Applied Artificial Intelligence
Vol. 00, No. 00, Month-Month 200x, 1–34

Real-Time Complex Event Recognition and Reasoning – A Logic Programming Approach

Darko Anicic⋆, Sebastian Rudolph†, Paul Fodor‡, Nenad Stojanovic*

FZI Research Center for Information Technology, Germany⋆*
AIFB, Karlsruhe Institute of Technology, Germany†
Stony Brook University, Stony Brook, NY, U.S.A‡

(*v1.0 December 2010*)

Complex Event Processing (CEP) deals with the analysis of streams of continuously arriving events with the goal of identifying instances of predefined meaningful *patterns* (complex events). Complex events are detected in order to trigger time-critical actions in many areas including sensors networks, financial services, transaction management, business intelligence, etc. In existing approaches to CEP, a complex event is represented as a composition of more simple events satisfying certain *temporal* relationships. In this article, we advocate a *knowledge-rich* CEP, which apart from events, also processes additional (contextual) knowledge (e.g., in order to prove semantic relations among matched events or to define more complex situations). In particular, we present a novel approach for realizing knowledge-rich CEP, including detection of semantic relations among events and *reasoning*. 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

Recently, there has been a significant paradigm shift toward *real-time* information processing in research as well as in industry. Most businesses today collect large volumes of data continuously, and it is absolutely essential for them to process this data in real time so that they can take time-critical actions Luckham (2002). Real-time computing has raised significant interest due to its wide applicability in areas such as sensor networks (for on-the-fly interpretation of sensor data), financial services (for dynamic tracking of stock fluctuations as well as surveillance for frauds and money laundering), ad-hoc business process management (to detect situations that demand process changes in a timely fashion), network traffic monitoring (to detect and predict potential traffic problems), location based services (for real-time tracking and service operation), Web click analysis (for real-time analysis of users interaction with a web site and adaptive content delivery) and so forth.

Classical database systems and data warehouses are concerned with what happened in the past. In contrast thereto, Complex Event Processing (CEP) is about processing *events* upon their occurrence, with the goal to detect what has just happened or what is about to happen. An *event* represents something that occurs, happens or changes the current state of affairs. For example, an event may represent a sensor reading, a stock price change, a complied transaction, a new piece of information, a content update made available by a Web service and so forth. In all these situations, it is reasonable to compose simple (atomic) events into *complex* events, in order to structure the course of affairs and describe more complex dynamic situations. CEP deals with *real-time recognition* of such complex events, i.e., it processes continuously arriving events with the aim of identifying occurrences of meaningful event patterns (complex events).

High throughput and timeliness represent two main requirements to today's CEP systems Agrawal *et al.* (2008); Mei & Madden (2009); Barga *et al.* (2007); Arasu *et al.* (2006); Krämer & Seeger (2009); Chandrasekaran *et al.* (2003). Facing high-frequency event occurrences and the necessity of real-time responses, the matching of event patterns against unbound event streams constitutes indeed a challenge in its own right. Yet, the question remains

Email: ⋆ darko.anicic@fzi.de, † rudolph@kit.edu, ‡ pfodor@cs.sunysb.edu, * nstojano@fzi.de

Applied Artificial Intelligence
ISSN 0952-813X print/ ISSN 1362-3079 online
http://www.tandf.co.uk/journals
DOI: 10.1080/09528130xxxxxxxxxxxx

whether sole pattern matching functionality is enough to ensure appropriate responses and meets the sophisticated requirements of event-driven applications. In many applications, real-time actions need to be triggered not only by events, but also upon evaluation of additional *background knowledge*. This knowledge captures the *domain of interest*, or *context* related to critical actions and decisions. Its purpose is to be evaluated during detection of complex events in order to on the fly *enrich* events with relevant background information; to detect more complex *situations*; to propose certain intelligent *recommendations* in real-time; or to accomplish complex event *classification*, *clustering*, *filtering* and so forth.

There exists already a lot of knowledge available online that can be used in conjunction with event processing. For example, the Linked Open Data (LOD) initiative[1] has made available on the Web hundreds of datasets and ontologies such as live-linked open sensor data[2], UK governmental data[3], the New York Times dataset[4], financial ontologies[5], encyclopedic data (e.g., DBpedia), linked geo-data[6]. This knowledge is commonly represented as *structured data* (using RDF Schema Brickley *et al.* (10 February 2004)). Structured data allows us to define meanings, structures and semantics of information that is understandable for humans and intelligently processable by machines. Moreover, structured data enables *reasoning* over explicit knowledge in order to infer new (implicit) information. Current CEP systems Agrawal *et al.* (2008); Mei & Madden (2009); Barga *et al.* (2007); Arasu *et al.* (2006); Krämer & Seeger (2009); Chandrasekaran *et al.* (2003) however cannot utilize this structured knowledge and cannot reason about it. In this work, we address this issue, and provide a framework for *event recognition* and *reasoning* over event streams and domain knowledge.

Our approach is based on declarative (logic) rules. It has been shown Artikis *et al.* (2010); Kowalski & Sergot (1986); Miller & Shanahan (1999); Lausen *et al.* (1998); Alferes *et al.* (2006); Bry & Eckert (2007); Haley (1987); Paschke *et al.* (2010) that logic-based approaches for event processing have various advantages. First, they are *expressive* enough and convenient to represent diverse complex event patterns and come with a clear *formal declarative semantics*; as such, they are free of operational side-effects. Second, integration of *query processing* with event processing is easy and natural (including, e.g., the processing of *recursive* queries). Third, our experience with the deployment of logic rules is very positive and encouraging in terms of implementation effort for the main constructs in CEP as well as in providing extensibility of a CEP system (e.g., the number of code lines is significantly smaller than in procedural programming). Ultimately, a logic-based event model allows for *reasoning* over events, their relationships, entire states, and possible contextual knowledge available for a particular domain. Simultaneously reasoning about *temporal* knowledge (concerning events) and *static* or *evolving* knowledge (such as facts, rules and ontologies) is a capability beyond the state of the art in CEP Agrawal *et al.* (2008); Mei & Madden (2009); Barga *et al.* (2007); Arasu *et al.* (2006); Krämer & Seeger (2009); Chandrasekaran *et al.* (2003). In this paper we present a framework, capable of complex event processing and reasoning over temporal and static knowledge.

1.1 *Contributions*

The main contributions of this paper are as follows:

- **Formalism for real-time event recognition and reasoning.** We define an expressive complex event description language, called *ETALIS Language for Events* with a rule-based syntax and a clear declarative formal semantics. We extend the language, as defined in our previous work Anicic *et al.* (2010), to accommodate *static* rules. While event rules are used to capture patterns of complex events, the static rules account for (static) background knowledge about the considered domain. In comparison to Anicic *et al.* (2010), we further extend the language to express complex *iterative* patterns over unbound event streams, and apply certain aggregation functions over *sliding windows*. The language is founded on a new execution model that compiles complex event patterns into logic rules and enables timely, event-driven detection of complex events. The proposed formalism is expressive enough to capture the set of all possible thirteen temporal relations on time intervals, defined in Allen's Interval Algebra Allen (1983). Since the language with its extensions is based on declarative semantics, it is suitable for

[1] see `http://linkeddata.org/`
[2] Live linked open sensor data: `http://sensormasher.deri.org/`
[3] OpenPSI project: `http://www.openpsi.org/`
[4] Linked Open Data from the New York Times: `http://data.nytimes.com/`
[5] Financial ontology: `http://www.fadyart.com/`
[6] LinkedGeoData: `http://linkedgeodata.org`

deductive reasoning over event streams and the domain knowledge. The language is also general enough to support extensions with respect to other operators and features required in event processing (e.g., event consumption policies).

- **Efficient execution model.** We develop an efficient, *event-driven*, execution model for patterns written in *ETALIS Language for Events*. We extend the operational semantics from Anicic *et al.* (2009, 2010) to accommodate static rules, as well as, iterative and aggregative patterns. The model has inferencing capabilities and yet good run-time characteristics. It provides a flexible transformation of complex event patterns into intermediate patterns, so called *goals*. The status of achieved goals (at the current state) shows the progress toward matching event patterns. Goals are automatically asserted (satisfied) as relevant events occur. They can persist over a period of time "waiting" in order to support detection of a more complex goal or a complete pattern. Important characteristics of these goals are that they are asserted only if they may be needed 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 Prolog-style rules, which are executed in the backward chaining mode. Finally, expired goals are also deleted by such rules to free up the memory.
- **Implementation and evaluation.** We have implemented the proposed *ETALIS Language for Events* in a Prolog-based prototype. The implementation is open source[1]. Further on, we have developed a set of experiments to evaluate the overall approach. Our experiments are related to a sensor network, dedicated to measurements of environmental phenomena (e.g., weather observations such as wind, temperature, humidity, precipitation, visibility etc.). The evaluation has been conducted on real sensor data, and results are presented here.

The paper is organized as follows. In Section 2, we introduce *ETALIS Language for Events*. We define the syntax, and the declarative semantics of the language. Further on, iterative and aggregative complex event patterns are discussed; and theoretical properties of the presented formalism are given. Section 3 describes in details an execution model of our language. It also explains how complex event patterns are incrementally computed in (near) real time. Event consumption policies and memory management techniques are also presented. In Section 5, we discuss how deductive reasoning can be used to extend Complex Event Processing. On few examples, we demonstrate use of logic rules for event classification, filtering, and reasoning over events and background knowledge. We discuss implementation details of our formalism, and give evaluation results of conducted experiments in Section 6. Section 7 reviews existing work in this area, and compares it to ours. Finally, Section 8 summarizes the paper, and gives an outline of the future work.

2 Expressive Logic Rule-based Formalism for Complex Event Processing

We now define syntax and semantics of the ETALIS formalism, featuring (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.

2.1 *Syntax*

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 $\mathbf{P}^{\mathrm{s}}_n$

[1] ETALIS source code: `http://code.google.com/p/etalis/`

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 / event) atoms* as the set of all expressions $\mathtt{p}_n(t_1, \ldots, t_n)$ where $\mathtt{p}$ is a (static / 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 $\mathbf{P}^{\mathrm{e}}_n$. 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 either a variable or a constant. 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, EQUALS, STARTS, or FINISHES.[1] 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 shape

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

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

Figure 1 demonstrates the various ways of constructing complex event descriptions from simpler ones in ETALIS Language for Events. Moreover, the figure informally introduces the semantics of the language, which will further be defined in Section 2.2.

Let us assume that instances of three complex events, P_1, P_2, P_3, are occurring in time intervals as shown in Figure 1. Vertical dashed lines depict different time units, while the horizontal bars represent detected complex events for the given patterns. In the following, we give the intuitive meaning for all patterns from the figure:

- $(P_1).3$ detects an occurrence of P_1 if it happens within an interval of length 3, i.e., 3 represents the (maximum) time window.
- P_1 SEQ P_3 represents a sequence of two events, i.e., an occurrence of P_1 is followed by an occurrence of P_3; here P_1 must end before P_3 starts.
- P_2 AND P_3 is a pattern that is detected when instances of both P_2 and P_3 occur no matter in which order.
- P_1 PAR P_2 occurs when instances of both P_1 and P_2 happen, provided that their intervals have a non-zero overlap.
- P_2 OR P_3 is triggered for every instance of P_2 or P_3.
- P_1 DURING $(0$ SEQ $6)$ happens when an instance of P_1 occurs during an interval; in this case, the interval is built using a sequence of two atomic time-point events (one with $q = 0$ and another with $q = 6$, see the syntax above).
- P_3 STARTS P_1 is detected when an instance of P_3 starts at the same time as an instance of P_1 but ends earlier.
- P_1 EQUALS P_3 is triggered when the two events occur exactly at the same time interval.

[1] Hence, the defined pattern language captures all possible 13 relations on two temporal intervals as defined in Allen (1983).

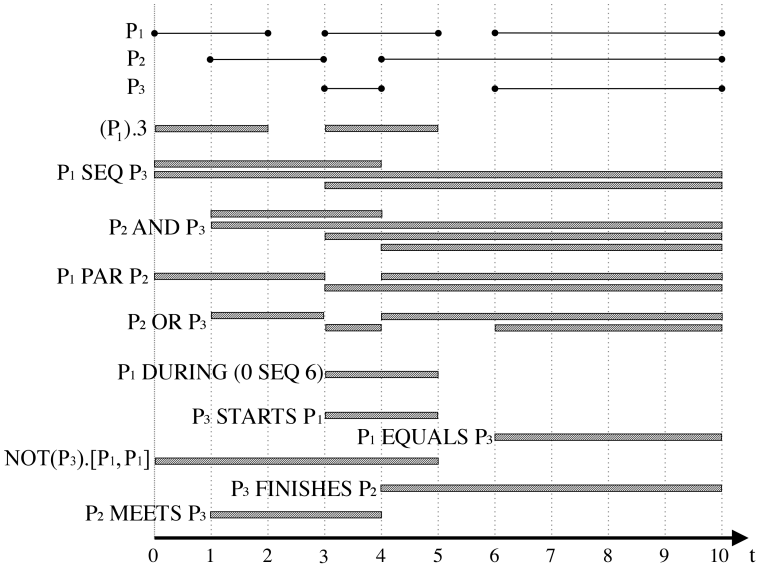

Figure 1. Language for Event Processing - Composition Operators

- NOT$(P_3).[P_1, P_1]$ represents a negated pattern. It is defined by a sequence of events (delimiting events) in the square brackets where there is no occurrence of P_3 in the interval. In order to invalidate an occurrence of the pattern, an instance of P_3 must happen in the interval formed by the end time of the first delimiting event and the start time of the second delimiting event. In this example delimiting events are just two instances of the same event, i.e., P_1. Different treatments of negation are also possible, however we use one from Adaikkalavan & Chakravarthy (2006) that is well adopted in CEP.
- P_3 FINISHES P_2 is detected when an instance of P_3 ends at the same time as an instance of P_1 but starts later.
- P_2 MEETS P_3 happens when the interval of an occurrence of P_2 ends exactly when the interval of an occurrence of P_3 starts.

It is worth noting that the defined pattern language captures the set of all possible 13 relations on two temporal intervals as defined in Allen (1983). The set can also be used for rich temporal reasoning.

In this example, event patterns are considered under the *unrestricted policy*. In event processing, consumption policies deal with an issue of *selecting* particular events occurrences when there are more than one event instance applicable and *consuming* events after they have been used in patterns. We will discuss different consumption policies and their implementation in ETALIS Language for Events in Section 4.

It is worthwhile to briefly consider the modeling capabilities of the presented pattern language. To do so, let us show few examples related to real-time observations and measurements of environmental phenomena (e.g., weather observations of temperature, relative humidity, wind speed and direction, precipitation etc.). For instance, one might be interested in defining an event that detects increase in wind speed at certain location Loc. Even more elaborate constraints can be put on the applicability of a pattern by endowing it with a boolean type term as filter[1]. Thus, we can detect a wind speed increase of at least 10%:

$$\begin{aligned}&\texttt{WindSpeedIncrease}(Loc, WSpd_2) \leftarrow \\ &\quad \texttt{Wind}(Loc, WSpd_1) \text{ SEQ } \texttt{Wind}(Loc, WSpd_2) \text{ WHERE } WSpd_2 > WSpd_1 \cdot 1.1.\end{aligned} \tag{1}$$

Let us now define an event denoting duration of a fire at certain location:

$$\begin{aligned}&\texttt{ActiveFire}(Loc) \leftarrow \\ &\quad \text{NOT}(\texttt{FireLocalized}(Loc))[\texttt{FireReported}(Loc), \texttt{FireLocalized}(Loc)].\end{aligned} \tag{2}$$

[1]Note that also comparison operators like $=, <$ and $>$ can be seen as boolean-typed binary functions and, hence, fit well into the framework.

