6] allows the application of negation over composite event types, they didn't provide a clear syntax for the specification of such nested pattern queries. The *SEL* language [18], while supporting nesting of operators, focusses in particular on temporal relationship specification. Semantics of nested negation appear ambiguous as negation itself is an operator and thus a match (or, negation match) presumably would need to be returned from the negation operator if nested. Subtle issues with predicates in the presence of negation operators are not explored in [18], but can be found in our work (see Section 3).

8 Conclusions

In this paper, the CEP query language *NEEL* able to succinctly express nested queries composed of sequence, negation, AND and OR operators is presented. *NEEL* allows users to specify fairly complex queries in a compact manner with predicates and negation over query nestings both well-supported. We also introduce the formal query semantics for *NEEL*. An algebraic query plan for the execution of nested CEP queries is designed. The proposal presented here permits a simple and direct implementation of nested CEP queries following the principle of nested query execution for SQL queries. Our case study in health care confirms the utility of applying nested complex event processing support for enabling real-time event analytics.

9 Acknowledgements

This work is supported by HP Labs Innovation Research Program and National Science Foundation under grants NSF 1018443 and NSF IIS 0917017, Turkish foundation under TUBITAK career award 109E194. We thank Di Wang, Han Wang and Richard T. Ellison III for the case study. We thank Database System Research Group members at WPI for many valuable comments.

References

1. E. Wu, Y. Diao, and S. Rizvi, High-performance complex event processing over streams, in SIGMOD Conference, 2006, pp. 407-418.
2. A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White, Cayuga: A general purpose event monitoring system. in CIDR, 2007, pp. 412-422.
3. J. M. Boyce and D. Pittet, Guideline for hand hygiene in healthcare settings, MMWR Recomm Rep., vol. 51, 2002, pp. 1-45.
4. Shnayder, V., Chen, B., Lorincz, K., Fulford-Jones, T.R.F., and Welsch, M. Sensor Networks for Medical Care. in Harvard University Technical Report TR-08-05, 2005.
5. J. A. Stankovic, Q. Cao, et al., Wireless sensor networks for in-home healthcare: Potential and challenges, In Proceedings of HCMDSS Workshop, 2005.
6. R. S. Barga, J. Goldstein, M. Ali, and M. Hong, Consistent streaming through time: A vision for event stream processing, in CIDR, 2007, pp. 363-374.
7. S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.K. Kim, Composite events for active databases: Semantics, contexts and detection, in VLDB, 1994, pp. 606-617.
8. C. Gupta, S. Wang, I. Ari, M. Hao, U. Dayal, A. Mehta, M. Marwah, and R. Sharma, Chaos: A data stream analysis architecture for enterprise applications, in CEC09, 2009, pp. 33-40.
9. I. inetats. stock trade traces. http://www.inetats.com/.
10. P. Seshadri, H. Pirahesh, and T. Y. C. Leung, Complex query decorrelation, in ICDE 96, pp. 450-458.
11. C. Beeri and R. Ramakrishnan, On the Power of Magic, J. Log. Program., vol. 10, 1991, pp. 255-299.
12. E. Wong and K. Youssefi, Decomposition - a strategy for query processing, ACM Trans. Database Syst., vol. 1, no. 3, 1976, pp. 223-241.
13. J. M. Smith and P. Y.-T. Chang, Optimizing the performance of a relational algebra database interface, Commun. ACM, vol. 18, no. 10, 1975, pp. 568-579.
14. R. Guravannavar, H. S. Ramanujam, and S. Sudarshan. Optimizing nested queries with parameter sort orders. In VLDB, 2005, pp. 481-492.
15. P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex query decorrelation. In ICDE, pp. 450-458. IEEE Computer Society, 1996.
16. M. Liu, M. Ray, E. Rundensteiner, D. Dougherty, et al, Processing strategies for nested complex sequence pattern queries over event streams, 7th International Workshop on Data Management for Sensor Networks (DMSN'2010).
17. D. Wang, E. Rundensteiner, R. Ellison III, Active complex event processing: applications in realtime health care, VLDB (demonstration paper), 2010.
18. D. Zhu and A.S. Sethi, SEL - A new event pattern specification language for event correlation, Proc. ICCCN-2001, Tenth International Conference on Computer Communications and Networks, 2001, pp. 586-589.
19. Narain H. Gehani, H. V. Jagadish, Oded Shmueli: Composite event specification in active databases: model & implementation. VLDB 1992, pp. 327-338
20. Sharma Chakravarthy, V. Krishnaprasad, etc, Composite events for active databases: semantics, contexts and detection. VLDB 1994, pp. 606-617