Retractable Complex Event Processing and Stream Reasoning

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 (CEP) deals with processing of continuously arriving events with the goal of identifying meaningful *patterns* (complex events). In existing stream database approaches, CEP is manly concerned by temporal relations between events. This paper advocates for a *knowledge-rich* CEP with Stream Reasoning capabilities. Secondly, we address the problem of *revision* in event processing. Events are often assumed to be immutable and therefore always correct. Revision in event processing deals with the circumstance that certain events may be revoked. This necessitates to reconsider complex events which might have been computed based on the original, flawy history as soon as part of that history is corrected.

In this paper, we present a novel approach for *knowledge-based CEP* and *Stream Reasoning*, including revisions of events too. We present a *rule-based language* for pattern matching over event streams with a precise syntax and the declarative semantics. We devise an execution model for the proposed formalism, and provide a prototype implementation. Extensive experiments have been conducted to demonstrate the efficiency and effectiveness of our approach.

1 Introduction

While existing semantic technologies and reasoning engines are constantly being improved in dealing with *time invariant* domain knowledge, they lack in support for processing *real-time* streaming data. Real-time data on Web is valuable only if it is captured, processed, and delivered instantly. Examples include traffic monitoring, real-time financial services, web click analysis and advertisement, various social web and real-time collaboration tools, and so forth.

Complex Event Processing (CEP) is a set of techniques and tools that help us in understanding and controlling real-time and event-driven systems [11]. As such, it is a technology that can help in processing real-time data on the Web too. CEP deals with processing continuously arriving events with the goal of identifying meaningful event patterns (complex events). An *event* represents something that occurs, happens, or changes the current state of affairs. For example, an event may represent a stock price change, a complied transaction, a new piece of information, knowledge made available by a Web service, and so forth. In all these situations, to structure the course of affairs and describe more complex *dynamic* situations, we compose simple (atomic) events into *complex* events. Today's CEP systems [1,13,4], however, focus on high throughput

N. Bassiliades et al. (Eds.): RuleML 2011 - Europe, LNCS 6826, pp. 122–137, 2011.

and timeliness as two important characteristics, while they do not meet the *complexity* requirements of event-driven applications. Pattern matching over streams poses two new challenges directly impacting the complexity of CEP systems:

Knowledge-based CEP & Stream Reasoning. According to [11], the *time critical* actions are supposed to be taken upon complex events. The question is, however, whether event patterns detectable by today's CEP systems are expressive enough to capture complex (business) events in all their aspects. How likely is that *critical* decisions are taken merely on event patterns of type, e.g., "event a is followed by event b in last 10 seconds"? For some applications such patterns are expressive enough; however, for *knowledge-rich* applications, they are certainly not. In such applications real-time actions are triggered not only by events, but also upon additional *knowledge*. This knowledge captures the *domain* of interest, or *context* related to business critical actions and decisions. Its purpose is to be evaluated during detection of complex events in order to *enrich* events with background information (context); or to detect more complex *situations*. The task of reasoning over streaming data (events) and the background knowledge constitutes a new challenge known as *Stream Reasoning* [15].

The Linked Open Data (LOD) initiative[1] has made available on the Web hundreds of datasets and ontologies. Examples also include the New York Times dataset[2], financial ontologies[3], encyclopedic data (e.g., DBpedia), Linked-GeoData[4], and so forth. This knowledge is commonly represented as *structured* data (e.g., using RDFS). Structured data enable machines to *reason* over explicit knowledge in order to infer new (implicit) information. However, current CEP systems [1,13,4] cannot utilize the structured knowledge, and they cannot do stream reasoning.

To achieve the aforementioned goal, various approaches have been proposed [6,5,10]. They are capable to process either additional background or structured knowledge (though varying in Complex Event Processing capabilities they provide). In this paper, we propose an approach that is capable to do both, Complex Event Processing and Stream Reasoning. Moreover, the goal of this paper is to provide an additional feature (in comparison to [6,5,10]), namely, *event revision*.

Non-blocking event revision. CEP systems such as [1,13,4] detect complex events based on reported atomic events. Once a complex event has been detected, typically there is no chance to *revise* this event later. Events are assumed to be immutable and therefore always correct. In practice, there is a number of reasons requiring *revisions* in event stream processing. For example, an event was reported by mistake, but did not happen in reality (and the mistake was realized later); an event happened, but it was not reported (due to failure of either a sensor, or failure of the event transmission system); or an event was triggered and later revoked due to the transaction failure. Also very often streaming data sources contend with noise (e.g., financial data feeds, Web streaming data, updates etc.) resulting in erroneous inputs and, therefore, erroneous complex event results. As recognised in [14], event stream sources may issue "revision tuples"

[1] such as e.g., `http://linkeddata.org/`
[2] Linked Open Data from the New York Times `http://data.nytimes.com/`
[3] Financial ontology: `http://www.fadyart.com/`
[4] LinkedGeoData: `http://linkedgeodata.org`

(revision events) that amend previously issued events. A CEP system should therefore be capable to take these revisions into account and produce correct revision outputs. There exist approaches for dealing with revision in event processing [9,12]). However, these approaches (as rooted in stream databases) cannot do Stream Reasoning.

The goal of this work is to provide a fundamental framework for processing event streams, exceeding the capabilities of today's CEP systems. We propose a formalism featuring an expressive *declarative* and *rule-based* semantics. As such, the formalism enables effective Complex Event Processing and Stream Reasoning. Apart from this, our approach naturally captures *revision* of acquired knowledge[5]. Extensive experiments have been conducted to demonstrate the practical efficiency and effectiveness of our approach.

2 Formal Model for Knowledge-Based Event Processing with Revision

We have defined a basic language for CEP in [3]. In this section, we extend the language to handle *retractions*. In order to keep the presentation of the overall formalism self-contained, we will also recall basics of the language from [3].