We can also combine `WindSpeedIncrease` event from (1) to form a new complex event, `FireAlarm`:

$$\begin{array}{l}\texttt{FireAlarm}(Loc) \leftarrow \\ \quad \text{NOT}(\texttt{FireLocalized}(Loc, WSpd)).[\texttt{FireReported}(Loc), \texttt{WindSpeedIncrease}(Loc, WSpd))].\end{array} \tag{3}$$

Similarly, we might be interested in detecting the heat index, i.e., an index that combines air temperature and relative humidity in an attempt to determine the human-perceived equivalent temperature (how hot it feels):

$$\begin{array}{l}\texttt{HeatIndex}(Loc, Index(Tmp, Hum)) \leftarrow \\ \quad \big(\texttt{Temperature}(Loc, Tmp) \text{ AND } \texttt{Humidity}(Loc, Hum)\big).30min\end{array} \tag{4}$$

For the definition of the function $Index$, see Wikipedia.[1] Note that we have also defined a time frame of 30 minutes in which temperature and humidity readings are expected from respective sensors. This event rule also shows, how event information (about an index or other data) can be "passed" on to the defined complex events by using variables. In general, variables may be employed to conditionally group events or their attributes into complex ones if they refer to the same entity.

We will gradually introduce more complex patterns later on in this section, as well as, in Sections 3 and 5 (as we introduce other aspects of the language).

2.2 *Declarative Semantics*

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:

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

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 $\epsilon : Ground^{\mathrm{e}} \to 2^{\mathbb{Q}^+}$ 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 $\mathcal{I} : Ground^{\mathrm{e}} \to 2^{\mathbb{Q}^+ \times \mathbb{Q}^+}$ 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 $g \in Ground^{\mathrm{e}}$. 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:

(i) $\langle q, q \rangle \in \mathcal{I}(g)$ for every $q \in \mathbb{Q}^+$ and $g \in Ground^{\mathrm{e}}$ with $q \in \epsilon(g)$
(ii) 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 Fig. 2.

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 $\mathcal{I}$.

THEOREM 2.1 *For every event stream ϵ and rule base $\mathcal{R}$ there is a unique minimal model $\mathcal{I}^{\epsilon,\mathcal{R}}$.*

[1] The heat index: `http://en.wikipedia.org/wiki/Heat_index`

pattern	$\mathcal{I}_\mu$(pattern)
$\mathtt{p}^{\mathrm{e}}(t_1,\ldots,t_n)$	$\mathcal{I}(\mathtt{p}^{\mathrm{e}}(\mu^*(t_1),\ldots,\mu^*(t_n)))$
P WHERE t	$\mathcal{I}_\mu(P)$ if $\mu^*(t) = true$ $\emptyset$ otherwise.
q	$\{\langle q,q\rangle\}$ for all $q{\in}\mathbb{Q}^+$
$(P).q$	$\mathcal{I}_\mu(P) \cap \{\langle q_1,q_2\rangle \mid q_2 - q_1 \leq q\}$
$P1$ SEQ $P2$	$\{\langle q_1,q_4\rangle \mid \langle q_1,q_2\rangle{\in}\mathcal{I}_\mu(P1)$ and $\langle q_3,q_4\rangle{\in}\mathcal{I}_\mu(P2)$ and $q_2{<}q_3\}$
$P1$ AND $P2$	$\{\langle \mathsf{min}(q_1,q_3),\mathsf{max}(q_2,q_4)\rangle \mid \langle q_1,q_2\rangle{\in}\mathcal{I}_\mu(P1)$ and $\langle q_3,q_4\rangle{\in}\mathcal{I}_\mu(P2)\}$
$P1$ PAR $P2$	$\{\langle \mathsf{min}(q_1,q_3),\mathsf{max}(q_2,q_4)\rangle \mid \langle q_1,q_2\rangle{\in}\mathcal{I}_\mu(P1)$ and $\langle q_3,q_4\rangle{\in}\mathcal{I}_\mu(P2)$ and $\mathsf{max}(q_1,q_3){<}\mathsf{min}(q_2,q_4)\}$
$P1$ OR $P2$	$\mathcal{I}_\mu(P1) \cup \mathcal{I}_\mu(P2)$
$P1$ EQUALS $P2$	$\mathcal{I}_\mu(P1) \cap \mathcal{I}_\mu(P2)$
$P1$ MEETS $P2$	$\{\langle q_1,q_3\rangle \mid \langle q_1,q_2\rangle{\in}\mathcal{I}_\mu(P1)$ and $\langle q_2,q_3\rangle{\in}\mathcal{I}_\mu(P2)\}$
$P1$ DURING $P2$	$\{\langle q_3,q_4\rangle \mid \langle q_1,q_2\rangle{\in}\mathcal{I}_\mu(P1)$ and $\langle q_3,q_4\rangle{\in}\mathcal{I}_\mu(P2)$ and $q_3{<}q_1{<}q_2{<}q_4\}$
$P1$ STARTS $P2$	$\{\langle q_1,q_3\rangle \mid \langle q_1,q_2\rangle{\in}\mathcal{I}_\mu(P1)$ and $\langle q_1,q_3\rangle{\in}\mathcal{I}_\mu(P2)$ and $q_2{<}q_3\}$
$P1$ FINISHES $P2$	$\{\langle q_1,q_3\rangle \mid \langle q_2,q_3\rangle{\in}\mathcal{I}_\mu(P1)$ and $\langle q_1,q_3\rangle{\in}\mathcal{I}_\mu(P2)$ and $q_1{<}q_2\}$
NOT$(P1).[P2,P3]$	$\mathcal{I}_\mu(P2$ SEQ $P3) \setminus \mathcal{I}_\mu(P2$ SEQ $P1$ SEQ $P3)$

Figure 2. 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.

Proof For every rational number q with $q \in \mathbb{Q}_\epsilon = \bigcup_{g\in Ground^{\mathrm{e}}} \epsilon(g)$, we define an interpretation $\mathcal{I}_q$ by bottom-up saturation of ϵ_q where $\epsilon_q(g) = \epsilon(g) \cap \{\langle q_1,q_2\rangle \mid q_2 \leq q\}$ under the rules of $\mathcal{R}$ where the NOT subexpressions are evaluated against $\bigcup_{q'\in\mathbb{Q}_\epsilon, q'<q} \mathcal{I}_{q'}$. The minimal model can then be defined by $\mathcal{I}^{\epsilon,\mathcal{R}} := \bigcup_{q\in\mathbb{Q}_\epsilon} \mathcal{I}_q$. Minimality is a straightforward consequence of the fact that derived intervals always contain the intervals associated to the premise atoms due to the definition of the semantics of patterns (cf. Fig. 2). □

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.[1] This 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.

2.3 *Complexity Properties*

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 Dantsin *et al.* (2001).

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 2.2 *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 where the number of variables per rule is bounded by 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. Dantsin *et al.* (2001).

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

[1] 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 $\epsilon_2,\mathcal{R} \models a^{[q_1,q_2]}$.

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 polynomial time, given the above mentioned variable 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. □

2.4 *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).[1] 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. Within the ETALIS formalism, this situation can be handled via recursive event rules.

As an example, assume `TempIncrease` event should be triggered whenever the temperature rises over a previous maximum, and further `TempAlarm` event is triggered if the maximum gets over 100 degrees Fahrenheit. For this, we have to iterate whenever there is a new maximum temperature indicated by the atomic `Temp` events. This can be realized by the below set of rules.

$$\begin{aligned} \mathtt{TempIncrease}(T) &\leftarrow \mathtt{Temp}(T). \\ \mathtt{TempIncrease}(T_2) &\leftarrow \mathtt{TempIncrease}(T_1) \text{ SEQ } \mathtt{Temp}(T_2) \text{ WHERE } T_2 > T_1. \\ \mathtt{TempAlarm}(T) &\leftarrow \mathtt{TempIncrease}(T) \text{ WHERE } T > 100. \end{aligned} \tag{5}$$

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 $\mathtt{A}^N$) can be recognized by `Iteration(A,N)` defined as follows:

$$\begin{aligned} \mathtt{Iteration}(\mathtt{A}, 1) &\leftarrow \mathtt{A}. \\ \mathtt{Iteration}(\mathtt{A}, K+1) &\leftarrow \mathtt{A} \text{ SEQ } \mathtt{Iteration}(\mathtt{A}, K). \end{aligned}$$

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). The following rules implement such a pattern in ETALIS for the length equal to N (N is typically predefined):

[1] 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 $(a \text{ SEQ } b \text{ SEQ } c) \text{ EQUALS NOT}(a \text{ OR } c).[a, c]$.

$$\begin{aligned} \texttt{Iteration}(\texttt{A}, 1) &\leftarrow \texttt{A}. \\ \texttt{Iteration}(\texttt{A}, K+1) &\leftarrow \text{NOT}(\texttt{A}).[\texttt{A}, \texttt{Iteration}(\texttt{A}, K)]. \\ \texttt{E} &\leftarrow \texttt{Iteration}(\texttt{A}, N). \end{aligned}$$

For instance, for N=5, E will be triggered every time when the system encounters five occurrences of A.

3 Operational Semantics

In Section 2, we have defined complex event patterns formally. This section describes how complex events, described in ETALIS Language for Events, can be detected at run-time (following the semantics of the language). Our approach is established on *goal-directed, event-driven* rules and decomposition of complex event patterns into *two-input intermediate events* (i.e., *goals*). Goals are automatically asserted by rules as relevant events occur. They can persist over a period of time "waiting" to support detection of a more complex goal. This process of asserting more and more complex goals shows the progress towards detection of a complex event. In the following subsections, we give more details about a *goal-directed, event-driven* mechanism with respect to event pattern operators (formally defined in Section 2.2).

3.1 *Sequence of Events*

Let us consider a sequence of events represented by rule (6), i.e., E is detected when an event A[1]. is followed by B, and followed by C. We can always represent the above pattern as E ← ((A SEQ B) SEQ C). In general, rules (7) represent two equivalent rules.[2]

$$\texttt{E} \leftarrow \texttt{A SEQ B SEQ C}. \tag{6}$$

$$\begin{aligned} &\texttt{E} \leftarrow \texttt{P BIN R BIN S... BIN T}. \\ &\texttt{E} \leftarrow (((\texttt{P BIN R}) \texttt{ BIN S}) \texttt{... BIN T}). \end{aligned} \tag{7}$$

We refer to this kind of "events coupling" as *binarization* of events. Effectively, in binarization we introduce *two-input* intermediate events (IE). For example, now we can rewrite rule (6) as IE ← A SEQ B, and the E ← IE 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. Using the binarization, it is more convenient to construct *event-driven* rules for three reasons. First, it is easier to implement an event operator when events are considered on a "two by two" basis. Second, the binarization increases the possibility for *sharing* among events and intermediate events, when the granularity of intermediate patterns is reduced. Third, the binarization eases the *management* of rules. As we will see later in this section, each new use of an event (in a pattern) amounts to appending one or more rules to the existing rule set. However it is important that for the management of rules, we do not need to *modify* existing rules when adding new ones[3].

In the following, we give more details about assigning rules to each monitored event. We also provide an algorithm (using Prolog syntax) for detecting a sequence of events.

Algorithm 3.1 accepts as input a rule referring to a binary sequence IE ← A SEQ B, and produces *event-driven backward chaining rules* (EDBCR), i.e., executable rules for the sequence pattern. The binarization step must precede the rule transformation. Rules, produced by Algorithm 3.1, belong to one of two different classes of rules.

[1] More precisely, by "an event A" is meant an *instance* of the event A.

[2] If no parentheses are given, we assume all operators to be left-associative. While in some cases, like SEQ sequences, this is irrelevant, other operators such as PAR are not associative, whence the precedence matters.

[3] This holds even if patterns with negated events are added.

We refer to the first class as *goal inserting rules*. The second class corresponds to *checking rules*. For example, rule (8) belongs to the first class as it inserts goal(B(_, _),A(T_1, T_2),IE(_, _)). The rule will fire when A occurs, and the meaning of the goal it inserts is as follows: "an event A has occurred at $[T_1, T_2]$,[4] and we are waiting for B to happen in order to detect IE". The goal does not carry information about times for B and IE, as we do not 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* event is to specify what we are waiting for to detect an event that is on the *third* position.

Algorithm 3.1 Sequence.
Input: event binary goal IE ← A SEQ B.
Output: event-driven backward chaining rules for SEQ operator.
Each event binary goal IE ← A SEQ B is converted into: {

$$\begin{aligned}
&\mathtt{A}(T_1, T_2) :- \mathtt{for_each}(\mathtt{A}, 1, [T_1, T_2]).\\
&\mathtt{A}(1, T_1, T_2) :- \mathtt{assert}(\mathtt{goal}(\mathtt{B}(_, _), \mathtt{A}(T_1, T_2), \mathtt{IE}(_, _))).\\
&\mathtt{B}(T_3, T_4) :- \mathtt{for_each}(\mathtt{B}, 1, [T_3, T_4]).\\
&\mathtt{B}(1, T_3, T_4) :- \mathtt{goal}(\mathtt{B}(T_3, T_4), \mathtt{A}(T_1, T_2), \mathtt{IE}), T_2 < T_3,\\
&\qquad\qquad \mathtt{retract}(\mathtt{goal}(\mathtt{B}(T_3, T_4), \mathtt{A}(T_1, T_2), \mathtt{IE}(_, _))), \mathtt{IE}(T_1, T_4).
\end{aligned}$$

}

Rule (9) belongs to the second class being a *checking rule*. It checks whether certain prerequisite goals already exist in the database, in which case it triggers the more complex event. For example, rule (9) will fire whenever B occurs. The rule checks whether goal(B(T_3, T_4),A(T_1, T_2),IE) already exists (i.e., A has previously happened), in which case the rule triggers IE by calling 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., A(T_1, T_2), and B(T_3, T_4), see Section 2.2). Calling IE(T_1, T_4), this event is effectively propagated either upward (if it is an intermediate event) or triggered as a finished complex event.

