A Declarative Framework for Matching Iterative and Aggregative Patterns against Event Streams

Darko Anicic[1], Sebastian Rudolph[2], Paul Fodor[3], and Nenad Stojanovic[1]

[1] FZI Research Center for Information Technology, Germany
[2] AIFB, Karlsruhe Institute of Technology, Germany
[3] State University of New York at Stony Brook, USA

Abstract. Complex Event Processing as well as pattern matching against streams have become important in many areas including financial services, mobile devices, sensor-based applications, click stream analysis, real-time processing in Web 2.0 and 3.0 applications and so forth. However, there is a number of issues to be considered in order to enable effective pattern matching in modern applications. A language for describing patterns needs to feature a well-defined semantics, it needs be rich enough to express important classes of complex patterns such as iterative and aggregative patterns, and the language execution model needs to be efficient since event processing is a real-time processing. In this paper, we present an event processing framework which includes an expressive language featuring a precise semantics and a corresponding execution model, expressive enough to represent iterative and aggregative patterns. Our approach is based on a logic, hence we analyse deductive capabilities of such an event processing framework. Finally, we provide an open source implementation and present experimental results of our running system.

1 Introduction

Pattern matching against event streams is a paradigm of processing continuously arriving events with the goal of identifying meaningful *patterns* (complex events). For instance, occurrence of multiple events form a complex event pattern by matching certain *temporal*, *relational* or *causal* conditions. Complex Event Processing (CEP) has recently aroused significant interest due to its wide applicability in areas such as financial services (e.g., dynamic tracking of stock fluctuations, surveillance for frauds and money laundering etc.), sensor-based applications (e.g., RFID monitoring), network traffic monitoring, Web click analysis etc.

While the pattern matching over continuously arriving events has been well studied [1,10,5,6,9], so far the focus was mostly on the high-performance and the pattern language expressivity. A common approach for stream query processing has been to use select-join-aggregation queries [5,6,9]. While such queries can specify a wide range of patterns, they are unable to express *Kleene* closure. Kleene closure can be used to extract from the input stream a finite yet

N. Bassiliades et al. (Eds.): RuleML 2011 - Europe, LNCS 6826, pp. 138–153, 2011.

unbounded number of events with a particular property. Recent study [1] has presented that non-deterministic finite automate (NFA) are suitable for pattern matching, including also the matching on unbounded events streams.

In this work, we propose a *logic rule-based* approach that supports the class of patterns expressible with select-join-aggregation queries, as well as with Kleene closure and transitive closure. In our formalism these patterns are realized as *iterative* rules.

We advocate here a *logic rule-based* approach because a rule-based formalism is expressive enough and convenient to represent diverse complex event patterns. Rules can easily express complex relationships between events by matching certain temporal, relational or causal conditions. Detected patterns may further be used to build more complex patterns (i.e., the head of one rule may be used in the body of other rule, thereby creating more and more complex events). Also declarative rules are free of side-effects. Moreover, with our rule-based formalism it is possible to realize not only a set of event patterns, but rather the whole *event-driven* application (realized in a single, uniform formalism). Ultimately, a logic-based event model enables *reasoning* over events, their relationships, entire state, and possible contextual *knowledge*. This knowledge captures the *domain* of interest, or *context* related to business critical actions and decisions (that are triggered in real-time by complex events). Its purpose is to be evaluated during detection of complex events in order to *enrich* recorded events with background information; to detect more complex *situations*; to propose certain intelligent *recommendations* in real-time; or to accomplish complex event *classification*, *clustering*, and *filtering*.

Our approach is based on an efficient, *event-driven*, model for detecting event patterns. The model has inference capabilities and yet good run-time characteristics (comparable or better than approaches with no reasoning capabilities). It provides a flexible transformation of complex patterns into *intermediate patterns* (i.e., *goals*) updated in the dynamic memory. The status of achieved goals at the current state shows the progress toward matching of one or more event patterns. Goals are automatically asserted as relevant events occur. They can persist over a period of time "waiting" in order to support detection of a more complex goal or complete pattern. Important characteristics of these goals are that they are asserted only if they are used later on (to support a more complex goal or an event pattern), goals are all unique, and goals persist as long as they remain relevant (after the relevant period they are deleted). Goals are asserted by declarative rules, which are executed in the backward chaining mode. We have implemented the proposed language in a Prolog-based prototype called ETALIS, and evaluated the implementation in Section 4.

2 A Language for Complex Event Processing

We have defined a basic language for CEP in [4]. In this and the following sections, we extend the language to handle *iterative* and *aggregative* event patterns. In order to keep the presentation of the overall formalism self-contained, in this section we also recall basics of the language from [4].

The syntax and semantics of the ETALIS formalism features (i) static rules accounting for static background information about the considered domain and (ii) event rules that are used to capture the dynamic information by defining patterns of complex events. Both parts may be intertwined through the use of common variables. Based on a combined (static and dynamic) specification, we will define the notion of entailment of complex events by a given event stream.

We start by defining the notational primitives of the ETALIS formalism. An ETALIS rule base is based on:

- a set $\mathbf{V}$ of *variables* (denoted by capitals X, Y, ...)
- a set $\mathbf{C}$ of *constant symbols* including *true* and *false*
- for $n \in \mathbb{N}$, sets $\mathbf{F}_n$ of *function symbols* of arity n
- for $n \in \mathbb{N}$, sets $\mathbf{P}^{\mathrm{s}}_n$ of *static predicates* of arity n
- for $n \in \mathbb{N}$, sets $\mathbf{P}^{\mathrm{e}}_n$ of *event predicates* of arity n, disjoint from $\mathbf{P}^{\mathrm{s}}_n$

Based on those, we define *terms* by:

$$t ::= v \mid c \mid \mathtt{p}^{\mathrm{s}}_n(t_1, \ldots, t_n) \mid \mathtt{f}_n(t_1, \ldots, t_n)$$

We define the set of *(static or event) atoms* as the set of all expressions $\mathtt{p}_n(t_1, \ldots, t_n)$ where $\mathtt{p}$ is a (static or event) predicate and $t_1, \ldots t_n$ are terms.