2.1 Event Processing Language Syntax

In this section, we present the formal syntax of our language for event processing, while in the remaining sections of the paper, we will gradually introduce other aspects of the language (i.e., the declarative and operational semantics as well as the performance of a prototypical implementation based on the language).

The syntax of our language allows for the description of *time* and *events*. We represent time instants as well as durations 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 described by ground atoms (i.e., predicates followed by arguments which are terms not containing variables). Intuitively, the arguments of a ground atom describing an atomic event denote information items (i.e., event data) that provide additional information about that event.

Atomic events can be composed to form *complex events* via *event patterns*. We use event patterns to describe how events can (or have to) be temporally situated relative to other events or absolute time points. The language P of event patterns is formally defined by

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

Thereby, pr a predicate name with arity n, 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. As a 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 following form:

[5] We focus on revision of *events*. Revision of background knowledge is out of scope.

$$\texttt{pr}(t_1, \ldots, t_n) \leftarrow p$$

where p is an event pattern containing every variable occurring in $\texttt{pr}(t_1, \ldots, t_n)$ at least once outside any function application.

Figure 1 demonstrates the various ways of constructing complex event descriptions from simpler ones in our language for event processing. Moreover, the figure informally introduces the semantics of the language, which will be formally defined in Section 2.3.

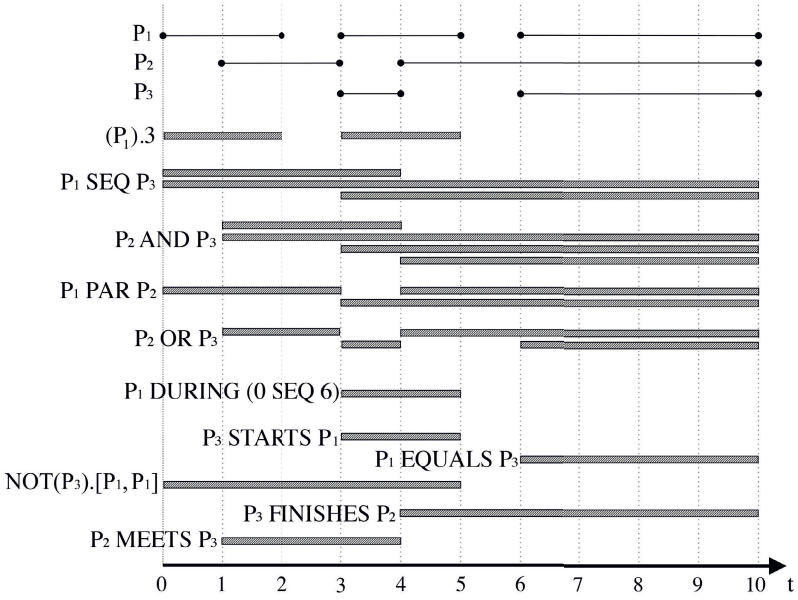

Fig. 1. Language for Event Processing - Composition Operators

It is worth noting that the language captures the set of all possible 13 relations on two temporal intervals as defined in [2] and can therefore be used for extensive temporal reasoning.

2.2 Examples

Let us briefly review the modeling capabilities of the presented pattern language.

General examples. One might be interested in defining an event matching stock market working days:

$$\texttt{workingDay}() \leftarrow \text{NOT}(\texttt{marketCloses}())[\texttt{marketOpens}(), \texttt{marketCloses}()].$$

Moreover, we might be interested in detecting the event of two bankruptcies happening on the same market working day:

$$\texttt{dieTogether}(X, Y) \leftarrow \big(\texttt{bankrupt}(X) \,\text{SEQ}\, \texttt{bankrupt}(Y)\big) \,\text{DURING}\, \texttt{workingDay}().$$

This event rule also shows how event information (about involved institutions, provenance, etc.) can be "passed" on to the defined complex events by using variables. Furthermore, variables may be employed to conditionally group events into complex ones if they refer to the same entity:

$$\texttt{indirectlyAcquires}(X, Y) \leftarrow \texttt{buys}(Z, Y) \text{ AND } \texttt{buys}(X, Z)$$

Knowledge-based patterns. Let us consider an example demonstrating *knowledge-based* pattern detection. Suppose that we want to detect the stock price increase in a supply chain system of companies. The following pattern monitors two stock price increases in two companies (occurred within certain time window), and checks whether the companies are parts of the supply chain system.

$$\texttt{trendIncrease}() \leftarrow \big(\texttt{stockIcr}(CompanyA) \text{ SEQ } \texttt{stockIcr}(CompanyB)\big).10 \\ \text{AND } \texttt{inSupChain}(CompanyA, CompanyB).$$

The supply chain system is represented as a set of explicit links between companies, e.g., with $\texttt{linked}(A, B)$ we represent two interconnected businesses involved in the ultimate provision of a product. We assume that such explicit relationships are continuously being updated via *information events* as, for instance, our data mining tool processes different information sources, delivering events of the form:

$$\texttt{linked}(CompanyA, CompanyB)$$
$$\ldots$$
$$\texttt{linked}(CompanyY, CompanyZ)$$

The above set of `linked` relations can be represented, with no restriction, as a set of RDF triples too. Our prototype implementation (see Section 4) uses Semantic Web Library[6] to represent an RDFS ontology as a set of Prolog rules and facts.

The following transitive closure pattern can then be used to span over semantic relationships between companies scenario where direct supply relationships are represented explicitly, and hence discover implicit relationships, i.e., whether two stock price increases also covered the whole supply chain system.

$$\texttt{inSupChain}(X, Y) \leftarrow \texttt{linked}(X, Y).$$
$$\texttt{inSupChain}(X, Z) \leftarrow \texttt{linked}(X, Y) \text{ AND } \texttt{inSupChain}(Y, Z).$$

To generalize, for a given set of events that satisfy certain *temporal* relationships, our approach may be used to additionally check whether these events satisfy certain *semantic* relationships with respect to domain knowledge that itself may be dynamically collected. Semantic relationships between occurring events is an important dimension, neglected in today's CEP systems. It helps discovering the *context* in which events occurred by combining *knowledge management* techniques (e.g., *deductive reasoning*) with event stream processing.