We see that our *backward* chaining rules compute goals in a *forward* chaining manner. The goals are crucial for computation of complex events. They show the current state of progress toward matching an event pattern. Moreover, they allow for determining the "completion state" of any complex event, at any time. For instance, we can query the current state and get information how much of a certain pattern is currently fulfilled (e.g., what is the current status of certain pattern, or notify me if the pattern is 90% completed). Further, goals can enable *reasoning* over events (e.g., answering which event occurred before some other event, although we do not know a priori what are explicit relationships between these two; correlating complex events to each other; establishing more complex constraints between them and so forth, see Section 5). Goals can persist over a period of time. It is worth noting that *checking rules* can also delete goals. Once a goal is "consumed", it is removed from the database[1]. In this way, goals are kept persistent as long as (but not longer than) needed. In Section 4, we will return to different policies for removing goals from the database.

$$A(1, T_1, T_2) :- \mathtt{assert}(\mathtt{goal}(\mathtt{B}(\_, \_), A(T_1, T_2), IE(\_, \_))). \tag{8}$$

$$\begin{aligned}
\mathtt{B}(1, T_3, T_4) :- &\ \mathtt{goal}(\mathtt{B}(T_3, T_4), \mathtt{A}(T_1, T_2), \mathtt{IE}), T_2 < T_3,\\
&\ \mathtt{retract}(\mathtt{goal}(\mathtt{B}(T_3, T_4), \mathtt{A}(T_1, T_2), \mathtt{IE}(_, _))), \mathtt{IE}(T_1, T_4).
\end{aligned} \tag{9}$$

$$\begin{aligned}
\mathtt{for_each}(Pred, N, L) :- &\ ((FullPred = ..[Pred, N, L]), \mathtt{event_trigger}(FullPred),\\
&\ (N_1 \; is \; N + 1), \mathtt{for_each}(Pred, N_1, L)) \vee \mathtt{true}.
\end{aligned} \tag{10}$$

Finally, in Algorithm 3.1 there exist more rules than the two mentioned types (i.e., rules inserting goals and checking rules). We see that for each different event type (i.e., A and B in our case) we have one rule with a

[4] Apart from the timestamp, an event may carry other data parameters. They are omitted here for the sake of readability.
[1] Removal of "consumed" goals is typically needed for space reasons but might be omitted if events are required in a log for further processing or analyzing.

for_each predicate. It is defined by rule (10). Effectively, it implements a loop, which for any occurrence of an event goes through each rule specified for that event (predicate) and fires it. For example, when A occurs, the first rule in the set of rules from Algorithm 3.1 will fire. This first rule will then loop, invoking all other rules specified for A (those having A in the rule head). In our case, there is only one such a rule, namely rule (8). However, in general, there may be as many of these rules as usages of a particular event in an event program. Let us observe a situation in which we want to extend our event pattern set with an additional pattern that contains the event A (i.e., additional usage of A). In this case, the rule set representing a set of event patterns needs to be updated with new rules. This can be done even at runtime. Let us assume the additional pattern to be monitored is $\texttt{IE}_j \leftarrow \texttt{K SEQ A}$. Then the only change we need to make is to add one rule to insert a goal and one checking rule (in the existing rule set). The change is sketched as an update of Algorithm 3.1 below[2].

Updating rules from Algorithm 3.1 to accommodate an additional usage of the event A.

$\texttt{A}(2, T_1, T_2) :- \texttt{assert}(\texttt{goal}(\texttt{B}(\_,\_), \texttt{A}(T_1, T_2), \texttt{IE}(\_,\_))).$

$\texttt{A}(3, T_1, T_2) :- \texttt{goal}(\texttt{A}(\_,\_), \texttt{K}(T_3, T_4), \texttt{IE}_j(\_,\_]), T_4 < T_1,$
$\texttt{retract}(\texttt{goal}(\texttt{A}(\_,\_), \texttt{K}(T_3, T_4), \texttt{IE}_j(\_,\_))), \texttt{IE}_j(T_3, T_2).$

So far, we have described in detail a mechanism for event processing with *data or event-driven backward chaining rules* (EDBCR). We have also described the transformation of event pattern rules into rules for real-time events detection using the *sequence* operator. In general, for a given set of rules (defining complex patterns) there will be as many transformed rules as there are usages of distinct atomic events. Some rules however may be *shared* among different patterns. As said, the binarization breaks up patterns into binary sub-patterns (intermediate events). If two or more patterns share the same sub-patterns, they will also share the same set of EDBCR. That is, during the transformation, only one set of EDBCR will be produced for a distinct event binary goal (no matter how many times the goal is used in the whole event program). In large programs (e.g., where event patterns are built incrementally, i.e., one pattern upon another one) such a sharing may improve the overall system performance as the execution of redundant rules is avoided.

The set of transformed rules is further accompanied by rules to implement loops (as many as there are distinct atomic events). The same procedure is repeated for intermediate events (for example, $\texttt{IE}_1$, $\texttt{IE}_2$). The complete transformation is proportional to the number and length of user defined event pattern rules, hence such a transformation is linear, and moreover is performed at design time.

Conceptually, our backward chaining rules for the sequence operator look very similar to rules for other operators. In the remaining part of this section we show the algorithms for other event operators, and briefly describe them.

3.2 *Conjunction of Events*

Conjunction is another typical operator in event processing. An event pattern based on conjunction occurs when all events which comprise that conjunction occur. Unlike the sequence operator, here the constitutive events can happen at times with no particular order between them. For example, $\texttt{IE} \leftarrow \texttt{A AND B}$ defines defines IE event as conjunction of events A and B.

Algorithm 3.2 shows the output of a transformation of *conjunction* event patterns into EDBCR (for conjunction). The procedure for dividing complex event rules into *binary event goals* is the same as in Algorithm 3.1. However, rules for *inserting* and *checking* goals are different. Both classes of rules are specific to conjunction. We have a pair of these rules created for both an event A as well as for B. Whenever A occurs (denoted as some interval (T_1, T_2)) the algorithm checks whether an instance of B has already happened (see rule (11) from Algorithm 3.2). An instance of B has already happened if the current database state contains $\texttt{goal}(\texttt{A}(\_,\_),\texttt{B}(T_1, T_2),\texttt{IE}(\_,\_))$. In this case the event $\texttt{IE}(T_5, T_6)$ is triggered (i.e., a call for $\texttt{IE}(T_5, T_6)$ is issued). Otherwise, a goal which states that an instance of A has occurred, is inserted (i.e., $\texttt{assert}(\texttt{goal}(\texttt{B}(\_,\_),\texttt{A}(T_1, T_2),\texttt{IE}(\_,\_)))$ is executed by rule (12)). Now if an

[2]Note that an *id* of rules is incremented for each next rule being added (i.e., 2,3...)

Algorithm 3.2 Conjunction.
Input: event binary goal IE ← A AND B.
Output: event-driven backward chaining rules for AND operator.
Each event binary goal IE ← A AND B is converted into: {

$$\begin{aligned}
\texttt{A}(T_1,T_2) :- & \ \texttt{for_each}(\texttt{A},1,[T_1,T_2]).\\
\texttt{A}(1,T_3,T_4) :- & \ \texttt{goal}(\texttt{A}(_,_),\texttt{B}(T_1,T_2),\texttt{IE}(_,_)),\\
& \ \texttt{retract}(\texttt{goal}(\texttt{A}(_,_),\texttt{B}(T_1,T_2),\texttt{IE}(_,_))),\\
& \ T_5=\texttt{min}\{T_1,T_3\},T_6=\texttt{max}\{T_2,T_4\},\texttt{IE}(T_5,T_6).\\
\texttt{A}(2,T_3,T_4) :- & \ \neg(\texttt{goal}(\texttt{A}(_,_),\texttt{B}(T_1,T_2),\texttt{IE}(_,_))),\\
& \ \texttt{assert}(\texttt{goal}(\texttt{B}(_,_),\texttt{A}(T_3,T_4),\texttt{IE}(_,_))).\\
\texttt{B}(T_1,T_2) :- & \ \texttt{for_each}(\texttt{B},1,[T_1,T_2]).\\
\texttt{B}(1,T_3,T_4) :- & \ \texttt{goal}(\texttt{B}(_,_),\texttt{A}(T_1,T_2),\texttt{IE}(_,_)),\\
& \ \texttt{retract}(\texttt{goal}(\texttt{B}(_,_),\texttt{A}(T_1,T_2),\texttt{IE}(_,_))),\\
& \ T_5=\texttt{min}\{T_1,T_3\},T_6=\texttt{max}\{T_2,T_4\},IE(T_5,T_6).\\
\texttt{B}(2,T_3,T_4) :- & \ \neg(\texttt{goal}(\texttt{B}(_,_),\texttt{A}(T_1,T_2),\texttt{IE}(_,_))),\\
& \ \texttt{assert}(\texttt{goal}(\texttt{A}(_,_),\texttt{B}(T_3,T_4),\texttt{IE}(_,_))).
\end{aligned}$$

}

instance of B happens later (at some (T_3, T_4)), rule (13) will succeed (if A has previously happened). Otherwise rule (14) will insert goal(A(_, _),B(T_1, T_2),IE(_, _)).

$$\begin{aligned}
\texttt{A}(1,T_3,T_4) :- & \ \texttt{goal}(\texttt{A}(_,_),\texttt{B}(T_1,T_2),\texttt{IE}(_,_)),\\
& \ \texttt{retract}(\texttt{goal}(\texttt{A}(_,_),\texttt{B}(T_1,T_2),\texttt{IE}(_,_))),\\
& \ T_5=\texttt{min}\{T_1,T_3\},T_6=\texttt{max}\{T_2,T_4\},\texttt{IE}(T_5,T_6).
\end{aligned} \tag{11}$$

$$\begin{aligned}
\texttt{A}(2,T_3,T_4) :- & \ \neg(\texttt{goal}(\texttt{A}(_,_),\texttt{B}(T_1,T_2),\texttt{IE}(_,_))),\\
& \ \texttt{assert}(\texttt{goal}(\texttt{B}(_,_),\texttt{A}(T_3,T_4),\texttt{IE}(_,_))).
\end{aligned} \tag{12}$$

$$\begin{aligned}
\texttt{B}(1,T_3,T_4) :- & \ \texttt{goal}(\texttt{B}(_,_),\texttt{A}(T_1,T_2),\texttt{IE}(_,_)),\\
& \ \texttt{retract}(\texttt{goal}(\texttt{B}(_,_),\texttt{A}(T_1,T_2),\texttt{IE}(_,_))),\\
& \ T_5=\texttt{min}\{T_1,T_3\},T_6=\texttt{max}\{T_2,T_4\},\texttt{IE}(T_5,T_6).
\end{aligned} \tag{13}$$

$$\begin{aligned}
\texttt{B}(2,T_3,T_4) :- & \ \neg(\texttt{goal}(\texttt{B}(_,_),\texttt{A}(T_1,T_2),\texttt{IE}(_,_))),\\
& \ \texttt{assert}(\texttt{goal}(\texttt{A}(_,_),\texttt{B}(T_3,T_4),\texttt{IE}(_,_))).
\end{aligned} \tag{14}$$

In Section 2.2 we have presented a *declarative* semantics of ETALIS Language for Events. We provide an implementation of the language in Prolog, and since Prolog is not purely declarative, we need to take care when using non-declarative features of Prolog[1]. Hence in the following we discuss whether the operational semantics - as presented so far in this section – corresponds to the declarative semantics of the language.

$$\begin{aligned}
\texttt{C} &\leftarrow \texttt{A}\ op_1\ \texttt{B}.\\
\texttt{D} &\leftarrow \texttt{B}\ op_2\ \texttt{C}.
\end{aligned} \tag{15}$$

Consider an example program defined by rules (15) and its corresponding graphical representation shown in Figure 3. Note that event B is used twice in rules (15), hence we have two edges in Figure 3. For each edge of B

[1]This remark applies, in general, when a declarative formalism is to be implemented with other non-declarative languages (e.g., procedural languages such as Java, C, C++, etc.)

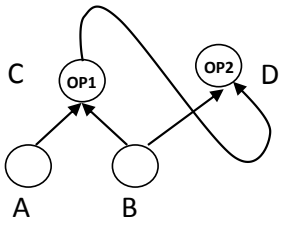

Figure 3. Example program

we will have one EDBC rule (e.g., if op_i is SEQ where i can be either 1 or 2) or two EDBC rules (e.g., if op_i is AND), see Algorithm 3.1 and Algorithm 3.2, respectively. To ensure the declarative property of the language, the order in which rules of these two edges are executed needs to be irrelevant. That is, if ETALIS system evaluates rule(s) from the first edge followed by evaluation of rule(s) from the second edge, we need to obtain the same results as if the order was the opposite. If this holds for every binary pair of events connected by any event operator in a program, then we can be sure that the operational semantics preserve the declarative property of the language.

Let us assume that both op_1 and op_2 in rules (15) is replaced by SEQ operator, and that event A happened followed by event B. In this situation we expect to derive event C only. Event D will not be triggered as event C did not strictly happened after event B. That is, T_2 of event B is not strictly smaller than T_1 of event C (essentially they are equal), see Algorithm 3.1. Consequentially, event D will not be detected regardless of the order in which rules for two B edges are evaluated.

Let us assume that op_2 in rules (15) is replaced by AND operator, and again, event A happened followed by event B. In this situation we expect to derive both event C and event D. When event B occurs, the system can first evaluate rule for the SEQ edge (op_1), and then rules for the AND edge (op_2), or vice versa. For both cases we expect event D to be triggered.

Suppose the SEQ edge of event B is evaluated first. The system will detect event C. This event will be used to start detection of the conjunction (defined by the second rule in rules (15)). Effectively, event B will trigger rule (13) and rule (14) in Algorithm 3.2[1]. Rule (13) will fail, and rule (14) will succeed by inserting goal (B $(\_, \_)$, C (T_3, T_4), D $(\_, \_)$). Next, when rules of the AND edge of event B are evaluated, rule (11) and rule (12) will fire[2]. Finally, rule (11) will succeed by triggering event D. We see that successful evaluation of rule (14), followed by successful evaluation of rule (11), leads to detection of event D.

Now suppose that the AND edge of event B is evaluated first. In this situation, rule (12) will be successfully evaluated followed by evaluation of rule (13). As a result, the detection will take place in the reverse order but it will be still possible to detect event D.

While Algorithm 3.1 enables detection of events in one direction, Algorithm 3.2 enables the detection in both directions. Therefore we use a modification of Algorithm 3.2 to handle other operators too (e.g., PAR , MEETS , FINISHES etc.), i.e., whenever binary events may come in both orders.

3.3 *Concurrency*

A concurrent or parallel composition of two events (IE $\leftarrow$ A PAR B) is detected when events A and B both occur, and their intervals overlap (i.e., we also say they happen *synchronously*).

Algorithm 3.3 shows what is an output of automated transformation of a *concurrent* event pattern into rules which serve a *data-driven backward chaining* event computation. The procedure for dividing complex event rules into *binary event goals* is the same (as already described), and takes place prior to the transformation. Rules for *inserting* and *checking* goals are similar to those in Algorithm 3.2. The only change in Algorithm 3.2 is a *sufficient* condition, ensuring the interval overlap (i.e., $T_3 < T_2$).

[1]Note that in the rule heads we now have event C.
[2]Note that in the rule heads we now have event B.