An ETALIS *rule base* $\mathcal{R}$ is composed of a static $\mathcal{R}^{\mathrm{s}}$ and an event part $\mathcal{R}^{\mathrm{e}}$. Thereby, $\mathcal{R}^{\mathrm{s}}$ is a set of Horn clauses using the static predicates $\mathbf{P}^{\mathrm{s}}_n$. Formally, a *static rule* is defined as $a :- a_1, \ldots, a_n$ with $a, a_1, \ldots, a_n$ static atoms. Thereby, every term that a contains must be a variable. Moreover, all variables occurring in any of the atoms have to occur at least once in the rule body outside any function application.

The event part $\mathcal{R}^{\mathrm{e}}$ allows for the definition of patterns based on *time* and *events*. Time instants and durations are represented as nonnegative rational numbers $q \in \mathbb{Q}^+$. Events can be atomic or complex. An *atomic event* refers to an instantaneous occurrence of interest. Atomic events are expressed as ground event atoms (i.e., event predicates the arguments of which do not contain any variables). Intuitively, the arguments of a ground atom representing an atomic event denote information items (i.e. event data) that provide additional information about that event.

Atomic events are combined to *complex events* by *event patterns* describing temporal arrangements of events and absolute time points. The language P of event patterns is defined by

$$\begin{aligned} P ::= {} & \mathtt{p}^{\mathrm{e}}(t_1, \ldots, t_n) \mid P \text{ WHERE } t \mid q \mid (P).q \\ & \mid P \text{ BIN } P \mid \text{NOT}(P).[P, P] \end{aligned}$$

Thereby, $\mathtt{p}^{\mathrm{e}}$ is an n-ary event predicate, t_i denote terms, t is a term of type boolean, q is a nonnegative rational number, and BIN is one of the binary operators SEQ, AND, PAR, OR, EQUALS, MEETS, DURING, STARTS, or FINISHES[1]. As a

[1] Hence, the defined pattern language captures all possible 13 relations on two temporal intervals as defined in [2].

side condition, in every expression p WHERE t, all variables occurring in t must also occur in the pattern p.

Finally, an *event rule* is defined as a formula of the shape

$$\mathtt{p}^{\mathrm{e}}(t_1, \ldots, t_n) \leftarrow p$$

where p is an event pattern containing all variables occurring in $\mathtt{p}^{\mathrm{e}}(t_1, \ldots, t_n)$.

We define the declarative formal *semantics* of our formalism in a model-theoretic way. Note that we assume a fixed interpretation of the occurring function symbols, i.e. for every function symbol f of arity n, we presume a predefined function $f^* : Con^n \to Con$. That is, in our setting, functions are treated as built-in utilities.

As usual, a *variable assignment* is a mapping $\mu : Var \to Con$ assigning a value to every variable. We let μ^* denote the canonical extension of μ to terms:

$$\mu^* : \begin{cases} v \mapsto \mu(v) & \text{if } v \in Var, \\ c \mapsto c & \text{if } c \in Con, \\ f(t_1, \ldots, t_n) \mapsto f^*(\mu^*(t_1), \ldots, \mu^*(t_n)) & \text{for } f \in \mathbf{F}_n, \\ \mathtt{p}(t_1, \ldots, t_n) \mapsto \begin{cases} \mathit{true} & \text{if } \mathcal{R}^{\mathrm{s}} \models p(\mu^*(t_1), \ldots, \mu^*(t_n)), \\ \mathit{false} & \text{otherwise.} \end{cases} \end{cases}$$

Thereby, $\mathcal{R}^{\mathrm{s}} \models p(\mu^*(t_1), \ldots, \mu^*(t_n))$ is defined by the standard least Herbrand model semantics.

In addition to $\mathcal{R}$, we fix an *event stream*, which is a mapping $\epsilon : \mathit{Ground}^{\mathrm{e}} \to 2^{\mathbb{Q}^+}$ from event ground predicates into sets of nonnegative rational numbers. It indicates what elementary events occur at which time instants.

Moreover, we define an interpretation $\mathcal{I} : \mathit{Ground}^{\mathrm{e}} \to 2^{\mathbb{Q}^+ \times \mathbb{Q}^+}$ as a mapping from the event ground atoms to sets of pairs of nonnegative rationals, such that $q_1 \leq q_2$ for every $\langle q_1, q_2 \rangle \in \mathcal{I}(g)$ for all $g \in \mathit{Ground}^{\mathrm{e}}$. Given an event stream ϵ, an interpretation $\mathcal{I}$ is called a *model* for a rule set $\mathcal{R}$ – written as $\mathcal{I} \models_\epsilon \mathcal{R}$ – if the following conditions are satisfied:

C1 $\langle q, q \rangle \in \mathcal{I}(g)$ for every $q \in \mathbb{Q}^+$ and $g \in \mathit{Ground}$ with $q \in \epsilon(g)$

C2 for every rule $\mathit{atom} \leftarrow \mathit{pattern}$ and every variable assignment μ we have $\mathcal{I}_\mu(\mathit{atom}) \subseteq \mathcal{I}_\mu(\mathit{pattern})$ where $\mathcal{I}_\mu$ is inductively defined as displayed in Fig. 1.

For an interpretation $\mathcal{I}$ and some $q \in \mathbb{Q}^+$, we let $\mathcal{I}|_q$ denote the interpretation defined by $\mathcal{I}|_q(g) = \mathcal{I}(g) \cap \{\langle q1, q2 \rangle \mid q2 - q1 \leq q\}$. Given interpretations $\mathcal{I}$ and $\mathcal{J}$, we say that $\mathcal{I}$ is *preferred* to $\mathcal{J}$ if $\mathcal{I}|_q \subset \mathcal{J}|_q$ for some $q \in \mathbb{Q}^+$. A model $\mathcal{I}$ is called *minimal* if there is no other model preferred to $\mathcal{I}$. Obviously, for every event stream ϵ and rule base $\mathcal{R}$ there is a unique minimal model $\mathcal{I}^{\epsilon,\mathcal{R}}$.

Finally, given an atom a and two rational numbers q_1, q_2, we say that the event $a^{[q_1,q_2]}$ is a *consequence* of the event stream ϵ and the rule base $\mathcal{R}$ (written $\epsilon, \mathcal{R} \models a^{[q_1,q_2]}$), if $\langle q_1, q_2 \rangle \in \mathcal{I}^{\epsilon,\mathcal{R}}_\mu(a)$ for some variable assignment μ.