Event revision. To illustrate how *event revision* can be useful in practise, let us consider the following example. An automated stock brokerage system sells stocks to its clients. The system emits an event described by `availableStock` to a client every time

[6] SWI-Prolog: `http://www.swi-prolog.org/pldoc/package/semweb.html`

when the respective stocks become available. The event contains information about the company's stock ID, the current price Pri, and the available amount of stocks Amt. A client (identified by CID) may now signal the request to buy the offered stocks by sending an event `trChecked` back to the system, stating the wanted amount Amt_1 of stocks. Event `availableStock` followed by event `trChecked` will trigger a complex event `buyStocks` according to the following rule:

$$\texttt{buyStocks}(CID, ID, Pri, Amt_1) \leftarrow \texttt{availableStock}(ID, Pri, Amt)$$
$$\text{SEQ } \texttt{trChecked}(CID, ID, Pri, Amt_1) \text{ WHERE } Amt_1 \leq Amt.$$

Upon detection, event `buyStocks` will trigger two transactions: the first transaction transfers money from the client's account to the broker's account, the second transaction maintains the balance of available stocks, by subtracting Amt_1 from Amt. The maintenance is necessary as available stocks are also offered to other interested clients. Since the stock trading is carried out in real-time, it is important that execution in the stock brokerage system is automated and that the transaction of one client does not block executions of other clients (as long as $Amt > 0$). Now, suppose that event `balanceChange` is triggered whenever the balance of available stocks changes from Amt_2 to Amt_3 by customer identified as CID (i.e., whenever the second transaction completes). For example, these events may be used for transaction execution monitoring, statistical analysis, etc. Let us furthermore assume that the following pattern is used to detect stock trades of suspiciously large volume, which may hint at a potential fraud.

$$\texttt{bigTrade}(CID, ID, Amt_1) \leftarrow \texttt{buyStocks}(CID, ID, Pri, Amt_1)$$
$$\text{SEQ } \texttt{balanceChange}(CID, Amt_2, Amt_3) \text{ WHERE } (Amt_2 - Amt_3) > 10000.$$

Many transactions concurrently change the balance, and after each change, event `balanceChange` is triggered. Now let us suppose that an event `bigTrade` has been detected, and a fraud investigation was initiated. Just a second afterwards, the money transfer transaction fails (due to insufficient account balance of a customer). In this situation, the amount of available stocks will be restored by executing a compensation transaction. Moreover, the corresponding `balanceChange` event needs to be retracted. Finally, the `bigTrade` complex event needs to be revoked too, leading to the cancelation of the fraud investigation.

The automated stock brokerage system operates with flexible policies, allowing customers to cancel their transaction within certain time. If after detection of event `bigTrade`, a customer cancels her transaction (by *retracting* event `trChecked`) the atomic event `buyStocks` will be revoked too, which in turn will necessitate the retraction of event `bigTrade`.

2.3 Declarative Semantics

We define the declarative formal semantics of our language for event processing 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 $f^* : Con^n \rightarrow Con$. That is, in our setting, functions are treated as built-in utilities.

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

$$\mu^* : \begin{cases} v \mapsto \mu(v) & \text{if } v \in Var, \\ c \mapsto c & \text{if } c \in Con, \\ f(t_1, \ldots, t_n) \mapsto f^*(\mu^*(t_1), \ldots, \mu^*(t_n)) & \text{otherwise.} \end{cases}$$

In addition to the set of rules $\mathcal{R}$, we define an *event stream* $\mathcal{S} = (\mathbb{E}, \mathsf{ev}, \mathsf{occ}, \mathsf{rev})$. Thereby, $\mathbb{E}$ is a set of events, $\mathsf{ev} : \mathbb{E} \rightarrow Ground$ a function assigning a ground atom (specifying the event type and possibly additional information) to every event and $\mathsf{occ}, \mathsf{rev} : \mathbb{E} \nrightarrow \mathbb{Q}^+$ partial functions assigning to events time points at which they occur or are revoked, respectively. As a side condition, we presume that for all $e \in \mathbb{E}$ with $\mathsf{rev}(e)$ defined, $\mathsf{occ}(e)$ is defined as well and $\mathsf{occ}(e) < \mathsf{rev}(e)$, i.e., an event can only be revoked after it has occurred. Moreover, we require the event stream to be free of accumulation points, i.e., for every $q \in \mathbb{Q}^+$, the set $\{q' \in \mathbb{Q}^+ \mid q' < q \text{ and } q' = \mathsf{occ}(e) \text{ for some } e \in \mathbb{E}\}$ is finite.

Given an event stream $\mathcal{S} = (\mathbb{E}, \mathsf{ev}, \mathsf{occ}, \mathsf{rev})$ and a time "viewpoint" $v \in \mathbb{Q}^+$, we now define the auxiliary function $\epsilon_v : Ground \rightarrow 2^{\mathbb{Q}^+}$ from ground atoms into sets of nonnegative rational numbers by

$$\epsilon_v(at) := \mathsf{occ}\big(\mathsf{ev}^{-1}(at) \cap \big(\mathsf{occ}^{-1}([0, v]) \setminus \mathsf{rev}^{-1}([0, v])\big)\big)$$

It thereby indicates at what time instants what event types occur according to all the (occurrence and revocation) information obtained up to the time viewpoint v.

Now, we define an interpretation $\mathcal{I} : Ground \rightarrow 2^{\mathbb{Q}^+ \times \mathbb{Q}^+}$ as a mapping from the 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 $g \in Ground$.