Algorithm 3.3 Concurrency.
Input: event binary goal IE ← A PAR B.
Output: event-driven backward chaining rules for PAR operator.
Each event binary goal IE ← A PAR B is converted into: {

$$\begin{aligned}
\texttt{A}(T_1,T_2) &:- \texttt{for_each}(\texttt{A},1,[T_1,T_2]).\\
\texttt{A}(1,T_3,T_4) &:- \texttt{goal}(\texttt{A}(_,_),\texttt{B}(T_1,T_2),\texttt{IE}(_,_)),T_3<T_2,\\
&\quad \texttt{retract}(\texttt{goal}(\texttt{A}(_,_),\texttt{B}(T_1,T_2),\texttt{IE}(_,_))),\\
&\quad T_5=\texttt{min}\{T_1,T_3\},T_6=\texttt{max}\{T_2,T_4\},\texttt{IE}(T_5,T_6).\\
\texttt{A}(2,T_3,T_4) &:- \neg(\texttt{goal}(\texttt{A}(_,_),\texttt{B}(T_1,T_2),\texttt{IE}(_,_))),T_3<T_2,\\
&\quad \texttt{assert}(\texttt{goal}(\texttt{B}(_,_),\texttt{A}(T_3,T_4),\texttt{IE}(_,_))).\\
\texttt{B}(T_1,T_2) &:- \texttt{for_each}(\texttt{B},1,[T_1,T_2]).\\
\texttt{B}(1,T_3,T_4) &:- \texttt{goal}(\texttt{B}(_,_),\texttt{A}(T_1,T_2),\texttt{IE}(_,_)),T_3<T_2,\\
&\quad \texttt{retract}(\texttt{goal}(\texttt{B}(_,_),\texttt{A}(T_1,T_2),\texttt{IE}(_,_))),\\
&\quad T_5=\texttt{min}\{T_1,T_3\},T_6=\texttt{max}\{T_2,T_4\},\texttt{IE}(T_5,T_6).\\
\texttt{B}(2,T_3,T_4) &:- \neg(\texttt{goal}(\texttt{B}(_,_),\texttt{A}(T_1,T_2),\texttt{IE}(_,_))),T_3<T_2,\\
&\quad \texttt{assert}(\texttt{goal}(\texttt{A}(_,_),\texttt{B}(T_3,T_4),\texttt{IE}(_,_))).
\end{aligned}$$

}

3.4 *Disjunction*

An algorithm for detecting *disjunction* (i.e., OR) of events is trivial. The disjunction operator divides rules into separate disjuncts, where each disjunct triggers the parent (complex) event. Therefore we omit presentation of the algorithm here.

3.5 *Negation*

Negation in event processing is typically understood as *absence* of an event that is negated. In order to create a time interval in which we are interested to detect absence of an event, we define a negated event in the scope of other complex events. Algorithm 3.5 describes how to handle negation in the scope of a sequence. It is also possible to detect negation in an arbitrarily defined time interval.

Algorithm 3.5 Negation.
Input: event pattern IE ← NOT(C).[A,B].
Output: event-driven backward chaining rules for negation.
Each event binary goal IE ← NOT(C).[A,B] is converted into: {

$$\begin{aligned}
\texttt{A}(T_1,T_2) &:- \texttt{for_each}(\texttt{A},1,[T_1,T_2]).\\
\texttt{A}(1,T_1,T_2) &:- \texttt{assert}(\texttt{goal}(\texttt{B}(_,_),\texttt{A}(T_1,T_2),\texttt{IE}(_,_))).\\
\texttt{B}(T_1,T_2) &:- \texttt{for_each}(\texttt{B},1,[T_1,T_2]).\\
\texttt{B}(1,T_5,T_6) &:- \texttt{goal}(\texttt{B}(_,_),\texttt{A}(T_1,T_2),\texttt{IE}(_,_)),\\
&\quad \neg(\texttt{goal}(_,\texttt{C}(T_3,T_4),_)),T_2<T_5,T_2<T_3,T_4<T_5,\\
&\quad \texttt{retract}(\texttt{goal}(\texttt{B}(_,_),\texttt{A}(T_1,T_2),\texttt{IE}(_,_))),\texttt{IE}(T_1,T_6))).\\
\texttt{C}(T_1,T_2) &:- \texttt{for_each}(\texttt{C},1,[T_1,T_2]).\\
\texttt{C}(1,T_1,T_2) &:- \texttt{assert}(\texttt{goal}(_,\texttt{C}(T_1,T_2),_)).\}
\end{aligned}$$

}

Rules for detection of negation are similar to rules from Algorithm 3.1. We need to detect a sequence (i.e., A SEQ B), and additionally to check whether an occurrence of C happened in-between the event A and B. That is why a rule $\texttt{B}(1,T_5,T_6)$ needs to check whether $\neg(\texttt{goal}(\_,C(T_3,T_4),\_))$ (with certain time condition) is true. If yes, this means that an C has not happened during a detected sequence (i.e., $\texttt{A}(T_1,T_2)$ SEQ $\texttt{B}(T_5,T_6)$), and $\texttt{IE}(T_1,T_6)$ will be triggered. It is worth noting that a non-occurrence of C is monitored from the time when A has been detected until the beginning of an interval which the event B is detected on.

3.6 *Interval-based Operations*

In the following part of this section we provide brief descriptions for the remaining relations between two intervals. Each relation is easily checkable with one rule.

Duration. An event happens during (i.e., DURING) another event if the interval of the first is contained in the interval of the second. Rule (16) takes two intervals as parameters[1]. First, it checks whether all parameters are actually defined as intervals (see rule (17)). Then it compares whether the start of the second interval (TI_2_S) is less than the start of the first interval (TI_1_S). Additionally it checks whether the end of the first interval (TI_1_E) is less than the end of the second interval (TI_2_E).

$$\begin{aligned} \texttt{duration}(TI_1, TI_2) :- & \; TI_1 = [TI_1\_S, TI_1\_E], \texttt{validTimeInterval}(TI_1), \\ & \; TI_2 = [TI_2\_S, TI_2\_E], \texttt{validTimeInterval}(TI_2), \\ & \; TI_2\_S@ < TI_1\_S, TI_1\_E@ < TI_2\_E. \end{aligned} \tag{16}$$

$$\texttt{validTimeInterval}(TI) :- TI = [TI\_S, TI\_E], TI\_S@ < TI\_E. \tag{17}$$

Note that, to implement the operator DURING , we still need EDBC rules, similar to those, e.g., for SEQ operator (see Algorithm 3.1), PAR operator (see Algorithm 3.3)etc. EDBC rules for DURING operator are similar to those of PAR operator with the following difference. Instead of the condition $T_3 < T_2$, which ensures that two time intervals overlap, we now have a condition that one interval needs to be contained in another one. Hence $T_3 < T_2$ in Algorithm 3.3 is simply replaced by `duration` predicate defined as rule (16). The same remark applies to other interval-based operations presented below. Therefore, for space reasons, we omit presentation of complete sets of EDBC rules for each interval-based operator. Instead, we give only a rule that implements the condition for a corresponding time interval operation.

Start Relation. We say that an event starts another if an instance of the first event starts at the same time as an instance of the second event, but ends earlier. Therefore rule (18) checks whether the start of both intervals are equal and whether the end of the first event is smaller than the end of the second one.

$$\begin{aligned} \texttt{starts}(TI_1, TI_2) :- & \; TI_1 = [TI_1\_S, TI_1\_E], \texttt{validTimeInterval}(TI_1), \\ & \; TI_2 = [TI_2\_S, TI_2\_E], \texttt{validTimeInterval}(TI_2), \\ & \; TI_1\_S = TI_2\_S, TI_1\_E@ < TI_2\_E. \end{aligned} \tag{18}$$

Equal Relation. Two events are equal if they happen right at the same time. Rule (19) implements this relation.

$$\begin{aligned} \texttt{equals}(TI_1, TI_2) :- & \; TI_1 = [TI_1\_S, TI_1\_E], \texttt{validTimeInterval}(TI_1), \\ & \; TI_2 = [TI_2\_S, TI_2\_E], \texttt{validTimeInterval}(TI_2), \\ & \; TI_1\_S = TI_2\_S, TI_1\_E = TI_2\_E. \end{aligned} \tag{19}$$

Finish Relation. One event finishes another one if an occurrence of the first ends at the same time as an occurrence of the second event, but starts later. Rule (20) check this condition.

$$\begin{aligned} \texttt{finishes}(TI_1, TI_2) :- & \; TI_1 = [TI_1\_S, TI_1\_E], \texttt{validTimeInterval}(TI_1), \\ & \; TI_2 = [TI_2\_S, TI_2\_E], \texttt{validTimeInterval}(TI_2), \\ & \; TI_2\_S@ < TI_1\_S, TI_1\_E = TI_2\_E. \end{aligned} \tag{20}$$

[1] Symbol '@' is used in Prolog built-in predicates ($>, <, \geq$ etc.) to compare terms alphabetically or numerically. When this symbol is omitted, terms are compared arithmetically.

Meet Relation. Two events meet each other when the interval of the first ends exactly when the interval of the second event starts. Hence, the condition $TI_1\_E = TI_2\_S$ in rule (21) is sufficient to detect this relation.

$$\begin{aligned} \texttt{meets}(TI_1, TI_2) :- & \; TI_1 = [TI_1\_S, TI_1\_E], \texttt{validTimeInterval}(TI_1), \\ & \; TI_2 = [TI_2\_S, TI_2\_E], \texttt{validTimeInterval}(TI_2), \\ & \; TI_1\_E = TI_2\_S. \end{aligned} \tag{21}$$

3.7 *Iterative and Aggregation Patterns*

In ETALIS Language for Events, *aggregate* functions are handled by utilizing *iterative* rules. The language offers a common set of aggregates[1]: $\texttt{sum}(Var)$ (sums the values of Var for all selected events), `count` (counts the number of solutions for all selected events from an unbound stream), `avg` (computes average, and is implemented as combination of sum and count), $\texttt{max}(Var)$ (computes the maximum value of Var for all selected events from an unbound stream), and $\texttt{min}(Var)$ (computes the minimum value of Var for all selected events).

The aggregate functions are computed incrementally, by starting with an initial value for the increment, and iterating the aggregate function over events. For any aggregate function we implement the following iterative pattern.

$$\begin{aligned} & \texttt{Iteration}(StartCntr = 0, StartVal) \leftarrow \\ & \quad \texttt{Start\_event}(StartVal). \\ & \texttt{Iteration}(OldCntr + 1, NewVal) \leftarrow \\ & \quad \texttt{Iteration}(OldCntr, OldVal) \text{ SEQ } \texttt{A}(AggArg) \\ & \quad \text{WHERE } \{\texttt{assert}(AggArg), \\ & \qquad \texttt{window}(WndwSize, OldCntr, OldVal, AggArg, NewVal)\}. \end{aligned} \tag{22}$$

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 in (22)). 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 operation (an aggregate function). To achieve this, the iterative rule contains the static part (WHERE clause) for two reasons: to save data from the seen events as history relevant w.r.t the aggregation function (see `assert`(AggArg) in Pattern (22)), 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).

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 defined by the following rule (written in Prolog syntax).

$$\begin{aligned} \texttt{window}(WndwSize, OldCntr, OldVal, AggArg, NewVal) :- & \\ & OldCntr + 1 >= WindowSize - > \\ & \texttt{retract}(LastItem), \\ & \texttt{spec\_aggregate}(OldValue, AggArg, NewValue); \\ & \texttt{spec\_aggregate}(OldValue, AggArg, NewValue). \end{aligned} \tag{23}$$

We check whether the current counter value exceeds the window size (i.e., the incremented old counter, $OldCntr{+}1 >= WindowSize$) in which case we retract the last item from the window (i.e., `retract`(LastItem)) and compute a specific aggregate function (`spec_aggregate`). If the current counter value does not exceed the window size, `spec_aggregate` function is computed without retraction. Recall that new data element ($AggArg$) was previously added by the second iterative rule (22).

[1] Custom aggregate functions, using different built-in operators, can also be implemented with no further restrictions.

Based on these iterative pattern and sliding window rules we can implement other various aggregation functions. For example, to give a reader feeling how the sum aggregate function can be implemented over a stream of events data, see pattern (24).

$$
\begin{aligned}
&\texttt{Sum}(StartCntr = 0, StartVal) \leftarrow \texttt{Start\_event}(StartVal).\\
&\texttt{Sum}(OldCntr + 1, NewSum) \leftarrow\\
&\qquad \texttt{Sum}(OldCntr + 1, OldSum)\ \text{SEQ}\ \texttt{A}(AggArg)\\
&\qquad\ \text{WHERE}\ \{\texttt{assert}(AggArg),\\
&\qquad \texttt{window}(WndwSize, OldCntr, OldSum + AggArg, AggArg, NewSum)\}.\\
&\texttt{window}(WndwSize, OldCntr, CurrSum, NewSum) :-\\
&\qquad OldCntr + 1 >= WindowSize- >\\
&\qquad \texttt{retract}(LastItem),\\
&\qquad NewSum = CurrSum - LastItem;\\
&\qquad NewSum = CurrSum - LastItem.
\end{aligned}
\tag{24}
$$

As already explained, the iteration begins when `Start_event` occurs and sets the $StartVal$. The iteration is further continued whenever event `A` happens. Note that events `Start_event` and `A` can be of the same type or two different events. We can additionally have WHERE clause to set filter conditions for both $StartVal$ and $AggArg$ (which we omit here to keep the pattern readable). However, it is clear that neither every `Start_event` must start the iteration, nor that every `A` must be accepted in an ongoing iteration. Finally, the `assert` predicate adds new data ($AggArg$) to the current `window`, 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). To do that, we can simply add $AvgVal = NewSum/(OldCntr + 1)$ in the WHERE clause of the second rule (as we have the current sum and the counter value).

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 an atomic event). We can also have *multiple* aggregations, computed on a single iterative pattern (and calculated over 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 over *sequences* of events (as it is common in other approaches Agrawal *et al.* (2008); Mei & Madden (2009); Gehani *et al.* (1992)). With no restriction, we can also use any other event operator, e.g., AND or PAR , instead of SEQ (in pattern (24)).

3.8 *Memory Management*

We have developed two ways to deal with outdated events (i.e., expired events with respect to a user specified time windows). The first technique modifies the binarization step by pushing the time constraints (set by pattern's time window information). The technique ensures that time window constraints are checked at each step during the incremental event detection. Therefore unnecessary intermediary sub-complex events will not be generated if the time constraints are violated.

Our second solution for garbage collection is to prune expired events (goals), by using periodic events generated by the system (EGS). Essentially, it enables events to be purged, depending on the time window constraints and the system clock. Similarly to the first technique, rules are defined with time window constraints and the binarization pushes the constraints to sub-components. This technique however does not check the constraints at each step during the incremental event detection, but periodically as EGS are scheduled to happen[1].

Our prototype implementation (see Section 6) also permits a general garbage collector removing events after

[1] The more memory is available in a running system, the rarer EGS are triggered.

absolute time (if one writes patterns without the time window constraints), as well as, dynamic time window constraints to be inserted/deleted on the fly.

4 Event Consumption Policies

When detecting a complex event, there may be several event occurrences (of the same type), that could be used to form that complex event. *Consumption policies* (or event contexts) deal with the issue of selecting particular occurrence(s), which will be used in the detection of a complex event. For example, let us consider rule (6) from Section 3.1, and a sequence of atomic events that happened in the following order: A(1), A(2), A(3), B(4), B(5), C(6) (where an event attribute denotes a time point when an event instance has occurred). We expect that, when an event of type B occurs, an intermediate event IE must be triggered. However, the question is, which occurrence of A will be selected to build that event, A(1), A(2) or A(3) (the same question applies for B)? Different consumption policies define different strategies. Here, we illustrate three widely used consumption policies: *recent*, *chronological*, and *unrestricted* policy Chakravarthy & Mishra (1994); Yoneki & Bacon (2005), and show how they can be naturally implemented by rules in our framework.