It can be easily verified that the behavior of the event stream ϵ beyond the time point q_2 is irrelevant for determining whether $\epsilon, \mathcal{R} \models a^{[q_1,q_2]}$ is the case[2]. This

[2] More formally, for any two event streams ϵ_1 and ϵ_2 with $\epsilon_1(g) \cap \{\langle q, q' \rangle \mid q' \leq q_2\} = \epsilon_2(g) \cap \{\langle q, q' \rangle \mid q' \leq q_2\}$ we have that $\epsilon_1, \mathcal{R} \models a^{[q_1,q_2]}$ exactly if $\epsilon_2, \mathcal{R} \models a^{[q_1,q_2]}$.

pattern	$\mathcal{I}_\mu$(pattern)
$\mathtt{pr}(t_1,\ldots,t_n)$	$\mathcal{I}(\mathtt{pr}(\mu^*(t_1),\ldots,\mu^*(t_n)))$
p WHERE t	$\mathcal{I}_\mu(p)$ if $\mu^*(t) = true$ $\emptyset$ otherwise.
q	$\{\langle q,q\rangle\}$ for all $q{\in}\mathbb{Q}^+$
$(p).q$	$\mathcal{I}_\mu(p) \cap \{\langle q_1,q_2\rangle \mid q_2 - q_1 = q\}$
p_1 SEQ p_2	$\{\langle q_1,q_4\rangle \mid \langle q_1,q_2\rangle{\in}\mathcal{I}_\mu(p_1)$ and $\langle q_3,q_4\rangle{\in}\mathcal{I}_\mu(p_2)$ and $q_2{<}q_3\}$
p_1 AND p_2	$\{\langle \mathsf{min}(q_1,q_3),\mathsf{max}(q_2,q_4)\rangle \mid \langle q_1,q_2\rangle{\in}\mathcal{I}_\mu(p_1)$ and $\langle q_3,q_4\rangle{\in}\mathcal{I}_\mu(p_2)\}$
p_1 PAR p_2	$\{\langle \mathsf{min}(q_1,q_3),\mathsf{max}(q_2,q_4)\rangle \mid \langle q_1,q_2\rangle{\in}\mathcal{I}_\mu(p_1)$ and $\langle q_3,q_4\rangle{\in}\mathcal{I}_\mu(p_2)$ and $\mathsf{max}(q_1,q_3){<}\mathsf{min}(q_2,q_4)\}$
p_1 OR p_2	$\mathcal{I}_\mu(p_1) \cup \mathcal{I}_\mu(p_2)$
p_1 EQUALS p_2	$\mathcal{I}_\mu(p_1) \cap \mathcal{I}_\mu(p_2)$
p_1 MEETS p_2	$\{\langle q_1,q_3\rangle \mid \langle q_1,q_2\rangle{\in}\mathcal{I}_\mu(p_1)$ and $\langle q_2,q_3\rangle{\in}\mathcal{I}_\mu(p_2)\}$
p_1 DURING p_2	$\{\langle q_3,q_4\rangle \mid \langle q_1,q_2\rangle{\in}\mathcal{I}_\mu(p_1)$ and $\langle q_3,q_4\rangle{\in}\mathcal{I}_\mu(p_2)$ and $q_3{<}q_1{<}q_2{<}q_4\}$
p_1 STARTS p_2	$\{\langle q_1,q_3\rangle \mid \langle q_1,q_2\rangle{\in}\mathcal{I}_\mu(p_1)$ and $\langle q_1,q_3\rangle{\in}\mathcal{I}_\mu(p_2)$ and $q_2{<}q_3\}$
p_1 FINISHES p_2	$\{\langle q_1,q_3\rangle \mid \langle q_2,q_3\rangle{\in}\mathcal{I}_\mu(p_1)$ and $\langle q_1,q_3\rangle{\in}\mathcal{I}_\mu(p_2)$ and $q_1{<}q_2\}$
NOT$(p_1).[p_2,p_3]$	$\mathcal{I}_\mu(p_2$ SEQ $p_3) \setminus \mathcal{I}_\mu(p_2$ SEQ p_1 SEQ $p_3)$

Fig. 1. Definition of extensional interpretation of event patterns. We use $p_{(x)}$ for patterns, $q_{(x)}$ for rational numbers, $t_{(x)}$ for terms and $\mathtt{pr}$ for event predicates.

justifies to take the perspective of ϵ being only partially known (and continuously unveiled along a time line) while the task is to detect event-consequences as soon as possible.

The theoretical properties of the presented formalism heavily depend on the conditions put on the formalism's signature. On the negative side, without further restrictions, the formalism turns out to be ExpTime-complete as a straightforward consequence from according results in [7]. On the other side, the formalism turns not only decidable but even tractable if both $\mathbf{C}$ and the arity of functions and predicates is bounded:

Theorem 1. *Given natural numbers k, m, the problem of detecting complex events in an event stream ϵ with an ETALIS rule base $\mathcal{R}$ which satisfies $|\mathbf{C}| \leq k$ and $\mathbf{F}_n = \mathbf{P}^{\mathrm{s}}_n = \mathbf{F}^{\mathrm{e}}_n = \emptyset$ for all $n \geq m$ is* PTIME*-complete w.r.t. $|\mathcal{R}| + |\epsilon|$.*

Proof. PTIME-hardness directly follows from the fact that the formalism subsumes function-free Horn logic which is known to be hard for PTIME, see e.g. [7].

For containment in PTIME, recall that in our formalism, function symbols have a fixed interpretation. Hence, given an ETALIS rule base $\mathcal{R}$ with finite $\mathbf{C}$, we can transform it into an equivalent function-free rule base $\mathcal{R}'$: we eliminate every n-ary function symbol $\mathtt{f}$ by introducing an auxiliary $n{+}1$-ary predicate $\mathtt{p_f}$ and "materializing" the function by adding ground atoms $\mathtt{p_f}(c_1,\ldots,c_n,\mathtt{f}^*(c_1,\ldots,c_n))$. This can be done in time polynomial time, given the above mentioned arity bound. Naturally, also the size of $\mathcal{R}'$ is polynomial compared to the size of $\mathcal{R}$.

Next, observe that under the above circumstances, the least Herbrand model of $\mathcal{R}^{s\prime}$ (which is then arity-bounded and function-free) can be computed in polynomial time (as there are only polynomially many ground atoms). Finally, note that the number of time points occurring in an event stream ϵ is linearly bounded by $|\epsilon|$, whence there are only polynomially many relevant "interval-endowed ground predicates" $a^{[q_1,q_2]}$ possibly entailed by ϵ and $\mathcal{R}^{e\prime}$. Finally these entailments can be checked in polynomial time in a forward-chaining manner against the respective (polynomial) grounding of $\mathcal{R}^{e\prime}$. This concludes the proof.