Given an event stream $\mathcal{S}$ and a viewpoint $v \in \mathbb{Q}^+$, we call an interpretation $\mathcal{I}$ *model* for a rule set $\mathcal{R}$ – written as $\mathcal{I} \models^v_{\mathcal{S}} \mathcal{R}$ – if the following conditions are satisfied:

C1 $\langle q, q \rangle \in \mathcal{I}(g)$ for every $g \in Ground$ and $q \in \epsilon_v(g)$.
C2 for every rule $atom \leftarrow pattern$ and every variable assignment μ we have $\mathcal{I}_\mu(atom) \subseteq \mathcal{I}_\mu(pattern)$ where $\mathcal{I}_\mu$ is inductively defined as displayed in Figure 2.

Given an interpretation $\mathcal{I}$ and some $q \in \mathbb{Q}^+$, we let $\mathcal{I}|_q$ denote the interpretation defined via $\mathcal{I}|_q(g) = \mathcal{I}(g) \cap \{\langle q_1, q_2 \rangle \mid q_2 - q_1 \leq q\}$.

Given two interpretations $\mathcal{I}$ and $\mathcal{J}$, we say that $\mathcal{I}$ is *preferred* to $\mathcal{J}$ if there exists a $q \in \mathbb{Q}^+$ with $\mathcal{I}|_q \subset \mathcal{J}|_q$.

A model $\mathcal{I}$ is called *minimal* if there is no other model preferred to $\mathcal{I}$. It is easy to show that for every event stream $\mathcal{S}$, viewpoint $v \in \mathbb{Q}^+$, and rule base $\mathcal{R}$ there is a unique minimal model $\mathcal{I}^{\mathcal{S},v,\mathcal{R}}$.

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 $\mathcal{R}$ at the viewpoint v (written $\mathcal{S}, v, \mathcal{R} \models a^{[q_1,q_2]}$), if $\langle q_1, q_2 \rangle \in \mathcal{I}^{\mathcal{S},v,\mathcal{R}}_\mu(a)$ for some variable assignment μ.

Clearly, the problem of deciding $\mathcal{S}, v, \mathcal{R} \models a^{[q_1,q_2]}$ is time polynomial with respect to the combined size of $\mathcal{R}$ and $\mathcal{S}$, given bounded arity of the used predicates and polynomial computation time for the built-in functions. This result is a straightforward consequence from the fact that there only polynomially many $a^{[q_1,q_2]}$ to be considered

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 $\langle q_3, q_4\rangle \in \mathcal{I}_\mu(p_2)$ for some $q_2, q_3 \in \mathbb{Q}^+$ with $q_2 < q_3\}$
p_1 AND p_2	$\{\langle \min(q_1, q_3), \max(q_2, q_4)\rangle \mid \langle q_1, q_2\rangle \in \mathcal{I}_\mu(p_1)$ and $\langle q_3, q_4\rangle \in \mathcal{I}_\mu(p_2)$ for some $q_2, q_3 \in \mathbb{Q}^+\}$
p_1 PAR p_2	$\{\langle \min(q_1, q_3), \max(q_2, q_4)\rangle \mid \langle q_1, q_2\rangle \in \mathcal{I}_\mu(p_1)$ and $\langle q_3, q_4\rangle \in \mathcal{I}_\mu(p_2)$ for some $q_2, q_3 \in \mathbb{Q}^+$ with $\max(q_1, q_3) < \min(q_2, q_4)\}$
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 $\langle q_2, q_3\rangle \in \mathcal{I}_\mu(p_2)$ for some $q_2 \in \mathbb{Q}^+\}$
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 $\langle q_3, q_4\rangle \in \mathcal{I}_\mu(p_2)$ for some $q_2, q_3 \in \mathbb{Q}^+$ with $q_3 < q_1 < q_2 < q_4\}$
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 $\langle q_1, q_3\rangle \in \mathcal{I}_\mu(p_2)$ for some $q_2 \in \mathbb{Q}^+$ with $q_2 < q_3\}$
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 $\langle q_1, q_3\rangle \in \mathcal{I}_\mu(p_2)$ for some $q_2 \in \mathbb{Q}^+$ with $q_1 < q_2\}$
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. 2. Definition of extensional interpretation of event patterns. We use $p_{(x)}$ for patterns, $q_{(x)}$ for rational numbers, $t_{(x)}$ for terms, and pr for predicates .

and their validity can be computed in a bottom-up way with increasing interval length. The computational overhead introduced by event revision is not measurable in terms of worst-case complexity which is PTime with and without the revision component.

In the sequel, we will see how this declarative, time-dependent semantics is realized incrementally, as v proceeds, i.e., the "computed semantics" at some time viewpoint v is revised to obtain the semantics at some latter stage, instead of computing everything from scratch.

3 A Rule-Based Execution Model

This section starts with a brief explanation on how complex events can be computed with *event-driven backward chaining* (EDBC) rules [3]. This is our basic mechanism for derivation of complex events in a data-driven fashion (with logic rules). Later on, we extend the mechanism to handle event revision too.

Sequence with event revision. Let us consider a sequence of events represented as a rule: e ⟵ a SEQ b SEQ c. Event e is detected when event a[7] is followed by b and in turn followed by event c. We can always represent the above pattern as $e \leftarrow ((a \text{ SEQ } b) \text{ SEQ } c)$.

We refer to this way of "coupling events" as *binarization* of events. Effectively, in binarization we introduce *two-input* intermediate events (goals). For example this allows us to rewrite the above sequence as $ie_1 \leftarrow a \text{ SEQ } b$, and $e \leftarrow ie_1 \text{ SEQ } c$. Every monitored event (either atomic or complex), including intermediate events, will be assigned with one or more *logic rules*, fired whenever that event occurs.

In the following, we give more details about assigning rules to each monitored event. Algorithm 1 accepts as input a rule referring to a binary sequence $e_i \leftarrow a \text{ SEQ } b$, and produces executable rules for the sequence pattern. A detected sequence can also be *retracted* by the given transformation. If this occurs, the retraction is further propagated amongst other patterns (built upon that sequence).