Note however that consumption policies in CEP is a subject that is not in-line with declarative principles. A consumption policy typically selects one, out of several events occurrences, and defines how multiple occurrences of the same event are consumed. This, however, has a direct impact on event pattern rules. For instance, if an event occurrence is consumed by rule r_1, it may not be available to rule r_2, and vice versa. As a consequence the *order* in which rules r_1 and r_2 are evaluated matters (what is against the principle of declarative programming).

On the other hand, consumption policies are widely used in CEP. Therefore in the remaining part of this section we show how different consumption policies can be implemented in our formalism. However it should be noted that the declarative property of the formalism does not hold any more, when a certain consumption policy (apart from the unrestricted policy) is used. This remark holds, in general, for the other declarative formalisms which enable the use of consumption policies.

4.1 *Consumption Policies Defined on Time Points*

In the above example, we assumed that the stream of events A(1), A(2), A(3), B(4), B(5), C(6) contains only atomic events.

Recent Policy. With this policy, the most recent event of its type is considered to construct complex events. In our example, when B(4) occurs, A(3) will be selected to compose IE(3,4). After a more recent occurrence B(5) occurs, older (which are less recent) occurrences of B are deleted (i.e., they are no longer eligible for further compositions). The next pair, A(3),B(5), is selected to form IE(3,5). It replaces the less recent occurrence IE. Finally, when C(6) occurs, it will trigger E(3,6) (using IE(3,5) as the more recent occurrence of IE in comparison to IE(3,4)).

The recent policy can be easily implemented in our framework. Let us consider Algorithm 3.1 particularly the rule which inserts a goal (in our example, goal(B,A,IE)). Whenever an instance of A occurs, there will be a new goal inserted with a corresponding timestamp. For instance, for A(1), the goal(B(_),A(1),IE(_,_)) is added; for A(2), the goal(B(_),A(2),IE(_,_)) will be asserted, and so forth). If we insert these goals into the database using LIFO (Last In First Out) structure, we obtain the *recent policy*. In our prototype implementation, this is done with a rule of the following form:

$$\texttt{assert}(\texttt{goal}(X)) : -assertA(\texttt{goal}(X)). \tag{25}$$

$asserta$ is a standard Prolog built-in that adds a term to the *beginning* of the database. Whenever a goal is inserted to the database, it is put on the top of a relation. Hence whenever we read a goal, the one inserted last will be returned.

Chronological Policy. This policy "consumes" the events in chronological order. In our example, this means that A(1) and B(4) will form IE(1,4), and further A(2) followed by B(5) will trigger IE(2,5). When C(6) happens, it will trigger E(1,6).

It is straightforward to implement the chronological policy too. Now, the goals in Algorithm 3.1 are inserted in a FIFO (First In First Out) mode. Equivalently, we use the following rule to realize the chronological policy:

$$\texttt{assert}(\texttt{goal}(X)) :- assertz(\texttt{goal}(X)). \tag{26}$$

$assertz$ is a standard Prolog built-in that adds a term to the *end* of the database. Whenever a goal is inserted to the database, it is put at the end of a relation. Consequently, whenever we read a goal, the first inserted goal will be returned first.

Unrestricted Policy. In this policy, all occurrences are valid. Consequently, no event is consumed (and no event is deleted), which makes this policy not suitable for practical use. Going back to our example, this implies that we detect the following instances of IE: IE(1,4), IE(2,4), IE(3,4), IE(1,5), IE(2,5), IE(3,5). The event E will be triggered just as many times, that is: E(1,6), E(2,6), E(3,6)...

We obtain the unrestricted policy simply by not using the construct for deleting goals (i.e., retract) from the database. If we replace the rule for B(1) in Algorithm 3.1 with rule (27), even consumed goals will not be deleted from the database[1]. Hence they will be available for future compositions.

$$B(1) :- \texttt{goal}(\texttt{B}, \texttt{A}, \texttt{IE})\ \textsc{seq}\ \texttt{IE}. \tag{27}$$

Consumption policies are an important part of an event processing framework. We notice that different policies change the semantics of event operators. For example, with the same operator we have detected different complex events (the recent policy detects E(2,6), while the chronological policy detects E(1,6)).

4.2 *Consumption Policies Defined on Time Intervals*

We have so far discussed consumption policies assuming that we consider atomic events (in an input stream). As atomic events happen in time points, it is possible to establish a *total order* of their occurrences. Consequently it is easy to answer which event instance, out of two, happened more recently. When we deal with complex events ($T_1 \neq T_2$), a total order is not always possible. This subsection provides possible options in defining consumption policies in such a case.

Recent Policy. Let us consider the following sequence of input events: A(1,30), A(15,30), B(35,50). In our example rule (6) (from Section 3.1), the question now is which instance of A is more *recent*, A(1,30) or A(15,30)? In our opinion, this question depends on a particular application domain. There are three possible options. First, an event detected on a *longer event duration* is selected to be the recent one (i.e., A(1,30)). This option is suitable when aggregation functions (for example, sum, average and so forth) are applied along time windows. Hence, events detected on longer durations possibly reflect more accurate results. The second option is to choose an event with a *shorter duration* (i.e., A(15,30)). This preference is suitable when indeed more recent events are desired. For example, we are interested in data (carried by events) that are as up to date as possible. Finally, the third possibility is to pick up an event instance based on *data value selection* i.e., non-temporal properties. For instance, events ending at the same time, $\texttt{A}(1, 30, X, Vol = 1000)$ and $\texttt{A}(15, 30, X, Vol = 10000)$, are selected based on an attribute value (e.g., greater volume Vol).

We implement these three cases with rules (28)-(30). When an A occurs, there is a policy check performed. In rule (28), for two events with the same ending (i.e., $\texttt{A}(T_1, T_3)$ and $\texttt{A}(T_2, T_3)$) we make sure that one with a longer path ($T_1 > T_2$) is selected. In rule (29), we replace goals if the time condition is opposite ($T_1 < T_2$). Finally, in data value (or attribute value) selection, we distinguish based on a chosen attribute (e.g., $Vol_1 > Vol_2$).

[1] Note that goals can still be deleted if their time window expires.

$$\begin{aligned}\texttt{event\_trigger}(\texttt{A}(T_1,T_3,Vol_1)) :- \; & \texttt{goal}(\_,\texttt{A}(T_2,T_3,\_,\_),\_), T_1 > T_2, \\ & \texttt{assert}(\texttt{goal}(\_,\texttt{A}(T3,T4,\_,\_),\_)), \\ & \texttt{assert}(\texttt{goal}(\_,\texttt{A}(T1,T3,Vol_1),\_)).\end{aligned} \tag{28}$$

$$\begin{aligned}\texttt{event\_trigger}(\texttt{A}(T_1,T_3,Vol_1)) :- \; & \texttt{goal}(\_,\texttt{A}(T_2,T_3,\_,\_),\_), T_1 < T_2, \\ & \texttt{retract}(\texttt{goal}(\_,\texttt{A}(T3,T4,\_,\_),\_)), \\ & \texttt{assert}(\texttt{goal}(\_,\texttt{A}(T1,T3,Vol_1),\_)).\end{aligned} \tag{29}$$

$$\begin{aligned}\texttt{event\_trigger}(\texttt{A}(T_1,T_3,Vol_1)) :- \; & \texttt{goal}(\_,\texttt{A}(T_2,T_3,Vol_2,\_),\_), Vol_1 > Vol_2, \\ & \texttt{retract}(\texttt{goal}(\_,\texttt{A}(T3,T4,Vol_2),\_)), \\ & \texttt{assert}(\texttt{goal}(\_,\texttt{A}(T1,T3,Vol_1),\_)).\end{aligned} \tag{30}$$

Policy rules (28)-(30) are fired before inserting a new goal. It is worth noting that such an update of a goal is performed incrementally. We pay an additional price for forcing a particular consumption policy. However, the policy rules (28)-(30) are rather simple rules. In return, they ensure that no more than one goal with the same timestamp (with respect to a certain policy) is kept in memory during the processing. Therefore the policy rules enable a better *memory management* in our framework.

Chronological Policy. The main principle in the implementation of this policy is the same as in the recent policy. The only difference is that now we consider the same start and the different ending in multiple event occurrences ($\texttt{A}(T_1,T_2)$,$\texttt{A}(T_1,T_3)$). To implement this policy, rule (28) will now contain the time condition from rule (29), and vice versa. Rule (30) remains unchanged, as well as *unrestricted policy* (which is the same as for the case with atomic events, see Subsection 4.1).

5 Deductive Reasoning in Complex Event Processing

So far, we have described a general framework for event recognition with rules. In this section, we explore an additional feature, namely *reasoning capability*. This feature is enabled by the *logic* nature of our approach.

Current CEP systems Agrawal *et al.* (2008); Mei & Madden (2009); Barga *et al.* (2007); Arasu *et al.* (2006); Krämer & Seeger (2009); Chandrasekaran *et al.* (2003) provide on-the-fly analysis of data streams, but cannot combine streams with evolving *knowledge*, and they cannot perform *reasoning* tasks. On-the-fly stream analysis enables real-time decisions and actions. Often, however, data streams are not sufficient to provide such an analysis. How likely is that *critical* real-time decisions are taken merely on fact that few events happened with certain temporal constellation if we know nothing about the *semantic* relations and the *context* of these events. *Background knowledge* is necessary to provide the *context* in which streaming data are to be interpreted. For example, two events that happened within the last ten seconds may constitute a complex event only if there exist certain *semantic* relation between them. The quality of an analytical service that processes on-line data and events greatly depends on a local *knowledge base* – usually containing the facts of a particular domain, expert knowledge etc. – which can be employed for a more intelligent processing of information than if it is merely based on the *temporal* relationship between occurring events.

Our framework addresses this issue and offers the possibility to express domain knowledge as a set of *facts* and *rules*. Further on, the fretwork is capable to perform *deductive reasoning* over that knowledge combined with the streaming data, i.e., to perform *stream reasoning*.

5.1 *Complex Events and Transitive Closure Rules*

To give the reader a feeling how deductive rules can be used in combination with the rest of the ETALIS framework, we present a set of illustrative examples.

Let us observe a common situation in aviation, related to detection of *clear air turbulence* (CAT) on jet streams. Jet streams are important for aviation, as flight time can be dramatically affected by either flying with the flow or against the flow of a jet stream. Clear air turbulence, a potential hazard to aircraft passenger safety, often is found in a jet stream's vicinity. In the following example, we define `JetStreamWarning` event as a dangerous situation whenever a clear air turbulence (denoted as `CAT` event) is followed by the `Airplane` position event.

$$\texttt{JetStreamWarning}(Loc_1, Loc_2) \leftarrow \big(\texttt{CAT}(Loc_1) \text{ AND } \texttt{Airplane}(Loc_2)\big).5hours \\ \text{WHERE } \texttt{jetLink}(Loc_2, Loc_1). \tag{31}$$

$$\begin{array}{l} \texttt{jetLink}(X,Y) :- \texttt{linked}(X,Y). \\ \texttt{jetLink}(X,Z) :- \texttt{linked}(X,Y), \texttt{jetLink}(Y,Z). \end{array} \tag{32}$$

$$\begin{array}{l} \texttt{linked(1,2).} \\ \texttt{linked(2,3).} \\ \texttt{linked(3,4).} \\ .... \end{array} \tag{33}$$

To make sure that the `CAT` affects the observed jet stream, we deploy transitive closure rules (32). The rules span over a set of facts (33), defining the jet stream as a set of connected points. Since both, the `CAT` and the `Airplane`, change their positions, the rules check whether they belong to the same jet stream. Note that the check is successful if position of the `CAT` is in front of the current position of the `Airplane`.

Transitive closure rules (32) are deductive rules[1], and together with the `linked` relation (33), they enable us to perform on-the-fly *reasoning* (i.e., to examine whether a new clear air turbulence is dangerous with respect to an observed jet stream or not).

According to US National Business Aviation Association[2] (NBAA) air routes are *dynamic*. This means that they can be modified as needed in order to take advantage of favorable winds, which change on a daily basis. Hence a solution based only on querying of jet stream static points would not be optimal. Concluding this example, we note that since facts (33) are dynamic, an occurrence of a new `CAT` is not known in advance, and the airplane position is changing too, our approach to combine CEP with deductive reasoning is an appropriate approach for on-the-fly jet stream monitoring.

5.2 *Rule-based Event Classification and Filtering*

As a next example we will demonstrate the use of background rules for event *classification* and *filtering*. Let us extend the `HeatIndex` event pattern (see pattern (4) from Section 2.1) to automatically generate shade values of the `HeatIndex`. Whenever there is a new sensor reading `HeatIndex`, we want the system to generate a human readable note (e.g., caution, danger etc.). Additionally, the system needs to generate an area for which the note applies.

$$\texttt{HeatIndexEffect}(Note, Area) \leftarrow \texttt{HeatIndex}(Loc, Index) \\ \text{WHERE } \{\texttt{shadeValuesRule}(Index, Note), \\ \texttt{areaRule}(Loc, Area)\}. \tag{34}$$

The following rules (written in Prolog syntax) serve to filter out the heat $Index$ values smaller than 80, and classify the ones remaining into four categories: *'Caution'* (between 80 and 90); *'ExtremeCaution'* (between 90

[1]The example could be extended to deal with `CAT` areas (instead of points). Also, by introducing an ID to a jet stream, we could monitor more than one jet stream at the same time.

[2]NBAA: `http://www.nbaa.org/ops/airspace/issues/wind-routes/`

Table 1. Namespace abbreviations.

Prefix	URI	Description
wt	`http://knoesis.wright.edu/ssw/page/ont/weather.owl#`	Weather ontology
xsd	`http://www.w3.org/2001/XMLSchema#`	XML Schema Vocabulary
rdf	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`	RDF Vocabulary
rdfs	`http://www.w3.org/2000/01/rdf-schema#`	RDF Schema Vocabulary

and 105); *'Danger'* (between 105 and 130); and *'ExtremeDanger'* for values greater than or equal to 130.

$$\begin{aligned} \texttt{shadeValuesRule}(Index,'Caution') &:- 80 =< Index, Index < 90, !. \\ \texttt{shadeValuesRule}(Index,'ExtremeCaution') &:- 90 =< Index, Index < 105, !. \\ \texttt{shadeValuesRule}(Index,'Danger') &:- 105 =< Index, Index < 130, !. \\ \texttt{shadeValuesRule}(Index,'ExtremeDanger') &:- 130 =< Index, !. \end{aligned} \tag{35}$$

Further on, background knowledge specified by rules (36) can be used to focus on certain monitoring areas, and to transform GPS coordinates into those areas.

$$\begin{aligned} \texttt{areaRule}(\texttt{loc}('N', X,'W', Y),'Area'_1) &:- 4042 < X, X < 4049, 7358 < Y, Y < 7370, !. \\ &.... \\ \texttt{areaRule}(\texttt{loc}('N', X,'W', Y),'Area'_n) &:- 4034 < X, X < 4040, 7368 < Y, Y < 7399, !. \end{aligned} \tag{36}$$

5.3 *Deductive Reasoning and Complex Event Processing*

In the following, we show how deductive reasoning can be combined with event processing. Assume we need to detect a complex event `EnhancedFire`, which arises when in the area of an active fire, there is an additional `WeatherObservation`. Some weather observations have significant influence on actions taken with respect to an ongoing wildfire. For example, strong wind may be particularly dangerous for an active fire area. The following pattern specifies such a situation.

$$\begin{aligned} \texttt{EnhancedFire}(Loc) \leftarrow \big(&\texttt{ActiveFire}(Loc) \text{ AND} \\ &\texttt{WeatherObservation}(Loc, Observ)\big).3hours \\ &\text{WHERE} \\ &\big(\texttt{rdfs:subClassOf}(Observ,'\texttt{wt:WindObservation}')\big). \end{aligned} \tag{37}$$

Let us now define background knowledge about weather observations. We use the Resource Description Framework (RDF) Klyne & Carroll (10 February 2004) as a common format for expressing graph-structured data. RDF Schema (RDFS) Brickley *et al.* (10 February 2004) adds additional expressivity in order to support the design of simple vocabularies also encoded in RDF. The following namespace definitions are used for brevity.