Example. The following pattern rules (1) demonstrates the usage of the ETALIS formalism by defining a common financial pattern called the "tick-shape" pattern. Let's consider a simple day trader pattern that looks for a peak followed by a continuous fall in price of stocks, followed by a rise in price. We are interested in a raise only if (and as soon as) it grows higher than the beginning price. The "tick-shape" pattern is monitored for each company symbol over online stock events, see rules (1).

$$
\begin{aligned}
\texttt{down}(I,P1,P2) &\leftarrow \text{NOT}(\texttt{stock}(I,P)).[\texttt{stock}(I,P1),\\
&\qquad \texttt{stock}(I,P2)] \text{ WHERE } P1 < P2.\\
\texttt{down}(I,P1,P3) &\leftarrow \text{NOT}(\texttt{stock}(I,P)).[\texttt{down}(I,P1,P2),\\
&\qquad \texttt{stock}(I,P3)] \text{ WHERE } P2 > P3.\\
\texttt{up}(I,P1) &\leftarrow \texttt{stock}(I,P1).\\
\texttt{up}(I,P2) &\leftarrow \text{NOT}(\texttt{stock}(I,P)).[\texttt{up}(I,P1),\texttt{stock}(I,P2)]\\
&\qquad \text{WHERE } P1 < P2.\\
\texttt{tickShape}(I) &\leftarrow \texttt{down}(I,P1,P2) \text{ MEETS}\\
&\qquad \text{NOT}(\texttt{stock}(I,P)).[\texttt{up}(I,P3),\texttt{stock}(I,P4)]\\
&\qquad \text{WHERE } P3 < P1 \ \wedge \ P4 > P1.
\end{aligned}
\tag{1}
$$

In this example, we first start detecting a short increase (in order to detect the peak) and subsequent fall in price using $\texttt{down}(I,P1,P2)$ iterative rules. Thereby, I takes the identifier of the monitored company, $P1$ the price at the peak directly preceding the decrease and $P2$ the price at the end of the interval. The usage of the NOT pattern ensures that no `stock` events in between are left out and hence, the decrease in price is monotone. Similarly we can detect a rise in price, defined by $\texttt{up}(I,P1)$ (where $P1$ assumes the price at the end of the interval). Finally, $\texttt{tickShape}(I)$ will be triggered when a `down` event meets an `up` event which ends at a prize value below the preceding peak, and the next incoming stock event for I reports a prize above that peak value.

2.1 Iterations and Aggregate Functions

In this section, we show how unbound iterations of events, possibly in combination with aggregate functions can be expressed within our defined formalism.

Many of the formalisms concerned with Complex Event Processing feature operators indicating that an event may be iterated arbitrarily often. Mostly, the notation of these operators is borrowed from regular expressions in automata theory: the *Kleene star* $(\cdot^*)$ matches zero or more occurrences whereas the *Kleene plus* $(\cdot^+)$ indicates one or more occurrences.

For example, the pattern expression a SEQ b^+ SEQ c would match any of the event sequences *abc*, *abbc*, *abbbc* etc. It is easy to see that – given our semantics – this

pattern expression is equivalent to the pattern a SEQ b SEQ c (as essentially, it allows for "skipping" occurring events)[3]. Likewise, all patterns in which this kind of Kleene iteration occurs can be transformed into non-iterative ones.

However, frequently iterative patterns are used in combination with *aggregate functions*, i.e. a value is accumulated over a sequence of events. Mostly, CEP formalisms define new language primitives to accommodate this feature. However, within the ETALIS formalism, this situation can be handled via recursive event rules.

As an example, assume an event should be triggered by a sequence of repeated selling events if the total income generated by them is above 100000$. For this, we have to sum over the single incomes indicated by the atomic selling events. This can be realized by the below set of rules.

$$\begin{aligned} \texttt{income}(Price) &\leftarrow \texttt{sell}(Item, Price). \\ \texttt{income}(P1 + P2) &\leftarrow \texttt{income}(P1) \text{ SEQ } \texttt{sell}(Item, P2). \\ \texttt{bigincome} &\leftarrow \texttt{income}(Price) \text{ WHERE } Price > 100000. \end{aligned} \tag{2}$$

In the same vein, every aggregative pattern can be expressed by sets of recursive rules, where we introduce auxiliary events that carry the intermediate results of the aggregation as arguments.

As a further remark, note that for a given natural number n, the n-fold sequential execution of an event a (a pattern usually written as a^n) can be recognized by $\texttt{iteration}(a, n)$ defined as follows:

$$\begin{aligned} \texttt{iteration}(a, 1) &\leftarrow a. \\ \texttt{iteration}(a, k+1) &\leftarrow a \text{ SEQ } \texttt{iteration}(a, k). \end{aligned} \tag{3}$$

This allows us to express patterns where events are repeated many times in a compact way.

A common scenario in event processing is to detect patterns on moving *length-based windows*. Such a pattern is detected when certain events are repeated as many times as the window length is. A sliding window moves on each new event to detect a new complex event (defined by the length of a window). Rules (4) implement such a pattern in ETALIS for the length equal to n (n is typically predefined). For instance, for n=5, e will be triggered every time when the system encounters five occurrences of a.

$$\begin{aligned} \texttt{iteration}(a, 1) &\leftarrow a. \\ \texttt{iteration}(a, k+1) &\leftarrow \text{NOT}(a).[a, \texttt{iteration}(a, k)]. \\ \texttt{e} &\leftarrow \texttt{iteration}(a, n). \end{aligned} \tag{4}$$

3 Execution Model

Complex event patterns that a user can create with the language proposed in Section 2 are not convenient to be used for *event-driven* computation. These are

[3] Note that due to the chosen semantics, this encoding would also match sequences like *acbbc* or *abbacbc*. However, if wanted, these can be excluded by using the slightly more complex pattern (a SEQ b SEQ c) EQUALS NOT(a OR c).$[a, c]$.

rather Prolog-style rules suitable for *backward chaining* evaluation. Such rules are understood as goals which, at certain time, either can or cannot be proved by an inference engine. The difficulty is that such an inference process cannot be done in an *event-driven* fashion.