[7] More precisely, by "event a" is meant an *instance* of the event of type a.

Algorithm 1. Sequence

Output: event-driven backward chaining rules for SEQ operator including revision.
Each event binary goal $\texttt{ie}_1 \leftarrow \texttt{a}$ SEQ $\texttt{b}$ is converted into: {

$\texttt{a}(ID,[T_1,T_2]) :- \texttt{for\_each}(\texttt{a},1,ID,[T_1,T_2]).$
$\texttt{a}(1,ID,[T_1,T_2]) :- \texttt{assert}(\texttt{goal}(\texttt{b}(\_,[\_,\_]),\texttt{a}(ID,[T_1,T_2]),\texttt{ie}_1(\_,[\_,\_]))).$
$\texttt{rev\_a}(ID,[T_3,T_4]) :- \texttt{for\_each}(\texttt{rev\_a},1,ID,[T_3,T_4]).$
$\texttt{rev\_a}(1,ID,[T_3,T_4]) :- \texttt{goal}(\texttt{b}(\_,[\_,\_]),\texttt{a}(ID,[T_1,T_2]),$
$\texttt{ie}_1(\_,[\_,\_])),\texttt{retract}(\texttt{goal}(\texttt{b}(\_,[\_,\_]),\texttt{a}(ID,[T_1,T_2]))).$
$\texttt{rev\_a}(2,ID,[T_3,T_4]) :- (\texttt{ie\_1}(ID,[T_1,T_2]),$
$\texttt{retract}(\texttt{ie\_1}(ID,[T_1,T_2])),\texttt{rev\_ie}_1(ID,[T_1,T_2]));\texttt{true}.$
$\texttt{b}(ID,[T_3,T_4]) :- \texttt{for\_each}(\texttt{b},1,ID,[T_3,T_4]).$
$\texttt{b}(1,ID,[T_3,T_4]) :- \texttt{goal}(\texttt{b}(\_,[\_,\_]),\texttt{a}(ID,[T_1,T_2]),$
$\texttt{ie}_1(\_,[\_,\_])),T_2 < T_3,\texttt{ie}_1(ID,[T_1,T_4]).$
$\texttt{rev\_b}(ID,[T_5,T_6]) :- \texttt{for\_each}(\texttt{rev\_a},1,ID,[T_5,T_6]).$
$\texttt{rev\_b}(1,ID,[T_5,T_6]) :- (\texttt{ie}_1(ID,[T_1,T_4]),$
$\texttt{retract}(\texttt{ie}_1(ID,[T_1,T_4])),\texttt{rev\_ie}_1(ID,[T_1,T_4]));\texttt{true}.$
$\texttt{ie\_1}(ID,[T_1,T_4]) :- \texttt{for\_each}(\texttt{ie\_1},1,ID,[T_1,T_4]).$
$\texttt{ie\_1}(1,ID,[T_1,T_4]) :- \texttt{assert}(\texttt{ie\_1}(ID,[T_1,T_4])).$}

}

The binarization step must precede the rule transformation. We first consider rules that handle sequence without event revision. These rules in Algorithm 1 do not have prefix rev_event_name (e.g., $rev\_a(1,ID,[T_3,T_4])$), and belong to one of two different classes of rules[8]. We refer to the first class as to *goal inserting rules*. The second class corresponds to *checking rules*. For example, the second rule in Algorithm 1 (i.e., with $\texttt{a}(1,ID,[T_1,T_2])$ in the rule head) belongs to the first class of rules, as it inserts $goal(b(\_,\_),a(T_1,T_2),ie_1(\_,\_))$. This rule will fire when an event of type a occurs, and the meaning of the inserted goal is as follows: "an event a has occurred at $[T_1,T_2]$,[9] and we are waiting for event b to happen in order to detect event $\texttt{ie}_1$." Obviously, the goal does not carry information about times for b and $\texttt{ie}_1$, as we don't know when they will occur. In general, the *second* event in a goal always denotes the event that has just occurred. The role of the *first* argument is to specify what we are waiting for, to detect an event that is on the *third* position.

The rule with $\texttt{b}(1,ID,[T_3,T_4])$ in the rule head (see Algorithm 1) belongs to the second class (i.e., *checking rule*). This rule checks whether certain prerequisite goals already have been asserted, in which case it triggers the more complex event. In this example, the rule will fire whenever event b occurs. The rule checks whether $goal(\texttt{b}(\_,[\_,\_]),\texttt{a}(ID,[T_1,T_2]),\texttt{ie}_1(\_,[\_,\_]))$ already exists (i.e., a has previously happened), in which case the rule triggers $\texttt{ie}_1$, by calling $\texttt{ie}_1(ID,[T_1,T_4])$. After detection of event $\texttt{ie}_1$, $goal(\texttt{b}(\_,[\_,\_]),\texttt{a}(ID,[T_1,T_2]),\texttt{ie}_1(\_,[\_,\_]))$ could be removed from

[8] There exist the third class of rules too (with for_each predicate). However these auxiliary rules, implementing a sort of "for each" loop, and ensuring that whenever an event of certain type happens, all rules with that event in the head fire.

[9] Apart from the timestamp, an event may carry other data parameters. They are omitted here for the sake of readability.

the database to free up memory (as it is "consumed"). However this is not the case, as the goal still may be useful if the revision of event a takes place (see below the case when event rev_a happens).

The time occurrence of $\texttt{ie}_1$ (i.e., $[T_1, T_4]$) is defined based on the occurrence of constituting events (i.e., $\texttt{a}(ID, [T_1, T_2])$, and $\texttt{b}(ID, [T_3, T_4])$, see Section 2.3). By calling $\texttt{ie}_1(ID, [T_1, T_4])$, this event will be inserted as a fact (see Algorithm 1). If later on, the revision process takes place, this fact will serve as a proof that event $\texttt{ie}_1$ occurred and hence may be retracted. If event $\texttt{ie}_1$ is further used in composition of other complex events, there will exist another rule with $\texttt{ie}_1$ in the rule head (apart from the current rules). The purpose of those rules would be to propagate the occurrence of event $\texttt{ie}_1$ upward (since it is an intermediate event).