We can define $WindObservation$ as a subclass of $WeatherObservation$[1], and further to define $Diablo$ and $Sundowner$ as two kinds of $WindObservation$.

```
wt:WindObservation  rdfs:subClassOf  wt:WeatherObservation .
wt:Diablo           rdfs:subClassOf  wt:WindObservation .
wt:Sundowner        rdfs:subClassOf  wt:WindObservation .
```

We assume that there exist various types of weather observations defined in the background knowledgebase. For example, `Observ_1` is a specific type of `wt:Diablo`, and in general there exist more than one instance for each type.

[1]According to Weather Ontology from Patni *et al.* (2010), $WeatherObservation$ is a subclass of $Observation$, and there exist various types of $WeatherObservation$ such as $PressureObservation$, $TemperatureObservation$, $RadiationObservation$, and $WindObservation$.

```
Observ_1
  rdf:type         wt:Diablo ;
  wt:speed         "60"^^xsd:int ;
  wt:temperature   "30"^^xsd:int ;
  wt:region        "California"^^xsd:string .

Observ_2
  rdf:type         wt:Sundowner ;
  wt:speed         "40"^^xsd:int ;
  wt:temperature   "100"^^xsd:int ;
  wt:region        "California"^^xsd:string .
```

Finally, let us use a subclass relation rule, stating that `A` is an instance of Y if X is subclass of Y and `A` is an instance of X (see rule (38)).

$$\texttt{rdf:type}(\texttt{A}, Y) :- \texttt{rdfs:subClassOf}(X, Y), \texttt{rdf:type}(\texttt{A}, X). \tag{38}$$

Now, if events `ActiveFire` and `WeatherObservation` both occur within 3 hours, the system needs to check the type of `WeatherObservation`. `EnhancedFire` pattern will be matched if `WeatherObservation` is of type wind. Let us assume that `WeatherObservation` carries `Observ_1` as a type. Retrieving the RDF description for `Observ_1`, the system has information that `Observ_1` is of type `wt:Diablo`. Then by using rule (38), the system will *deduce* that `wt:Diablo` is a $WindObservation$ and it will finally trigger `EnhancedFire` pattern.

Moreover, the pattern will be also detected if `WeatherObservation` was detected having `Observ_2` as a type (since `Observ_2` is of type `wt:Sundowner`, and the latest is a `WindObservation`).

In this example we have arguably demonstrated the power of our formalism which combines event processing and deductive reasoning. In order to detect complex situations, events need to satisfy *temporal* constellations (e.g., both events need to happen within three hours), as well as, *semantic* relations (e.g., data carried by events need to satisfy , for example, class/subclass or other domain specific relations).

6 Evaluation Results

As a proof of concept, we have provided an open-source implementation of the ETALIS Language for Events. The system, called ETALIS[1], is based on the execution model of the language described in Section 3, i.e., it is established on *goal-directed* EDBCR and decomposition of complex event patterns into *intermediate events*, i.e., *goals*). ETALIS automatically compiles the user-defined complex event descriptions into EDBC (Prolog) rules. A user may additionally specify deductive rules as a background knowledge (see Section 5). These rules can be directly written in Prolog, or alternatively, a user may specify background knowledge in form of RDFS ontologies.

In this section we present experimental results, obtained with the ETALIS system. All tests were carried out on a workstation with Intel Core Quad CPU Q9400 2,66GHz, 8GB of RAM, running Windows Vista x64. ETALIS was run on SWI Prolog[2] engine.

To demonstrate the usefulness of our framework in practice, we have developed an application using real sensor data. The application is connected to a sensor network called MesoWest[3], which provides measurements of environmental phenomena (e.g., weather observations such as wind, temperature, humidity, precipitation, visibility and so forth). The goal of our application is to demonstrate how simple sensor readings can be analyzed on the fly, and hence used to detect more complex weather observations (e.g., blizzards, hurricanes etc.). Further on, we demonstrate how sensor data can be integrated over time and geographical space. For instance, observations of a blizzard, detected by few nearby sensors within a certain time frame identify an affected blizzard area. A blizzard warning may be issued as soon as the application detects such a situation. Moreover, the application utilizes GeoNames

[1] ETALIS: `http://code.google.com/p/etalis/`
[2] SWI Prolog: `http://www.swi-prolog.org/`
[3] MesoWest: `http://mesowest.utah.edu/`

semantic information[4] to identify all important geographic locations (e.g., schools, hospitals, motorways, airports, tunnels, railroads etc.) affected by that weather observation, so that further (security) actions can be taken in case of an emergency.

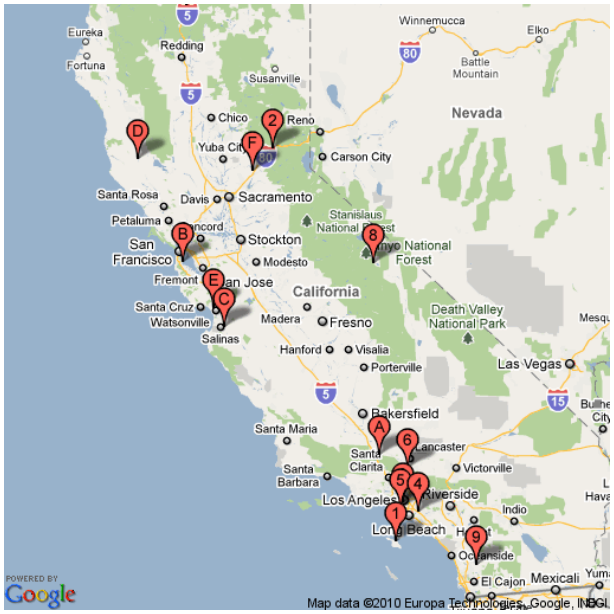

Figure 4. Sensor Location Map

MesoWest is a cooperative project between researchers at the University of Utah, forecasters at the Salt Lake City National Weather Service Office, the NWS Western Region Headquarters, and many other participating agencies, universities, and commercial firms. The network includes around 20,000 weather stations in the United States. For this experiment we have selected 15 sensors from California (as density of available sensors in California is high). Locations of the selected sensors are indicated by red markers in Figure 4 (enumerated with hexadecimal numbers: 1,2,3,...,F). Experiments are conducted on sensor data starting from 2007-12-31 until 2010-20-11. In our running example, the goal was to detect a blizzard from MesoWest streaming data. According to National Oceanic and Atmospheric Administration[1] (NOAA), a blizzard occurs when the following conditions prevail over a period of 3 hours or longer: high wind speed (35 miles an hour or greater); considerable falling snow; and low visibility (less than 1/4 mile). The following event pattern (39) is used to detect a blizzard settling situation.

$$
\begin{aligned}
&\texttt{BlizzardSettling}(T_1, T_1, ID, 1) \leftarrow \\
&\qquad\qquad \texttt{sensor}(T_1, ID, Temp, Wind, WtherCond, Visib) \\
&\qquad\qquad \text{WHERE } (Wind > 35, WtherCond ==' snow', Visib < 0.25). \\
&\texttt{BlizzardSettling}(T_1, T_3, ID, C+1) \leftarrow \\
&\qquad\qquad \texttt{BlizzardSettling}(T_1, T_2, ID, C) \text{ SEQ} \\
&\qquad\qquad \texttt{sensor}(T_3, ID, Temp, Wind, WtherCond, Visib) \\
&\qquad\qquad \text{WHERE } (Wind > 35, WtherCond ==' snow', Visib < 0.25).
\end{aligned}
\tag{39}
$$

The first rule operates on `sensor` reading events that carry a timestamp T[2], as well as, a number of other parameters: a weather station ID, the current temperature $Temp$, wind speed $Wind$, weather condition $WtherCond$, and visibility measure $Visib$. The rule detects a sensor reading which satisfies the blizzard condition, and triggers a `BlizzardSettling` event. This event will start the second, iterative rule in (39). Every new sensor reading

[4] GeoNames Ontolgy: `http://www.geonames.org/ontology`
[1] NOAA: `http://www.noaa.gov/`
[2] Since `sensor` is an atomic event, it is defined on the time point (not an interval $[T_1, T_2]$)

Table 2. Complex Events from Live Sensor Data.

Sensor ID	No. of Events	Pattern (39)	Pattern (40)
KAVX	38156	2995	161
KBLU	1998	2327	157
KCQT	1164	0	0
KFUL	29341	28	0
KHHR	30118	31	0
KMER	28999	281	16
KMHS	1364	0	0
KMMH	36783	161	2
KRNM	1307	148	8
KSDB	1464	1167	89
KSFO	1277	241	12
KSNS	31958	794	32
KUKI	1267	52	2
KWVI	34132	420	28

(matching the same ID, and passing the filter condition) will trigger a new `BlizzardSettling` event and increase a counter `C`. The counter is used to implement situation in which the blizzard conditions *prevail* over a period of time. This means that, in order to detect a blizzard, not every `sensor` reading needs to satisfy the conditions. Instead, it is enough to detect sufficiently many of satisfying readings. Since in average MesoWest sensor updates its readings every 30 min, 4 events would be sufficient to satisfy this condition (as 6 events in total happen within 3 hours). Note that with each next iteration `BlizzardSettling` event will have a longer time interval (T_1, T_3) which the event is defined on. Finally, if the interval gets at least three hours long (with at least 4 iterations passed), rule (40) will detect a `BlizzardWarning` event. To ensure the upper interval limit (e.g., between 3 and 6 hours) in settling a `BlizzardWarning`, we can set a garbage collection (see Section 3.8).

$$\begin{aligned}&\texttt{BlizzardWarning}(T_1, T_2, ID) \leftarrow \\ &\qquad \texttt{BlizzardSettling}(T_1, T_2, ID, C) \text{ WHERE } (C \geq 4, T_2 - T_1 \geq 3hours).\end{aligned} \tag{40}$$

Table 2 presents evaluation results that we have obtained from MesWest sensor data. The first two columns show the sensor ID and the number of events produced by the corresponding sensor (in the period from 2007-12-31 until 2010-20-11). The third and fourth columns show the number of complex events, produced by evaluating pattern (40) and pattern (42), respectively. To increase the number of complex detections we have weakened the blizzard definition. In particular, we have removed the requirement for the considerable falling snow, and have decreased the wind speed condition to 15mph or greater (instead of 35mph).

A `BlizzardWarning` event is detected from data provided by a single sensor. Very often to monitor development of a blizzard (or other phenomena) in an area, it is necessary to *integrate* different observations from multiple sensors in that area. To analyze the observations over a certain geographical space, the system will require awareness of sensor locations in that space. Real-time integration of sensor observations from different geographic locations is not the only challenge. The heterogeneity of data provided by various sensors pose a big challenge too. For example, not all sensors provide the same measurements (e.g., some weather stations measure the wind speed, and other do not); measurements from various sensors are not provided in the same format, metric unit, or precision.

To overcome these and similar challenges, we utilize a domain specific *ontology* as a single view over the whole sensors network. Such an ontology for the MesoWest sensor network is available from Patni *et al.* (2010). This sensor ontology, for example, defines concepts such as $Observation$ (specified as an act of observing a property or phenomenon, with the goal of producing an estimate of the value of the property), and $Feature$ (defined as an abstraction of real world phenomenon). Further on, it defines major properties of an observation such as a feature of interest ($featureOfInterest$), observed property ($observedProperty$), sampling time ($samplingTime$) and so forth.

The work in Patni *et al.* (2010) also provides an RDF dataset containing expressive descriptions of about 20000 weather stations across the United States. On average, there are five sensors per weather station measuring phenomena such as temperature, visibility, precipitation, pressure, wind speed, humidity, see Section 9.2 in Appendix for description of one such a weather station. The description also contains the sensor location (altitude, latitude, and longitude). In our application we utilize this information in order to eventually detect a blizzard area (once a station detects a blizzard).

The first rule in the complex event pattern (41) is triggered whenever a `BlizzardWarning` event occurs. To

Table 3. Computation for pattern (42) from live sensor data.

Area	Start [date/time]	End [date/time]	Iterations
KSDB, KRNM	2008-01-14 02:00	2008-01-15 12:30	1
KAUN, KBLU	2008-02-01 10:35	2008-02-01 13:30	1
KBLU, KWVI	2008-02-23 09:53	2008-02-24 03:00	5
KWVI, KBLU	2008-02-24 07:47	2008-02-24 13:07	1
KBLU, KWVI	2008-02-24 07:54	2008-02-25 02:22	5
KSDB, KAVX	2010-01-03 02:52	2010-01-03 07:02	7
KAVX, KSDB	2010-01-03 09:52	2010-01-04 07:02	1
KSDB, KAVX	2010-01-05 03:22	2010-01-05 09:02	3
KAVX, KSDB	2010-01-05 11:42	2010-01-06 08:02	1
KBLU, KSFO	2010-02-02 04:21	2010-02-02 12:58	1
KWVI, KSFO	2010-11-07 08:06	2010-11-07 08:22	1

evaluate the WHERE clause of the rule, ETALIS will access the background knowledge (i.e., the weather station RDF descriptions) and retrieve the sensor location. The first rule will also start an iteration, which is then continued by the second rule. This rule will fire an `AreaSettling` event every time there is a new `BlizzardWarning` in an area close to the initial `BlizzardWarning`. The distance is calculated by the `getDistance` predicated, and its implementation is provided as a background rule (see Section 9.1 in Appendix). In our example pattern we want to make sure that the distance is less than 300km or 186miles.

$$
\begin{aligned}
&\texttt{AreaSettling}(ID, ID, Lat, Lng) \leftarrow \\
&\qquad \texttt{BlizzardWarning}(T_1, T_2, ID) \\
&\qquad \text{WHERE } \texttt{getLatLong}(ID, Lat, Lng). \\
&\texttt{AreaSettling}(ID_1, ID_2, Lat_1, Lng_1) \leftarrow \\
&\qquad \texttt{AreaSettling}(ID_1, Lat_1, Lng_1) \text{ SEQ} \\
&\qquad \texttt{BlizzardWarning}(T_1, T_2, ID_2) \\
&\qquad \text{WHERE } \big(\texttt{getLatLong}(ID_2, Lat_2, Lng_2) \\
&\qquad\qquad \texttt{getDistance}(Lat_1, Lng_1, Lat_2, Lng_2, Dist), \\
&\qquad\qquad 0 < Dist < 300\big).
\end{aligned}
\tag{41}
$$

Finally, `BlizzardArea` event is detected when `AreaSettling` event occurs within the next 9 hours.

$$
\begin{aligned}
&\texttt{BlizzardArea}(T_1, T_2, ID) \leftarrow \\
&\qquad \big(\texttt{AreaSettling}(ID_1, ID_2, Lat_1, Lng_1)\big).9hours.
\end{aligned}
\tag{42}
$$

Table 3 shows results for the complex event pattern (42). ETALIS has detected different areas (with weather conditions as defined above) eleven times. The table presents which weather stations contributed to a particular area; a starting and ending date/time of an observation; and how many iterations were involved in creating that observation.