Our execution model is based on a *goal-directed event-driven* rules. The approach is established on decomposition of complex event patterns into *two-input intermediate events* (i.e., *goals*). The status of achieved goals at the current state shows the progress toward completeness of an event pattern. Goals are automatically asserted by rules as relevant events occur. They can persist over a period of time "waiting" in order to support detection of a more complex goal or pattern. In the remaining part of this subsection we explain the *transformation* of user-defined patterns into goal-directed event-driven rules (i.e., executable rules capable to detect events as soon as they really occur).

Algorithm 1. Sequence

Input: event binary goal $\texttt{e} \leftarrow \texttt{a}$ SEQ $\texttt{b}$ WHERE $\texttt{t}$.
Output: event-driven backward chaining rules for SEQ operator and a static rule $\texttt{t}$.
Each event binary goal $\texttt{ie} \leftarrow \texttt{a}$ SEQ $\texttt{b}$ is converted into: {

$\texttt{a}(T_1, T_2) :- \texttt{for\_each}(\texttt{a}, 1, [T_1, T_2])$.
$\texttt{a}(1, T_1, T_2) :- \texttt{assert}(\texttt{goal}(\texttt{b}(\_, \_), \texttt{a}(T_1, T_2), \texttt{ie}(\_, \_)))$.
$\texttt{b}(T_3, T_4) :- \texttt{for\_each}(\texttt{b}, 1, [T_3, T_4])$.
$\texttt{b}(1, T_3, T_4) :- \texttt{goal}(\texttt{b}(T_3, T_4), \texttt{a}(T_1, T_2), \texttt{ie}), T_2 < T_3,$
$\texttt{retract}(\texttt{goal}(\texttt{b}(T_3, T_4), \texttt{a}(T_1, T_2), \texttt{ie}(\_, \_))), \texttt{ie}(T_1, T_4)$.
}
$\texttt{ie}(T_1, T_4) :- \texttt{t}, \texttt{e}(T_1, T_4)$.

Let us first consider a sequence of events $e \leftarrow p_1$ SEQ p_2 SEQ p_3... SEQ p_n where e is detected when an event p_1 is followed by p_2,.., followed by p_n. We can always represent the above pattern as $e \leftarrow (((p_1$ SEQ $p_2)$ SEQ $p_3)$... SEQ $p_n)$. We refer to this coupling of events as *binarization* of events. Effectively, in binarization we introduce intermediate events (goals), e.g., $ie_1 \leftarrow p_1$ SEQ p_2, $ie_2 \leftarrow ie_1$ SEQ p_3, etc. Every monitored event (either atomic or complex), including intermediate events, will be assigned with one or more *logic rules* which are fired whenever that event occurs. Using the binarization, it is more convenient to construct *event-driven* rules. First, it is easy to implement an event operator when events are considered on a "two by two" basis. Second, the binarization increases the *sharing* among events and intermediate events (when detecting complex patterns). Third, the binarization eases the management of rules. For example, each new use of an event (in a pattern) amounts to appending only one rule to the existing rule set.

Algorithm 1 accepts as input a binary sequence $\texttt{e} \leftarrow \texttt{a}$ SEQ $\texttt{b}$ WHERE $\texttt{t}$, and produces event-driven backward chaining rules (i.e., executable rules). Additionally a user needs to define a static rule for a predicate $\texttt{t}$ and add it into a rule base. As discussed, $\texttt{t}$ is application specific, and can be used for event enrichment, filtering, querying historical data, as well as for reasoning about the context.

Event-driven backward chaining rules produced by Algorithm 1 belong to two different classes of rules. We refer to the first class as to rules used to *generate goals*. The second class corresponds to *checking rules*.

When an `a` event occurs at some (T_1, T_2) it will trigger the first rule, which in turn will trigger each $\mathtt{a}(n, T_1, T_2)$[4]. In this case $n = 1$, since the `a` event is used only once in the pattern. In general there can be more than one rule of this type, e.g., $\mathtt{a}(1, T_1, T_2)...\mathtt{a}(3, T_1, T_2)$, if the `a` event appears three times in user's complex event patterns.

$\mathtt{a}(1, [T_1, T_2])$ is a rule that generates $goal(\mathtt{b}([\_, \_]), \mathtt{a}([T_1, T_2]), \mathtt{ie}([\_, \_]))$. Its interpretation is that "an event `a` has occurred at $[T_1, T_2]$[5], and we are waiting for `b` to happen in order to detect `ie`". Obviously, the goal does not carry information about times for `b` and `ie`, as we don't know when they will occur. In general, the *second* event in a goal always denotes an event that has just occurred. The role of the *first* event is to specify what we are waiting for to detect an event that is on the *third* position. $\mathtt{b}(1, [T_3, T_4])$ belongs to the *checking rules*. They check whether certain goals already exist in the database, in which case they trigger more complex events. For example, rule $\mathtt{b}(1, [T_3, T_4])$ will fire whenever b occurs. The rule checks whether $goal(\mathtt{b}([T_3, T_4]), \mathtt{a}([T_1, T_2]), \mathtt{ie}([\_, \_]))$ already exists (i.e., an `a` has previously happened), in which case it triggers `ie` (by calling $\mathtt{ie}([T_1, T_4])$. The time occurrence of `ie` (i.e. $[T_1, T_4]$) is defined based on the occurrence of constituting events (i.e. $\mathtt{a}[T_1, T_2]$, and $\mathtt{b}[T_3, T_4]$).

The $\mathtt{ie}([T_1, T_4])$ event will trigger the last rule. If the static predicate, `t`, evaluates to true, then the rule will call the `e` event. Calling $\mathtt{e}[T_1, T_4]$, this event is effectively propagated either upward (if it is an intermediate event) or triggered as a complex event.

More detailed description of event-driven computation in ETALIS (including other operators from the language too) can be found in [3]. Other issues regarding the execution model, such as the various consumption policies and memory management were also studied in [8].

3.1 Kleene Plus Closure

The main principle behind the execution model of Kleene closure is similar as for the sequence operator. To explain how this closure can be computed in ETALIS let us go back to example rules (2), Section 2.1. Algorithm 1 can be used to transform these rules into event-driven backward chaining rules, which can be directly executed by ETALIS prototype.

Essentially these rules handle an unbounded stream of *sell* $(Item, Price)$ events, compute the sum of their prices and detect *bigincome* if the sum is greater than 100000 $. The first rule sets a condition which defines when the pattern detection should start[6]. In our example it is just an occurrence of *start*

[4] By using a predicate, for_each. Implementation details for this predicate can be found in [3].