Let us now explain how Algorithm 1 handles *event revision* in a sequence of two events. If once detected, event $\texttt{ie}_1$ may be retracted by an occurrence of either event rev_a or rev_b. That is why there are two sets of *revision* rules: rev_a and rev_b, see Algorithm 1. Additionally, events rev_a and rev_b may retract other detected events, if they were used in their detections and their IDs match. The identification (ID) is used to make a distinction between possible retractions of instances of the same event types.

If an event rev_a happens, rules $\texttt{rev\_a}(1, ID, [T_3, T_4])$ and $\texttt{rev\_a}(2, ID, [T_3, T_4])$ aim to nullify a prior occurrence of an event a. In particular, if an event a has happened, a goal $goal(\texttt{b}(\_, [\_, \_]), \texttt{a}(ID, [T_1, T_2]), \texttt{ie}_1(\_, [\_, \_]))$ will be inserted into the database. Therefore the subsequent occurrence of rev_a needs to delete that goal. The rule $\texttt{rev\_a}(1, ID, [T_3, T_4])$ does that. If the following sequence of events occurs: $\texttt{a}, \texttt{rev\_a}, \texttt{b}$, then event $\texttt{ie}_1$ will not be detected (as rev_a has nullified the occurrence of a). If event rev_a happens after event b, event $\texttt{ie}_1$ will need to be retracted (as it has already been detected). The rule with $\texttt{rev\_a}(2, ID, [T_3, T_4])$ in the head is used in the latter scenario.

In the previously described algorithm, we assumed that all events in a binarized pattern have the same ID (i.e., $\texttt{ie}_1(ID) \leftarrow \texttt{a}(ID)$ BIN $\texttt{b}(ID)$). It is worth noting that some intermediate or complex events may be composed of events with different IDs. In such cases, an additional ID may be added, e.g., $\texttt{ie}_1(ID_1, ID_2)$. ID_1 will then denote an ID of the left-hand-side event ($\texttt{a}(ID_1)$), and ID_2 will denote an ID of the right-hand-side event ($\texttt{b}(ID_2)$). Checking these IDs when certain events are retracted allows to employ event revision using the presented algorithms with no further restriction.

Rules produced by the transformation in Algorithm 1 are executable rules (Prolog rules). With no restriction these rules may be accompanied by other Prolog rules, used for example to express the background or domain *knowledge* (see Example "Knowledge-based patterns" from Section 2.2). To also enable use of existing online knowledge bases expressed as RDFS ontologies (e.g., from LOD initiative and other sources, see Section 1), we use existing tools for conversion of RDFS to Prolog, such as SWI-Prolog Semantic Web Library. This conversion is done at design-time, and has no impact at run-time characteristics of our framework.

Rule transformations for other language constructs – defined in Section 2 – are omitted for space reasons.

4 Experimental Results

As a proof of concept, we have provided a prototype implementation of the presented framework for knowledge-based CEP with event revision capabilities. The implementation is part of our open-source engine for event processing called ETALIS[10]. Since our approach is based on deductive rules, it was convenient to provide the implementation in Prolog.[11] The prototype automatically compiles the user-defined complex patterns, written in the presented language (see Section 2) into Prolog rules. Also, our engine can automatically load an accompanied RDFS ontology (as a domain background knowledge base) into Prolog. YAP Prolog version 5.1.3[12] is then used to execute the compiled rules. All tests were carried out on a workstation with Intel Core Quad CPU Q9400 2,66GHz, 8GB of RAM, running Windows Vista x64. To run tests on streaming data, we have implemented an event stream generator that creates time series data with probabilistic events. We present a test with real data set too.

Knowledge-based CEP test. As a concrete example, we show the evaluation of the $trendIncrease$ complex pattern from Section 2.2. We varied the pool of companies in the transitive closure, ranging from 100 to 100,000 $linked$ companies. Figure 3(a) shows the throughput in thousands of events/second, obtained after detection of $stockIcr$ events. To prove the supply-chain connectivity between two companies, the system needs to evaluate transitive closure rules, i.e., it needs to perform Stream Reasoning (see $inSupChain$ rules from Section 2.2). It can be seen that the computation of the recursive relation $inSupChain$ has a relatively small effect, $\sim$10%, on the overall complex processing execution time (even when the system needs to traverse 100,000 $links$ in between two $stockIcr$ events). Our system detects more 20000 complex events per second, where for each complex event, the system additionally needed to process background knowledge consisting of 100000 facts (or RDF triples).

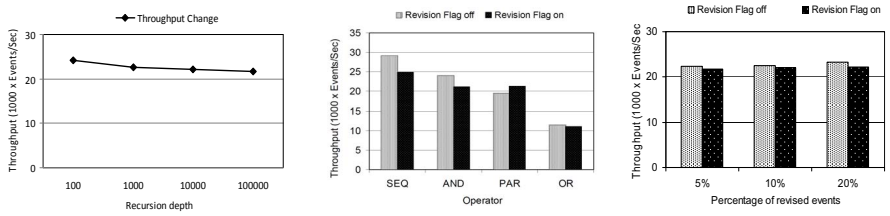

Fig. 3. (a) CEP with Stream Reasoning (b) Throughput comparison (c) Negation and revision

Event revision CEP experiments. Figure 3(b) shows experimental results we obtained for an event pattern represented by rule (1). In particular, Figure 3(b) shows the throughput comparison with and without handling event revision. We did the measurement for a pattern that exhibits different event operators (i.e., BIN instantiated by SEQ , AND , OR) of two events and the join operation on their ID attribute. The y-axis shows the