Figure 5 shows marked wind areas as calculated from patterns (39)-(42). Weather stations that have detected one or more blizzards (during the observed period) are marked yellow, and those that have not are small and blue. Finally, the wind areas are marked red.

In addition to location attributes (latitude, longitude, and elevation), the RDF dataset contains also links to locations in GeoNames[1] near a weather station, see Section in Appendix. The distance from a GeoNames location to a weather station is also provided. We use GeoNames as a worldwide geographical knowledge base. If a sensor detects a blizzard, GeoNames can provide all important geographic locations (e.g., schools, hospitals, motorways, airports etc.) within a certain radius from the sensor location, so that our application can issue an early warnings. Table 4, for example, shows GeoNames locations[2] for the KSFO weather station (i.e., the San Francisco International Airport).

[1] GeoNames: `http://www.geonames.org/`

[2] For space reasons we have listed only 7 locations. The complete list for this weather station contains 51 items.

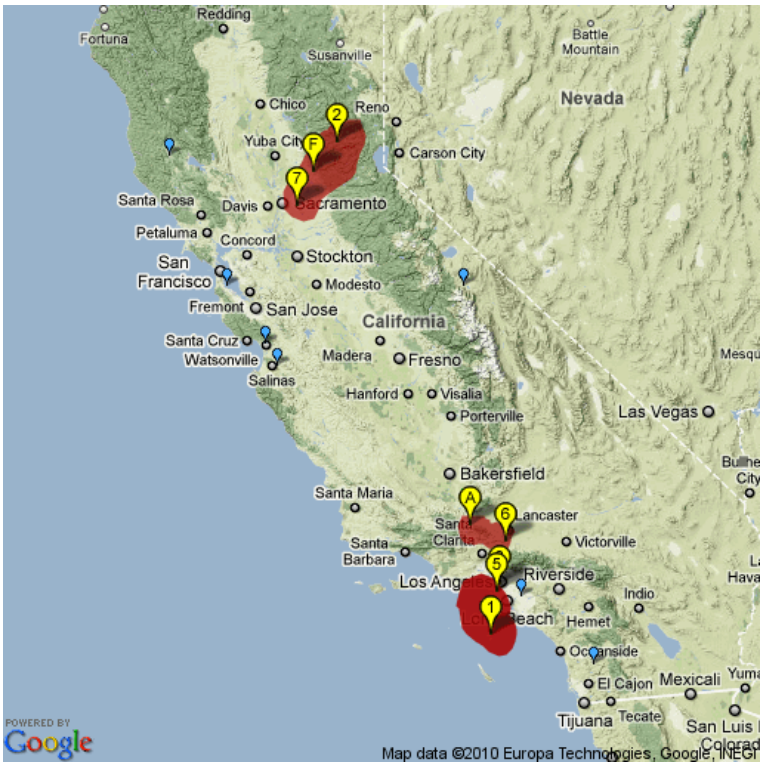

Figure 5. Sensor location map with marked wind areas

Table 4. GeoNames locations nearby KSFO weather station (SFO Airport).

GeoName ID	Location name	Latitude	Longitude
5394116	Seaplane Harbor	37.63216	-122.38164
7229706	San Mateo School	37.61196	-122.42842
7256223	Exit 5B	37.62861	-122.43167
7256211	Exit 41	37.59639	-122.41917
7256225	Exit 6A	37.63361	-122.40528
7256243	Exit 421	37.6025	-122.38028
7256245	Exit 423A	37.63111	-122.40278
...	...	...	...

Performance results for patterns (39)-(42) are presented in Figure 6. The throughput is obtained so that time between sensor readings is ignored. Different sensors produce data with different frequency. The goal of our performance test was to take into account ETALIS processing time, and to show the throughput accordingly. When only event processing time is considered (e.g., network latencies are ignored etc.), the throughput for patterns (39)-(42) are 24696, 37437, and 3900 events per second, respectively (see Figure 6 (a)).

We see also that the throughput for patterns (39) and (40) is significantly higher than for pattern (42). This pattern is however the most complex one, as for every `BlizzardWarning` event the system needs on-the-fly: to find the location from the RDF dataset; to compute the distance; and further to find out whether two sensors are close to each other. Taking into account that in average MesoWest sensors update information every half an hour, the throughput of 3900 events per second (or 7020000 events per 30 minutes) arguably demonstrates the use of our framework for real-time event recognition and reasoning, as this means that the same number of sensors can be handled by a single instance of our running system. Note that the complexity of the overall processing is high, i.e., additional knowledge bases are accessed and evaluated in the real-time during the detection of complex events, hence the achieved throughput is indeed promising.

Figure 6 (b) shows the memory consumption for patterns (39)-(42). We have calculated the overall memory consumption (i.e., not only memory picks). Pattern (42) has the lowest consumption (despite its complexity) and pattern (39) has the highest one. This comes as a consequence of the number of produced complex events. For example, from the KAVAX sensor stream, pattern (39) has been detected 2995 times and pattern (40) only 161

times, see Table 2. This stream has contributed to pattern (42) only four times. Hence although ETALIS needed to keep certain ontology data in memory (i.e., not only events), it still had a low memory consumption.

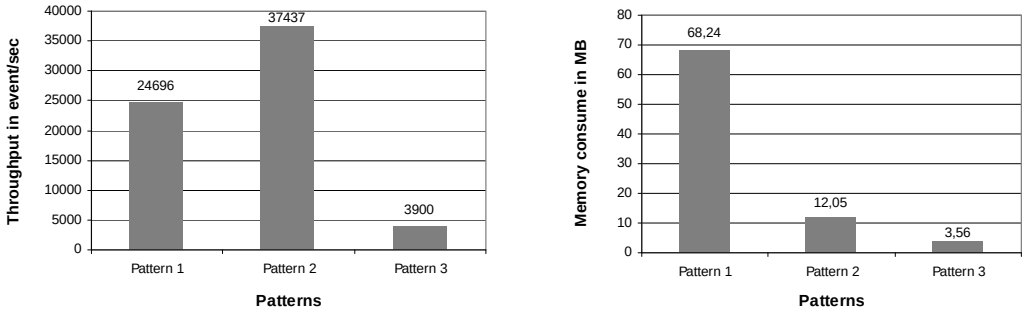

Figure 6. (a) Complex event throughput (b) Memory consumption

We have set up an on-line demo for the presented application[1] to continuously monitor live data provided by the MesoWest sensor network and detect weather observations in real-time.

7 Related Work

The work related to ours mainly fits into three areas: *Streaming Database* systems Agrawal *et al.* (2008); Mei & Madden (2009); Chandrasekaran *et al.* (2003); Cherniack *et al.* (2003), *temporal RDF* models Gutierrez *et al.* (2007); Perry *et al.* (2011); Tappolet & Bernstein (2009), and *Stream Reasoning* approaches Barbieri *et al.* (2010b,a); Walavalkar *et al.* (2008); Bolles *et al.* (2008).

7.1 *Streaming Databases*

Database approaches Agrawal *et al.* (2008); Mei & Madden (2009); Barga *et al.* (2007); Arasu *et al.* (2006); Krämer & Seeger (2009); Chandrasekaran *et al.* (2003); Cherniack *et al.* (2003) are based on languages with SQL-like syntaxes, and database execution models adapted to process streaming data. These approaches are dominant today due to their capability to handle large volumes of streaming data with low latency. As such database approaches are widely used in automated stock trading, logistic services, transaction management, business intelligence etc. However, they are not well suited for applications including structured data, ontologies, and other forms of knowledge bases where support for *semantic-based* event processing and *reasoning* is required.

7.2 *Rule-based CEP*

A lot of work Motakis & Zaniolo (1995); Artikis *et al.* (2010); Lausen *et al.* (1998); Paschke *et al.* (2010); Bry & Eckert (2007) in the area of rule-based CEP has been carried out, proposing various kinds of *logic rule*-based approaches to process complex events. As pointed out in Bry & Eckert (2007), rules can be effectively used for describing so-called "virtual" event patterns. There exist a number of other reasons to use rules: Rules serve as an abstraction mechanism and offer a higher-level event description. Also, rules allow for an easy extraction of different views of the same reactive system. Rules are suitable to mediate between the same events differently represented in various interacting reactive systems. Finally, rules can be used for reasoning about causal relationships between events.

One significance of rule based CEP systems Motakis & Zaniolo (1995); Lausen *et al.* (1998); Paschke *et al.* (2010) is the type of interaction they use. Namely, these systems interact based on a *request-response* paradigm. That is, given (and triggered by) a request, an inference engine will search for and respond with an answer. This

[1] available at: `http:\\etalis.fzi.de`

means that, for a given event pattern, an event inference engine needs to check if this pattern has been satisfied or not. The check is performed at the time when such a request is posed. If satisfied by the time when the request is processed, a complex event will be reported. If not, the pattern is not detected until the next time the same request is processed (though it can become satisfied in between the two checks, being undetected for the time being).

To overcome this issue, an expressive language XChange^{EQ} was proposed in Bry & Eckert (2007); Eckert (2008). The language features deductive and reactive rules for events, as well as event queries, event composition capabilities, event accumulation, possibilities to express temporal (and other) relationships between events and so forth. The language is accompanied with an incremental evaluation that avoids recomputing certain intermediate results every time a new event arrives. The authors use techniques from relational algebra based on incremental maintenance of materialized views Gupta & Mumick (1999) and finite differencing Eckert (2008).

The evaluation of XChange^{EQ} patterns is performed in *set at time* mode, i.e., events are processed in sets (relations). In contrast to that, our approach adheres to *event at time* processing mode, i.e., events are processed individually (similarly as in non-deterministic finite automata (NFA) approaches Gehani *et al.* (1992); Agrawal *et al.* (2008) or tree-based approaches Mei & Madden (2009)).

In summary, our approach is also based on deductive rules, and hence it benefits from the afore mentioned arguments. However our approach also differs from related work Motakis & Zaniolo (1995); Artikis *et al.* (2010); Lausen *et al.* (1998); Paschke *et al.* (2010); Bry & Eckert (2007). The main difference lays in the execution model (based on EDBCR). We have proposed an execution model, which is efficient with respect to real-time computation; event-driven computation; and also it is capable to handle different classes of expressive event patterns (e.g., supports temporal reasoning in complete Allen's Interval Algebra, iterative event patterns, various consumption policies etc.).

7.3 *Streaming over RDF data*

The Resource Description Framework (RDF) Klyne & Carroll (10 February 2004) has been widely used for expressing graph-structured data. The work in Gutierrez *et al.* (2007) introduced *time* as a new dimension in RDF graphs. The authors provided a semantics for temporal RDF graphs and a temporal query language for RDF, following concepts of temporal databases. Following this, various approaches have emerged, providing query languages for streaming RDF data. Some of them are surveyed in the following.

SPARQL-ST Perry *et al.* (2011) is an extension of SPARQL language[1] for complex spatial and temporal queries. The language and a corresponding implementation deal with temporal data (and possible reasoning about that data). However SPARQL-ST queries need to be triggered rather than being continuously active (they are not event-driven). The same argumentation also applies to other SPARQL approaches like Temporal SPARQL Tappolet & Bernstein (2009), stSPARQL Koubarakis & Kyzirakos (2010), and T-SPARQL Grandi (2010).

Continuous SPARQL (C-SPARQL) Barbieri *et al.* (2010a) is a language for continuous query processing with reasoning capabilities. It extends the SPARQL language by adding support for window and aggregation operations. C-SPARQL, however, does not provide *event processing* capabilities: after determining the set of currently valid RDF statements, classical reasoning on that RDF set is performed as if it were static. In particular, C-SPARQL offers no way of detecting occurrences of RDF triples in a specific temporal order. We strongly believe that additionally *temporal* relatedness between events (e.g., an event happened before another event) as defined in streaming database systems Agrawal *et al.* (2008); Chandrasekaran *et al.* (2003); Cherniack *et al.* (2003) is required to capture more complex patterns over RDF streaming data. Additionally, in C-SPARQL queries are divided into static and dynamic parts. The static part is evaluated by a RDF triple storage, while a stream processing engine evaluates the dynamic part of the query. In such settings, these two parts act as "black boxes" and C-SPARQL cannot take advantage of a query pre-processing and optimizations over the unified (static and dynamic) data space. We propose an approach based on logic rules where the both parts are handled in a uniform framework.

Finally, the work in Bolles *et al.* (2008) introduces Streaming SPARQL. The approach is built on temporal relational algebra, and the authors provide an algorithm to transform SPARQL queries to that algebra. Similarly as in Barbieri *et al.* (2010a), the approach is lacking event processing capabilities, i.e. detecting RDF triple sequences occurring in a specific order.

[1] SPARQL: `www.w3.org/TR/rdf-sparql-query/`

8 Conclusion

While in existing CEP approaches, complex events consist merely of simple (temporally situated) events, we argued that in *knowledge-rich* applications such complex events are not expressive enough to assess complex situations in real-time. We proposed a *logic-based* event processing, advocating a richer formalism for CEP. The formalism is capable not only to match patterns based on *temporal* relations among events, but also to evaluate *contextual knowledge*, and *reason* about their non-temporal *semantic* relations. Further, our contribution includes an execution model which detects complex events in a data-driven fashion (based on *goal-directed event-driven rules*). We have also provided an open-source implementation of our formalism, which allows for specification of complex events and their detection at occurrence time. The approach goes beyond existing event-driven systems by providing declarative semantics and an efficient logic-programming-based execution model that enables event-driven deductive *reasoning*, enabling a new generation of event-driven applications in Artificial Intelligence. We have developed a sensor network application in the weather observation domain and conducted a set of experiments to demonstrate efficiency and usefulness of our approach in a real-life scenario.

As the next steps, we will continue to investigate and exploit the advantages of our framework over non-logic-based CEP. In particular, we plan to investigate how a rule representation of complex events (in large pattern bases) may help in *verification* of event patterns (e.g., discovering patterns that can never be detected according to inconsistency problems). We also plan to utilize *machine learning* techniques to automatically generate both event patterns and the domain knowledge required for knowledge-based CEP (see Artikis *et al.* (2010), and XHAIL system Ray (2009)). Further, event *revision* is another area where logic reasoning will help in managing consequences when certain events are *retracted*. Likewise, *out-of-order* events can also be handled in a logic CEP framework. Event retraction and out-of-order events can be seen as facts being retracted or added late to an event processing knowledge base, respectively. Hence an inference system can be deployed to *reason* about logical consequences of retracted or events added late on the whole pattern detection process. *Dynamic* event pattern management (i.e., patterns are created or discarded on the fly when certain situations are detected) is another interesting topic where the logic approach may help to control event-driven computation.

9 Acknowledgments

This work was partially supported by the European Commission funded projects ALERT (FP7-258098) and PLAY (FP7-20495), as well as by the ExpresST project funded by the German Research Foundation (DFG). We thank Vesko Georgiev, Ahmed Khalil Hafsi, Jia Ding for their help in implementation and testing ETALIS.