[5] Apart from the time stamp, an event may carry other data parameters that are omitted here in order to make the presentation more readable.

[6] It also sets the starting *Price* value to 0.

event (e.g. it can be at the beginning of a day, a month or just an event occurrence denoting that something significant to our business happened). An occurrence of $start([T_1, T_2])$ event[7] will unconditionally cause an occurrence of *income* event with $Price = 0$, and the same timestamp $[T_1, T_2]$. As *income* is used to build a sequence of events in the second rule, $goal(sell\ (Item, P2, [\_, \_]), income(P1, [T_1, T_2]), income(P1 + P2, [\_, \_]))$ will be inserted. The goal states that an instance of *income* event occurred at $[T_1, T_2]$, and the CEP engine waits for *sell* to happen to detect another *income* (iteratively). If *sell* occurs at some $[T_3, T_4]$, $T_2 < T_3$, a corresponding checking rule will check whether $goal(sell\ (Item, P2, [\_, \_]), income(P1, [T_1, T_2]), income(P1 + P2, [\_, \_]))$ is already in the database, in which case it will trigger $income(P1 + P2, [T_1, T_4])$ (adding price $P2$ to the current aggregated value, $P1$). Events of type *income* are intermediate events in our overall complex pattern. The third rule monitors these events in order to detect *bigincome*. The rule sets a condition which defines when the pattern detection should stop (taking into account that we deal with an unbounded stream of events).

3.2 Implementation of Iterative Rules and Common Aggregate Functions

The aggregate functions are computed incrementally, by starting with an initial value for the increment, and iterating the aggregate function over events. However, window size and the sliding window require us to use efficient data structures and algorithms in Logic Programming (e.g., in Prolog) to obtain fast implementations.

For any aggregate function we implement the following two rules.

$$\begin{aligned}
&\texttt{iteration}(StartCntr = 0, StartVal) \leftarrow \texttt{start_event}(StartVal).\\
&\texttt{iteration}(OldCntr + 1, NewVal) \leftarrow\\
&\qquad \texttt{iteration}(OldCntr, OldVal)\ \text{SEQ}\ \texttt{a}(AggArg)\\
&\qquad\ \text{WHERE}\ \{\texttt{assert}(AggArg),\\
&\qquad\qquad \texttt{window}(WndwSize, OldCntr, OldVal, AggArg, NewVal)\}.
\end{aligned} \tag{5}$$

The first rule starts the iteration process (when `start_event`) occurs with its initial value and possible condition on that value (see the first rule). The second rule defines the iteration itself, i.e., whenever an event participating in the iteration occurs (event `a`), it will trigger the rule and generate a new `iteration` event.

In each iteration it is possible to calculate certain operations (an aggregate function). To achieve this, the iterative rule contains the static part (the WHERE clause) for two reasons: to save data from the seen events as history relevant w.r.t the aggregation function (see $\texttt{assert}(AggArg)$), and to compute the sliding window incrementally (i.e., to delete events that expired from the sliding window and calculate the aggregate function on the rest, see the `window` expression).

[7] As *start* is an atomic event, $T_1 = T_2$.

The functionality of `assert` predicate is simply to add data on which aggregation is applied (i.e., an aggregation argument $AggArg$) to database. Sliding `window` functionality is also simple, and it is realized by rule (6).

$$\begin{aligned}&\texttt{window}(WndwSize, OldCntr, OldVal, AggArg, NewVal) :- \\ &\quad OldCntr + 1 >= WindowSize - > \\ &\quad \texttt{retract}(LastItem), \\ &\quad \texttt{spec\_aggregate}(OldValue, AggArg, NewValue); \\ &\quad \texttt{spec\_aggregate}(OldValue, AggArg, NewValue).\end{aligned} \tag{6}$$

We check whether the current counter value (i.e., the incremented old counter, $OldCntr + 1$) exceeds the window size (line 2) in which case we retract the last item from the window (line 3) and compute a specific aggregate function (line 4). Recall that new data element (AggArg) was previously added by the iteration rule (`assert`($AggArg$)). If the counter does not exceed the window's value, we simply compute a specific aggregate function (line 5).

Based on these iterative pattern and sliding window rules we can implement other various aggregation functions. The iterative rules (7) (SUM aggregate function) implement the sum of certain values from selected events (see *SUM aggregate function*).

As we already explained, the iteration begins when `start_event` occurs and sets the $StartVal$. The iteration is further continued whenever event `a` occurs. Note that events `start_event` and `a` can be of the same type. We can additionally have WHERE clause to set filter conditions for both $StartVal$ and $AggArg$. We omit filters here to keep the pattern rules simple, however it is clear that neither every `start_event` must start the iteration nor that every `a` must be accepted in an ongoing iteration. The `assert` predicate adds new data ($AggArg$) to the current sum, and the window rule deducts the expired (last) value from the window in order to produce $NewSum$.

Note that the same rules can be used to compute the moving average (AVG) (hence we omit to repeat them to save space). As we have the current sum and the counter value, we can simply add $AvgVal = NewSum/(OldCntr+1)$ in the WHERE clause of the second rule.

$$\begin{aligned}&\texttt{sum}(StartCntr = 0, StartVal) \leftarrow \texttt{start\_event}(StartVal). \\ &\texttt{sum}(OldCntr + 1, NewSum) \leftarrow \\ &\quad \texttt{sum}(OldCntr + 1, OldSum) \text{ SEQ } \texttt{a}(AggArg) \\ &\quad \text{WHERE } \{\texttt{assert}(AggArg), \\ &\qquad \texttt{window}(WndwSize, OldCntr, \\ &\qquad OldSum + AggArg, AggArg, NewSum)\}.\end{aligned} \tag{7}$$

$$\begin{aligned}&\texttt{window}(WndwSize, OldCntr, CurrSum, NewSum) :- \\ &\quad OldCntr + 1 >= WindowSize - > \\ &\quad \texttt{retract}(LastItem), \\ &\quad NewSum = CurrSum - LastItem; \\ &\quad NewSum = CurrSum - LastItem.\end{aligned}$$