[10] ETALIS: `http://code.google.com/p/etalis/`
[11] With similar effort, our revision model could be implemented in other rule languages too.
[12] YAP Prolog: `http://www.dcc.fc.up.pt/~vsc/Yap/`

event throughput achieved by our prototype when events are, and are not, retractable. The x-axis shows different event operators in rule (1). The performance loss when revision is handled is moderate, and it happens mainly due to the fact that more events (goals) are kept in memory; hence more data needs to be indexed and processed.

$$\mathtt{e}(ID) \leftarrow \mathtt{a}(ID) \text{ BIN } \mathtt{b}(ID). \tag{1}$$

We also present an in-comparison throughput for negation. The tested pattern with negation is depicted by rule (2). The pattern detects an event `a` followed by an event `b`, with no occurrence of an event `c` in between (provided that all event instances must have the same ID). Figure 3(c) shows evaluation results for this pattern. We compare two throughputs, one obtained by processing streams without retracted events; and another with retracted events. The percentage of negated events (i.e., those of type `c`) in both streams varies from 5% to 20%. Additionally, streams with retracted events contain negated events with the same percentage (i.e., from 5% to 20%). The achieved results are similar to those from other operators.

$$\mathtt{e}(ID) \leftarrow \text{NOT}(\mathtt{c}(ID)).[\mathtt{a}(ID) \text{ SEQ } \mathtt{b}(ID)]. \tag{2}$$

We have also tested the *latency* caused by retraction of a hierarchy of complex events (i.e., not only complex events detected directly from an input stream). Complex events in this tests are chained events, as represented by rule (3). That is, when event $\mathtt{e}_1$ occurs, it will trigger other n events in a chain. Also if event $\mathtt{e}_1$ is retracted, all n chained events will be retracted. We have created event chains of different sizes, ranging from 1000 events to 50000 events. Once the chains are created, we retract the first event in the chain and measure the time required to retract all other triggered events. Figure 4 shows the experiment results. Retraction of 1000 event is done in 31 ms and all up to 10000 events the delay seems fairly negligible (less than a second). However to retract 20000 and specially for 50000 events, the time increases exponentially (i.e., approx. 3 s and 16 s). Note that this test is rather hard as we assumed that all 50000 events have the same ID, so no goal could have been removed while computing and retracting all of them. Obviously, this fact has its consequences on the performance.

$$\begin{array}{l} \mathtt{e}_2(ID) \leftarrow \mathtt{e}_1(ID). \\ \mathtt{e}_3(ID) \leftarrow \mathtt{e}_2(ID). \\ \ldots \\ \mathtt{e}_{n+1}(ID) \leftarrow \mathtt{e}_n(ID). \end{array} \tag{3}$$

All presented tests so far were carried out with probabilistic synthetic data streams. We could not find available real data sets with revision tuples (as they are usually kept proprietarily). Still to present a more realistic scenario, we took a stream of IBM stocks from 1962 year up to now, provided by Yahoo Finance[13]. We artificially inserted 5% of revision tuples to this stream. Format of events provided by Yahoo Finance is $\mathtt{stock}(ID, Date, Opn, High, Low, Cls, Vol, Adj)$ where ID is a company ID; $Date$ is a current date; $Opn, High, Low, Cls$ denote the opening, the highest, the lowest, and closing price, respectively; Adj is the closing price adjusted for dividends and splits.

[13] Yahoo Finance: `http://finance.yahoo.com/`

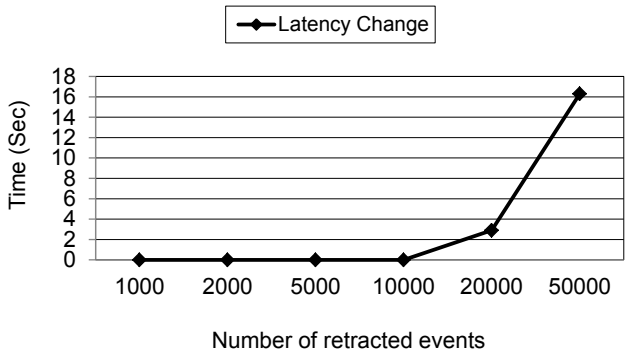

Fig. 4. Event latency

The event pattern is represented by rule (4). We monitored the price increase of two successive stock updates w.r.t Adj data. Additionally a filter for the price increase was specified by X, where X varied between 0% and 10%. Figure 5(b) compare results obtained for the original stream and the one modified with revision tuples.

$$\begin{aligned} &\texttt{stockIncr}(ID, Adj_1, Adj_2) \leftarrow \\ &\quad \texttt{stock}(ID, Date_1, Opn_1, High_1, Low_1, Cls_1, Vol_1, Adj_1) \\ &\quad \text{SEQ} \\ &\quad \texttt{stock}(ID, Date_2, Opn_2, High_2, Low_2, Cls_2, Vol_2, Adj_2) \\ &\quad \text{WHERE } (Adj_1 * X < Adj_2). \end{aligned} \tag{4}$$

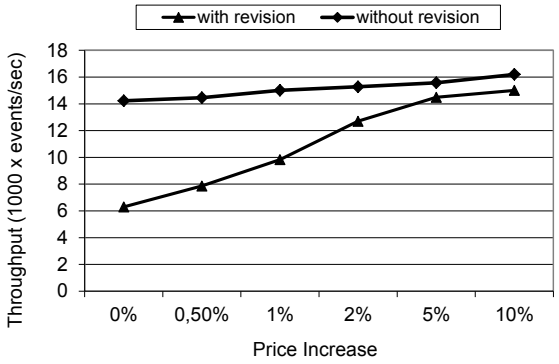

Fig. 5. Stock price change on a real data set

First, we see that the throughput without revision is lower than the one obtained from a similar test (see Figure 3(b)). Our closer investigation has shown that this difference was not caused by the use of real data set. Instead it has to do with more efficient indexing in the former test (Figure 3(b)). Note that in the real stream, all events are of the same type (i.e., $stock$) whereas in the synthetic data set we have two types (i.e.,

a and b). Our engine is more effective when events are discriminated upon their types (rather than on data attributes, e.g., an ID). Second, we can observe that the throughput without revision slightly increases as the filter condition gets tighter. This result is understandable, since in this case, less complex events, are computed and the throughput (based on the input stream) raises up.