References

Adaikkalavan, R. & Chakravarthy, S. (2006). SnoopIB: Interval-based event specification and detection for active databases. *Data Knowledge Engineering*, *59*(1), 139–165.

Agrawal, J., Diao, Y., Gyllstrom, D., & Immerman, N. (2008). Efficient pattern matching over event streams. In J. T.-L. Wang (Ed.), *Proceedings of the 28th ACM SIGMOD Conference*. New York, USA, SIGMOD'08, 147–160.

Alferes, J. J., Banti, F., & Brogi, A. (2006). An event-condition-action logic programming language. In M. Fisher, W. van der Hoek, B. Konev, & A. Lisitsa (Eds.), *Proceedings of the 10th European Conference on Logics in Artificial Intelligence*. Berlin,Heidelberg: Springer-Verlag, JELIA'06, 29–42.

Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM*, *26*, 832–843. URL `http://doi.acm.org/10.1145/182.358434`.

Anicic, D., Fodor, P., Rudolph, S., Stühmer, R., Stojanovic, N., & Studer, R. (2010). A rule-based language for complex event processing and reasoning. In P. Hitzler & T. Lukasiewicz (Eds.), *Proceedings of the 4th International Conference on Web Reasoning and Rule Systems*. Berlin,Heidelberg: Springer-Verlag, RR'10, 42–57.

Anicic, D., Fodor, P., Stühmer, R., & Stojanovic, N. (2009). Event-driven approach for logic-based complex event processing. In *Proceedings of the 12th IEEE International Conferences on Computational Science and Engineering (CSE'09)*. Washington, DC, USA: IEEE Computer Society, CSE'09, 56–63.

Arasu, A., Babu, S., & Widom, J. (2006). The CQL continuous query language: semantic foundations and query execution. *VLDB Journal*, *15*(2), 121–142. URL http://dx.doi.org/10.1007/s00778-004-0147-z.

Artikis, A., Paliouras, G., Portet, F., & Skarlatidis, A. (2010). Logic-based representation, reasoning and machine learning for event recognition. In J. Bacon, P. R. Pietzuch, J. Sventek, & U. Çetintemel (Eds.), *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems*. New York, NY, USA: ACM, DEBS'10, 282–293.

Barbieri, D. F., Braga, D., Ceri, S., & Grossniklaus, M. (2010a). An execution environment for C-SPARQL queries. In I. Manolescu, S. Spaccapietra, J. Teubner, M. Kitsuregawa, A. Leger, F. Naumann, A. Ailamaki, & F. Ozcan (Eds.), *Proceedings of the 13th International Conference on Extending Database Technology*. ACM, EDBT'10, 441–452.

Barbieri, D. F., Braga, D., Ceri, S., Valle, E. D., & Grossniklaus, M. (2010b). Incremental reasoning on streams and rich background knowledge. In L. Aroyo, G. Antoniou, E. Hyvönen, A. ten Teije, H. Stuckenschmidt, L. Cabral, & T. Tudorache (Eds.), *Proceedings of the 7th Extended Semantic Web Conference*. Berlin,Heidelberg: Springer-Verlag, ESWC'10, 1–15.

Barga, R. S., Goldstein, J., Ali, M. H., & Hong, M. (2007). Consistent streaming through time: A vision for event stream processing. In G. Weikum, J. Hellerstein, & M. Stonebraker (Eds.), *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research*. CIDR'07, 363–374.

Bolles, A., Grawunder, M., & Jacobi, J. (2008). Streaming SPARQL - Extending SPARQL to process data streams. In S. Bechhofer, M. Hauswirth, J. Hoffmann, & M. Koubarakis (Eds.), *Proceedings of the 5th European Semantic Web Conference*. Berlin,Heidelberg: Springer-Verlag, ESWC'08, 448–462.

Brickley, D., Guha, R., & McBride, B. (Eds.) (10 February 2004). *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C Recommendation. http://www.w3.org/TR/rdf-schema/.

Bry, F. & Eckert, M. (2007). Rule-based composite event queries: The language xchange$^{\text{eq}}$ and its semantics. In M. Marchiori, J. Z. Pan, & C. de Sainte Marie (Eds.), *Proceedings of the 1st International Conference on Web Reasoning and Rule Systems*. Berlin, Heidelberg: Springer-Verlag, RR'07, 16–30.

Chakravarthy, S. & Mishra, D. (1994). Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering*, *14*(1), 1–26.

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. (2003). TelegraphCQ: Continuous dataflow processing for an uncertain world. In G. Weikum, J. Hellerstein, & M. Stonebraker (Eds.), *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research*. www.crdrdb.org, CIDR'03.

Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Xing, Y., & Zdonik, S. B. (2003). Scalable distributed stream processing. In G. Weikum, J. Hellerstein, & M. Stonebraker (Eds.), *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research*. www.crdrdb.org, CIDR'03.

Dantsin, E., Eiter, T., Gottlob, G., & Voronkov, A. (2001). Complexity and expressive power of logic programming. *ACM Computing Surveys*, *33*(3), 374–425.

Eckert, M. (2008). *Complex Event Processing with XChangeEQ: Language Design, Formal Semantics and Incremental Evaluation for Querying Events*. Dissertation/Ph.D. thesis, Institute of Computer Science, LMU, Munich. URL http://www.pms.ifi.lmu.de/publikationen/#DISS_Michael.Eckert. PhD Thesis, Institute for Informatics, University of Munich, 2008.

Gehani, N. H., Jagadish, H. V., & Shmueli, O. (1992). Composite event specification in active databases: Model & implementation. In L.-Y. Yuan (Ed.), *Proceedings of the 18th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., VLDB'92, 327–338.

Grandi, F. (2010). T-SPARQL: a TSQL2-like temporal query language for RDF. In *International Workshop on on Querying Graph Structured Data*. CEUR (online), ADBIS'10, 21–30.

Gupta, A. & Mumick, I. S. (1999). Materialized views. Cambridge, MA, USA: MIT Press, chap. Maintenance of materialized views: problems, techniques, and applications. 145–157. URL http://dl.acm.org/citation.cfm?id=310709.310737.

Gutierrez, C., Hurtado, C. A., & Vaisman, A. A. (2007). Introducing time into RDF. *The IEEE Transactions on Knowledge and Data Engineering*, *19*(2), 207–218.

Haley, P. (1987). Data-driven backward chaining. In *International Joint Conferences on Artificial Intelligence*.

Milan, Italy.

Klyne, G. & Carroll, J. J. (Eds.) (10 February 2004). *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation. http://www.w3.org/TR/rdf-concepts/.

Koubarakis, M. & Kyzirakos, K. (2010). Modeling and querying metadata in the Semantic Sensor Web: The model stRDF and the query language stSPARQL. In L. Aroyo, G. Antoniou, E. Hyvönen, A. ten Teije, H. Stuckenschmidt, L. Cabral, & T. Tudorache (Eds.), *Proceedings of the 7th Extended Semantic Web Conference (ESWC'10)*. Berlin,Heidelberg: Springer-Verlag, Lecture Notes in Computer Science, 425–439.

Kowalski, R. & Sergot, M. (1986). A logic-based calculus of events. *New Generation Computing*, *4*, 67–95. URL http://portal.acm.org/citation.cfm?id=10030.10034.

Krämer, J. & Seeger, B. (2009). Semantics and implementation of continuous sliding window queries over data streams. *ACM Transactions on Database Systems*, *34*(1), 1–49.

Lausen, G., Ludäscher, B., & May, W. (1998). On active deductive databases: The statelog approach. In B. Freitag, H. Decker, M. Kifer, & A. Voronkov (Eds.), *Transactions and Change in Logic Databases*. Berlin,Heidelberg: Springer-Verlag, Lecture Notes in Computer Science, 69–106.

Luckham, D. (2002). *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Reading, MA, USA: Addison-Wesley.

Mei, Y. & Madden, S. (2009). Zstream: a cost-based query processor for adaptively detecting composite events. In *Proceedings of the 29th ACM SIGMOD Conference*. 193–206.

Miller, R. & Shanahan, M. (1999). The event calculus in classical logic - alternative axiomatisations. *Electronic Transactions on Artificial Intelligence*, *3*(A), 77–105.

Motakis, I. & Zaniolo, C. (1995). Composite temporal events in active database rules: A logic-oriented approach. In T. W. Ling, A. O. Mendelzon, & L. Vieille (Eds.), *Proceedings of the 4th International Conference on Deductive and Object-Oriented Databases*. London, UK: Springer-Verlag, DOOD '95, 19–37.

Paschke, A., Kozlenkov, A., & Boley, H. (2010). A homogeneous reaction rule language for complex event processing. *CoRR*, *abs/1008.0823*.

Patni, H., Sahoo, S. S., Henson, C., & Sheth, A. (2010). Provenance aware linked sensor data. In *2nd Workshop on Trust and Privacy on the Social and Semantic Web, Greece*. CEUR Workshop Proceedings.

Perry, M., Sheth, A. P., & Jain, P. (2011). SPARQL-ST: Extending SPARQL to support spatiotemporal queries. In N. Ashish & A. P. Sheth (Eds.), *Geospatial Semantics and the Semantic Web*. Berlin,Heidelberg: Springer-Verlag, 61–86.

Ray, O. (2009). Nonmonotonic abductive inductive learning. *Journal of Applied Logic*, *7*(3), 329–340.

Tappolet, J. & Bernstein, A. (2009). Applied temporal RDF: Efficient temporal querying of RDF data with SPARQL. In *Proceedings of the 6th European Semantic Web Conference*. Berlin, Heidelberg: Springer-Verlag, ESWC'09, 308–322.

Walavalkar, O., Joshi, A., Finin, T., & Yesha, Y. (2008). Streaming knowledge bases. In *International Workshop on Scalable Semantic Web Knowledge Base Systems*.

Yoneki, E. & Bacon, J. (2005). Unified semantics for event correlation over time and space in hybrid network environments. In R. Meersman, Z. Tari, M.-S. Hacid, J. Mylopoulos, B. Pernici, Ö. Babaoglu, H.-A. Jacobsen, J. P. Loyall, M. Kifer, & S. Spaccapietra (Eds.), *OTM Conferences*. Berlin,Heidelberg: Springer-Verlag, Lecture Notes in Computer Science, 366–384.

Appendix

9.1 *Distance Calculation*

Distance between two points, defined by their latitude and longitude ($Lat_1, Long_1, Lat_2, Long_2$, respectively) is calculated with Formula (43). The Earth Radius (ER) is constant, equals to 6378,137.

$$\texttt{getDistance} = \left\{ 2 \cdot \arcsin \left\{ \sqrt{\left[\sin \frac{rad(Lat_1)-rad(Lat_2)}{2}\right]^2 + \cos\left[rad(Lat_1)\right] \cdot \cos\left[rad(Lat_2)\right] \cdot \left[\sin \frac{rad(Long_1)-rad(Long_2)}{2}\right]^2} \right\} \right\} \cdot ER \quad (43)$$

The following rule (written in Prolog syntax) implements Formula (43). The rule was evaluated in the WHERE clause of the second rule in complex pattern (41), every time when `BlizzardWarning` event occurred (see experiments from Section 6).

```
getDistance(Lat1,Long1,Lat2,Long2,Distance) :-
    ER is 6378.137,
    getRad(Lat1,RadLat1),
    getRad(Long1,RadLong1),
    getRad(Lat2,RadLat2),
    getRad(Long2,RadLong2),
    A is RadLat1 - RadLat2,
    B is RadLong1 - RadLong2,
    TempA1 is A / 2,
    TempB1 is B / 2,
    SinA is sin(TempA1),
    SinB is sin(TempB1),
    TempA2 is SinA ** 2,
    TempB2 is SinB ** 2,
    CosA is cos(RadLat1),
    CosB is cos(RadLat2),
    Temp1 is CosA * CosB,
    Temp2 is Temp1 * TempB2,
    Temp3 is TempA2 + Temp2,
    Sqrt is sqrt(Temp3),
    Asin is asin(Sqrt),
    S is Asin * 2,
    Distance is S * ER.

getRad(Deg,Rad) :-
    Temp1 is Deg * pi,
    Rad is Temp1 / 180.
```

9.2 *Linked Sensor Data for Weather Stations*

An RDF dataset that describes sensors of KFSO weather station is shown below (see Patni *et al.* (2010)). In particular, the station measures phenomena such as temperature, dew point, humidity, visibility, wind direction, wind gust, and wind speed. The description also contains geo-location of the station, as well as, a GeoNames link with all known nearby locations.

```
<rdf:RDF xmlns="http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#"
    xmlns:log="http://www.w3.org/2000/10/swap/log#"
    xmlns:om-owl="http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:sens-obs="http://knoesis.wright.edu/ssw/"
    xmlns:weather="http://knoesis.wright.edu/ssw/ont/weather.owl#"
    xmlns:wgs84="http://www.w3.org/2003/01/geo/wgs84_pos#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#">

    <LocatedNearRel rdf:about="http://knoesis.wright.edu/ssw/LocatedNearRelKSFO">
        <distance rdf:datatype="http://www.w3.org/2001/XMLSchema#float">0.9813</distance>
        <hasLocation rdf:resource="http://sws.geonames.org/5391989/"/>
        <uom rdf:resource="http://knoesis.wright.edu/ssw/ont/weather.owl#miles"/>
    </LocatedNearRel>

    <System rdf:about="http://knoesis.wright.edu/ssw/System_KSFO">
        <ID>KSFO</ID>
        <hasLocatedNearRel rdf:resource="http://knoesis.wright.edu/ssw/LocatedNearRelKSFO"/>
        <hasSourceURI rdf:resource="http://mesowest.utah.edu/cgi-bin/droman/meso_base.cgi?stn=KSFO"/>
        <parameter rdf:resource="http://knoesis.wright.edu/ssw/ont/weather.owl#_AirTemperature"/>
        <parameter rdf:resource="http://knoesis.wright.edu/ssw/ont/weather.owl#_DewPoint"/>
```

```
        <parameter rdf:resource="http://knoesis.wright.edu/ssw/ont/weather.owl#_RelativeHumidity"/>
        <parameter rdf:resource="http://knoesis.wright.edu/ssw/ont/weather.owl#_Visibility"/>
        <parameter rdf:resource="http://knoesis.wright.edu/ssw/ont/weather.owl#_WindDirection"/>
        <parameter rdf:resource="http://knoesis.wright.edu/ssw/ont/weather.owl#_WindGust"/>
        <parameter rdf:resource="http://knoesis.wright.edu/ssw/ont/weather.owl#_WindSpeed"/>
        <processLocation rdf:resource="http://knoesis.wright.edu/ssw/point_KSFO"/>
    </System>

    <wgs84:Point rdf:about="http://knoesis.wright.edu/ssw/point_KSFO">
        <wgs84:alt rdf:datatype="http://www.w3.org/2001/XMLSchema#float">10</wgs84:alt>
        <wgs84:lat rdf:datatype="http://www.w3.org/2001/XMLSchema#float">37.61972</wgs84:lat>
        <wgs84:long rdf:datatype="http://www.w3.org/2001/XMLSchema#float">-122.36472</wgs84:long>
    </wgs84:Point>
</rdf:RDF>
```