In general, the iterative rules give us possibility to realize essentially any aggregate functions on event streams, no matter whether events are *atomic* or *complex* (note that there is no assumption whether event `a` is atomic or complex). We can also have *multiple* aggregations, computed on a single iterative pattern (when they are supposed to be calculated on the same event stream). For instance, the same iterative rules can be used to compute the average and the standard deviation. This feature can potentially save computation resources and increase the overall performance. Finally, it is worth noting that we are not constrained to compute the Kleene plus closure only on *sequences* of events (as it is common in other approaches [1,10]). With no restriction, instead of SEQ we can also put (in line 3) other event operators such as AND or PAR . The following iterative pattern computes the *maximum* over a sliding window of events.

$$
\begin{array}{l}
\texttt{max}(StartCntr = 0, StartVal) \leftarrow \texttt{start\_event}(StartVal). \\
\texttt{max}(OldCntr + 1, NewMax) \leftarrow \\
\qquad \texttt{max}(OldCntr + 1, OldMax) \text{ SEQ } \texttt{a}(AggArg) \\
\qquad\ \text{WHERE } \{\texttt{assert}(AggArg), \\
\qquad\qquad\ \texttt{window}(WndwSize, OldCntr, NewMax)\}.
\end{array}
\tag{8}
$$

$$
\begin{array}{l}
\texttt{window}(WndwSize, OldCntr, NewMax) : - \\
\qquad OldCntr + 1 >= WindowSize- > \\
\qquad \texttt{retract}(LastItem), , get(NewMax); \\
\qquad get(NewMax).
\end{array}
$$

The rules are very similar to rules for other aggregation functions (e.g., see rules (8)). However there is one difference in implementation of the window rule. The history of events necessary for computing aggregations on sliding windows can be kept in the memory using different data structures. Essentially we need a *queue* where the latest event (or its aggregation value) is inserted into the queue and the oldest event from the window is removed. For example, we implemented efficiently the sum and the average using two data structures: *stacks* and *difference lists*. Stacks can be easy implemented in Prolog using *assert* and *retract* commands, and difference list are convenient as the cost for deleting the oldest element that expired from the window is O(1).

Queues with difference lists are however not good enough for computing aggregations such as the *maximum* and the *minimum*. For these functions, searching the maximum (or the minimum) in a sliding window when the current maximum (minimum) is deleted requires a price of O(Window) (to find the new maximum or the minimum). Still to provide an efficient implementation we use balanced binary search trees. We know what is the event that will be deleted from the history queue. We keep a red-black (RB) balanced tree to be indexed on the aggregate argument, so that we can do cleanup of overdue events efficiently. In each node, we keep a counter with how many times that an event with the aforementioned key came. At each time the maximum (minimum) is the rightmost (leftmost) leaf. Additionally we can also keep the timestamp of events. This allows us also to prune events (data) based on the *time* w.r.t the sliding window.

With the balanced tree this search is reduced to O(logN). For instance, for a window of 1000 events, the price of 1000 operations is reduced to at most 10 at each step ($2^{10} = 1024$).

Pruning events based on their timestamps is the basis for *time-based* sliding windows. So far we have discussed *count-based* sliding windows (i.e., the pruning is based on the number of events in the window). For event patterns with time-based sliding windows, we do not need the window rule (e.g., rule (6)). Instead, we use only iterative patterns with a garbage collector (set to prune events out of the specified sliding window). Events are stored internally in order as they come (we index them on the timestamp information $[T_2, T_1]$). This eases the process of pruning expired events, using either of our two memory management techniques.

$$\begin{array}{l} \texttt{iteration}(StartCntr = 0, StartVal) \leftarrow \texttt{start\_event}(StartVal). \\ \texttt{iteration}(NewCntr) \leftarrow \\ \qquad \texttt{iteration}(OldCntr) \text{ SEQ } \texttt{a}(AggArg) \\ \qquad \text{WHERE } \{NewCntr = \texttt{getCount}([T_2, T_1]), \texttt{window}(3min)\}. \end{array} \tag{9}$$

The *count* aggregation is typically used on time-based sliding windows, see the pattern (9). Whenever a relevant event occurs (e.g., event a), its timestamp will be asserted by the *getCount* predicate and the current counter number will be returned. Additionally we set a garbage collector to incrementally remove outdated timestamps, so that *getCount* always returns the correct result. In the same vein, we have realized other aggregate functions with the time-based sliding windows (i.e., SUM, AVG, MAX, MIN).

4 Performance Evaluation

We have implemented the proposed framework for iterative and aggregative patterns. In this section we present experimental results we have obtained with our open-source implementation, called ETALIS[8]. Experimental results compare our logic programming-based implementation with Esper 3.3.0[9]. Esper is a state-of-the-art engine primarily relying on NFA. We choose Esper as it is available as open source, and also it is a commercially proven system.

We have evaluated the *sum* aggregation function, defined by iterative pattern (7) (we omit rewriting the pattern here to save space). The moving sum is computed over the stream of complex events. Complex events are defined as a *conjunction* of two events, joined on their ID (see pattern rule (10)). The sum is aggregated on the attribute X of complex events $\texttt{a}(ID, X, Y)$. Figure 2(a) shows the performance results. In particular, the figure shows how the throughput depends on different sizes of the sliding window. Our system ETALIS was run in two modes: using the window implementation based on the stack and

[8] ETALIS, can be found on: `http://code.google.com/p/etalis/`
[9] Esper: `http://esper.codehaus.org`

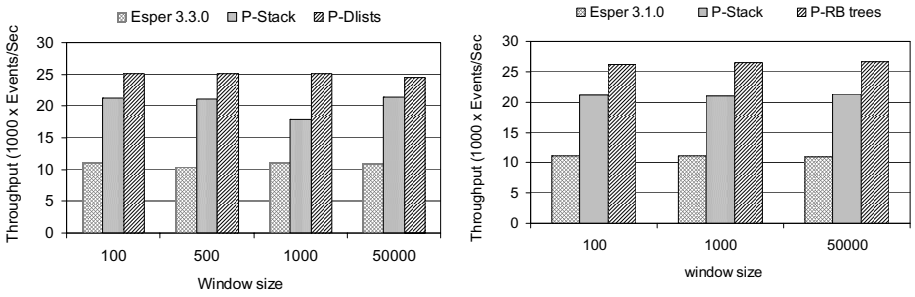

Fig. 2. (a) SUM-AND: throughput vs. window size (b) AVG-SEQ: throughput vs. window size

difference lists, denoted as P-Stack and P-Dlists, respectively. In both modes our implementation has outperformed Esper 3.3.0 (see Figure 2(a)).

$$\mathtt{a}(ID, X, Y) \leftarrow \mathtt{b}(ID, X) \text{ AND } \mathtt{c}(ID, Y). \tag{10}$$

In the next test we computed the moving *average* (avg) over the stream of complex events. Complex events were defined by rule (10) where operator AND was replaces with the *sequence* SEQ . Again ETALIS was run with windows implemented with the stack and different lists. Results are presented in Figure 2(b), showing again the dominance of our system.