At the end, it is worth mentioning that costs of compilation of an event program (written in the formalism, proposed in Section 2) into Prolog rules are minor. Typically, a program is compiled in few micro seconds, and the compilation is done only once at the design-time. Hence, the compilation does not cause a significant overhead.

5 Related Work

Work related to ours goes in two directions. The first direction reviews existing approaches from Data Stream Management Systems (DSMS) that also handle event revision (retraction). The other discusses Knowledge-based CEP & Stream Reasoning. We are not aware of any approach covering both aspects.

DSMS approaches. The Borealis CEP engine [9] features a mechanism for revision processing. The mechanism handles erroneous input events by generating corrections of previously output query results on data streams.

This work has been extended in [12] by proposing a revision model based on "replay" of event history. The technique assumes that a stream engine maintains an archive of recent data seen on each of its input streams. These archives are revised when revision tuples occur, and reprocessing (replaying) the sequence of input tuples than generates any of the query results invalidated by the revision.

While this technique is general and works well for all classes of patterns supported by Borealis system [9], it requires the event history to be kept (persisted). The history is kept as long as revision needs to be guaranteed. In our approach we also need to keep extra data in order to enable revision. However we saw (in Section 3) that we do not need to keep the *whole* event history (i.e., during the period of time in which revision is guaranteed). We keep only intermediate results (goals) relevant w.r.t detected complex events. Moreover we do not need to *replay* the whole history when computing revisions. The intermediate results (goals) represent partial results, hence they enable us to obtain revisions without re-computing them from scratch.

In [7] revision is considered as a problem caused by out of order events, i.e., it is possible to revise the occurrence time as well as the time when an event is reported to the system. We consider a general case where not only times can be revised, but the whole event can be retracted. Moreover, the consequences of that retraction are amended not only on detected patterns but also on complex patterns that are built out of them (i.e., hierarchies of complex events). The work in [7] is based on *buffering* and synchronization points. An input stream may be *blocked* in between synchronization points until events are reordered. On the other hand, we propose an approach that never blocks the input events. Further on, we never buffer the input stream and reorder it.

Knowledge-based CEP & Stream Reasoning. Continuous SPARQL (C-SPARQL) [5] is a language for continuous query processing over streams of RDF data. It extends

the SPARQL language by adding support for window and aggregation operations. The work in [8] introduces Streaming SPARQL. The approach is built on temporal relational algebra, and the authors provide an algorithm to transform SPARQL queries to that algebra. As in [5], the approach is lacking event processing capabilities, i.e., detecting RDF triple sequences occurring in a specific time relatedness.

Finally, in [10] an approach for integrating sensor streams with LOD background knowledge has been presented. As a part of this work, a continuous query language, CQELS, has been proposed. The language supports sliding windows, aggregations, and other operators supported by SPARQL language (which are now adapted to stream processing).

Our work is similar to this and other, previously mentioned, approaches. We, however, follow completely a deductive rule-based paradigm, providing an effective solution for CEP and Stream Reasoning. Additionally our approach handles revision in event processing too.

6 Conclusions and Future Work

Complex Event Processing (CEP) deals with processing of continuously arriving events with the goal of identifying meaningful patterns, (complex events). In existing CEP approaches complex events consist merely of more simple (temporally situated) events. We proposed a *knowledge-based* event processing, advocating a richer formalism for CEP, capable not only to match patterns based on *temporal* relations among events but also to evaluate *contextual knowledge* and prove their *semantic* relations. Moreover, we proposed a framework which enables *revision* in pattern matching. We have demonstrated that our deductive rule-based approach for CEP represents a natural way to realize knowledge-based CEP with Stream Reasoning, and express routines required for event revisions.

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., 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)

4. Arasu, A., Babu, S., Widom, J.: The cql continuous query language: semantic foundations and query execution. VLDB Journal 15, 121–142 (2006)
5. Barbieri, D.F., Braga, D., Ceri, S., Grossniklaus, M.: An execution environment for c-sparql queries. In: EDBT, pp. 441–452 (2010)
6. Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: Incremental reasoning on streams and rich background knowledge. In: Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T. (eds.) ESWC 2010. LNCS, vol. 6088, pp. 1–15. Springer, Heidelberg (2010)
7. Barga, R.S., Goldstein, J., Ali, M.H., Hong, M.: Consistent streaming through time: A vision for event stream processing. In: Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR 2007), pp. 363–374 (2007)
8. Bolles, A., Grawunder, M., Jacobi, J.: Streaming SPARQL - extending SPARQL to process data streams. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 448–462. Springer, Heidelberg (2008)
9. Carney, D., et al.: Monitoring streams: a new class of data management applications. In: VLDB 2002, pp. 215–226 (2002)
10. Le-Phuoc, D., Parreira, J.X., Hausenblas, M., Hauswirth, M.: Unifying stream data and linked open data. DERI Technical Report (August 15, 2010)
11. Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley, Reading (2002)
12. Maskey, A.S., Cherniack, M.: Replay-based approaches to revision processing in stream query engines. In: SSPS, pp. 3–12 (2008)
13. Mei, Y., Madden, S.: Zstream: a cost-based query processor for adaptively detecting composite events. In: SIGMOD, pp. 193–206 (2009)
14. Ryvkina, E., Maskey, A.S., Cherniack, M., Zdonik, S.: Revision processing in a stream processing engine: A high-level design. In: ICDE 2006, USA, pp. 141–143 (2006)
15. Valle, E.D., Ceri, S., van Harmelen, F., Fensel, D.: It's a streaming world! reasoning upon rapidly changing information. IEEE Intelligent Systems 24, 83–89 (2009)