Example application: supply chain. CEP can be combined with evaluation of the *background knowledge* to detect (near) real-time situations of interest. To demonstrate this functionality, let us consider the following example. Suppose we monitor a shipment delivery process in a supply chain system. The following rules represent a complex pattern (`delivery` event), triggered by every `shipment` event. This iterative pattern may be used to *aggregate* certain values carried by `shipment` events.

$$\begin{aligned} \mathtt{delivery}(start, start) &\leftarrow \mathtt{shipment}(start). \\ \mathtt{delivery}(From, To) &\leftarrow \mathtt{delivery}(From, PrevTo) \\ &\quad \text{SEQ } \mathtt{shipment}(To) \\ &\quad \text{WHERE } \mathtt{inSupChain}(From, To). \end{aligned} \tag{11}$$

Additionally there is a constraint that every shipment on its way needs to pass a number of sites, defined with a *delivery path*. Valid paths are represented as sets of explicit links between sites, e.g., with $\mathtt{linked}(site_3, site_4)$ we represent two connected sites. If for that shipment there exists also another connection $\mathtt{linked}(site_4, site_5)$, the system can *infer* that the path $site_3, site_4, site_5$ is a valid path (performing the *reasoning* over the following transitive closure and available background knowledge).

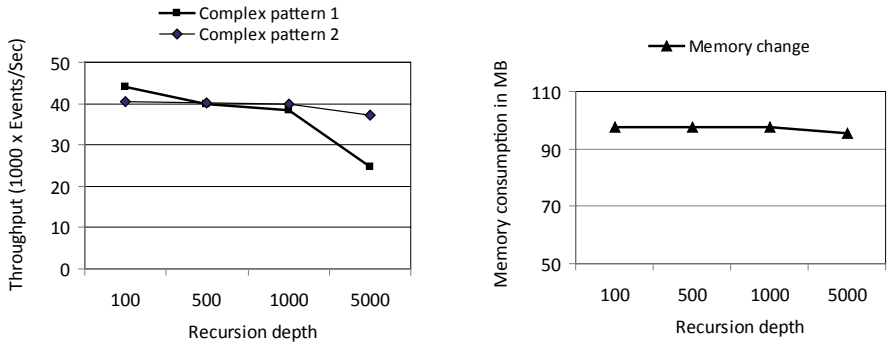

Fig. 3. (a) Throughput comparison (b) Memory consumption

$\texttt{inSupChain}(X, Y) :- \texttt{linked}(X, Y).$
$\texttt{inSupChain}(X, Z) :- \texttt{linked}(X, Y)$ AND $\texttt{inSupChain}(Y, Z).$

We have evaluated the iterative `delivery` pattern for different sizes of supply chain paths (between 100 and 5000 links), see Figure 3 (a). In "Complex pattern 1" we enforce that for each new `shipment` event, the valid path must be proved from its beginning (see $\texttt{inSupChain}(From, To)$ in rule (11)). For longer paths (e.g., 5000 links) this is a significant overhead, and we see that the throughput declines. But if we relax the check so that for every new event the path must be checked with respect only to the last `delivery` event, i.e., we replace $\texttt{inSupChain}(From, To)$ with $\texttt{inSupChain}(PrevTo, To)$ in rule (11)) we obtain the throughput which is almost constant (see "Complex pattern 2" in Figure 3 (a)). Figure 3 (b) shows the total memory consumption for the presented test. There is no difference in memory consumption for complex patterns 1 and 2, hence we present only one curve.

5 Conclusions

We have presented an extended formalism for logic-based event processing. The formalism is rather general, however in this paper we put emphasis on handling iterative and aggregative patterns matched against unbounded event streams. The paper presents syntax and declarative semantics of ETALIS Language for Events, demonstrates its use for more knowledge-oriented and intelligent event processing, provides an execution model, and finally shows performance evaluation of our prototype implementation.

Acknowledgments

This work was partially supported by the European Commission funded project PLAY (FP7-20495) and by the ExpresST project funded by the German Research Foundation (DFG). We thank Jia Ding and Ahmed Khalil Hafsi for their help in implementation and testing ETALIS.

References

1. Agrawal, J., Diao, Y., Gyllstrom, D., Immerman, N.: Efficient pattern matching over event streams. In: SIGMOD, pp. 147–160 (2008)
2. Allen, J.F.: Maintaining knowledge about temporal intervals. Communications of the ACM 26, 832–843 (1983)
3. Anicic, D., Fodor, P., Rudolph, S., Sthmer, R., Stojanovic, N., Studer, R.: Reasoning in Event-based Distributed Systems. In: Etalis: Rule-Based Reasoning in Event Processing. Series in Studies in Computational Intelligence, Sven Helmer, Alex Poulovassilis and Fatos Xhafa (2010)
4. Anicic, D., Fodor, P., Rudolph, S., Stühmer, R., Stojanovic, N., Studer, R.: A rule-based language for complex event processing and reasoning. In: Hitzler, P., Lukasiewicz, T. (eds.) RR 2010. LNCS, vol. 6333, pp. 42–57. Springer, Heidelberg (2010)
5. Arasu, A., Babu, S., Widom, J.: The cql continuous query language: semantic foundations and query execution. VLDB Journal 15, 121–142 (2006)
6. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.A.: Telegraphcq: Continuous dataflow processing for an uncertain world. In: Proceedings of the 1st Biennial Conference on Innovative Data Systems Research, CIDR 2003 (2003)
7. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Computing Surveys 33, 374–425 (2001)
8. Fodor, P., Anicic, D., Rudolph, S.: Results on out-of-order event processing. In: Rocha, R., Launchbury, J. (eds.) PADL 2011. LNCS, vol. 6539, pp. 220–234. Springer, Heidelberg (2011)
9. Krämer, J., Seeger, B.: Semantics and implementation of continuous sliding window queries over data streams. ACM Transactions on Database Systems 34 (2009)
10. Mei, Y., Madden, S.: Zstream: a cost-based query processor for adaptively detecting composite events. In: SIGMOD, pp. 193–206 (2009)