A Rule-Based Language for Complex Event Processing and Reasoning

Darko Anicic[1] Paul Fodor[2] Sebastian Rudolph[3] Roland Stühmer[1] Nenad Stojanovic[1] Rudi Studer[1]

[1] FZI Research Center for Information Technology, University of Karlsruhe, 76131, Germany
[2] State University of New York at Stony Brook, USA
[3] Institut AIFB, University of Karlsruhe, Karlsruhe, Germany

Abstract. Complex Event Processing (CEP) is concerned with timely detection of complex events within multiple streams of atomic occurrences. It has useful applications in areas including financial services, mobile and sensor devices, click stream analysis etc. Numerous approaches in CEP have already been proposed in the literature. Event processing systems with a logic-based representation have attracted considerable attention as (among others reasons) they feature formal semantics and offer reasoning service. However logic-based approaches are not optimized for run-time event recognition (as they are mainly query-driven systems). In this paper, we present an expressive logic-based language for specifying and combining complex events. For this language we provide both a syntax as well as a formal declarative semantics. The language enables efficient run time event recognition and supports deductive reasoning. Execution model of the language is based on a compilation strategy into Prolog. We provide an implementation of the language, and present the performance results showing the competitiveness of our approach.

1 Introduction

Recently there has been made a significant paradigm shift toward *real-time* computing in the research, as well as, in industry. Databases and data warehouses are about looking what happened in the past. On the other hand, Complex Event Processing (CEP) is about processing real-time events, i.e., about detecting 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 signify a problem or an impending problem, a threshold, an opportunity, an information becoming available, a deviation etc. The general task of CEP can be described as follows. Within some dynamic setting, events take place. Those *atomic events* are instantaneous, i.e., they happen at one specific point in time and have a duration of zero. Notifications about these occurred events together with their timestamps and possibly further associated data (such as involved entities, numerical parameters of the event, or provenance data) enter the CEP system in the order of their occurrence.

The CEP system further features a set of *complex event descriptions*, by means of which *complex events* can be specified as temporal constellations of atomic events. The complex events thus defined can in turn be used to compose even more complex events and so forth. As opposed to atomic events, those complex events are not considered instantaneous but are endowed with a time *interval* denoting when the event started and when it ended.

The purpose of the CEP system is now to detect complex events within this input stream of atomic events. That is, the system is supposed to notify that the occurrence of a certain complex event has been detected, as soon as the system is notified of an atomic event that completes a sequence which makes up the complex event due to the complex event description. This notification may be accompanied by additional information composed from the atomic events' data. As a consequence of this detection (and depending on the associated data), responding actions can be taken, yet this is outside the scope of this paper.

Our approach for CEP is based on declarative (logic) rules. It has been shown elsewhere [13, 16, 15, 2, 7, 12, 17] that logic-based approaches for event processing have various advantages. First, they are *expressive* enough and convenient to represent diverse complex event patterns. They come with a *formal declarative* semantics. Moreover declarative rules are free of side-effects (e.g. confluence problem). Second, integration of *query processing* with event processing is easy and natural (e.g. processing of *recursive* queries). Third, our experience with use of logic rules in implementation of the main constructs in CEP as well as in providing extensibility of a CEP system is very positive and encouraging (e.g. number of code lines in logic programming is significantly smaller than in procedural programming). Ultimately, a logic-based event model allows for *reasoning* over events, their relationships, entire state, and possible *contextual knowledge* available for a particular domain. Reasoning about *temporal* knowledge (i.e., events) and *static* or *evolving* knowledge (i.e., facts, rules and ontologies) is a feature beyond of the state-of-the-art in CEP [1, 6, 14].

Apart from the above mentioned strengths, event processing systems [13, 16, 15, 12, 17] based on various logic formalism have some shortcomings too. One significant shortcoming is *data* or *event-driven* computation. Deductive systems are rather suited for a *request-response* computation. That is, for given a *request*, an inference engine will evaluate available knowledge (i.e. rules and facts) and *respond* with an answer. This means that the event inference engine needs to check if this pattern can be deduced 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). Contrary to this, event processing demands *data-driven* computation (as handled by various approaches such as non-deterministic finite automata (NFA) [1], Petri Nets [11], RETE algorithm [10] etc.). Unfortunately approaches grounded on NFA and Petri Nets do not feature reasoning capabilities; and RETE based approaches may be integrated with deductive rules [4] but have difficulties to handle aggregates over event streams, and to implement different event consumption policies [8].

[17] follows the mentioned request-response (or so called *query-driven*[4]) approach. It proposes to define queries that are processed repetitively at given intervals, e.g. every

[4] If a request is represented as a query (what is a usual case).

10 seconds, trying to discover new events. However, generally events are not periodic or if so might have differing periods and nevertheless complex events should be detected as soon as they occur (not in a predefined time window). To overcome this issue, in [7], an incremental evaluation was proposed. The approach is aimed at avoiding redundant computations (particularly re-computation of joins) every time a new event arrives. The authors suggest to utilize relational algebra evaluation techniques such as incremental maintenance of materialized views.

Our language for CEP, ETALIS, is developed to close the gap between event-driven and logic-based systems. We present a *rule-based language* with a clear syntax and a declarative formal semantics. The language is powerful enough to effectively *express* and *evaluate* all thirteen Allen's temporal relationships [3]. Unlike other non-logic-based CEP languages [1, 11], our language features *inference* capabilities; and unlike other logic-based approaches, it has a different execution model that compiles complex event patterns into logic rules and enables timely, *event-driven* detection of complex events. Finally unlike RETE-based approaches, recursive rules of our language enable processing of unbound event streams and applying aggregation functions on them; yet recursive rules are out of scope of this paper. The contribution also includes an *implementation* of the language, and experimental results of our evaluation.

2 Rule-Based Language for Event Processing and Reasoning

2.1 Syntax

In this section we present the formal syntax of the our language for event processing, while in the remaining sections of the paper, we will gradually introduce other aspects of the language (i.e. the declarative semantics and run-time detection of complex events as well as the performance of a prototype based on the language[5]).

The syntax of the our language allows for the description of *time* and *events*. We represent time instants as well as durations as nonnegative rational numbers $q \in \mathbb{Q}^+$. Events can be atomic or complex. An *atomic event* refers to an instantaneous occurrence of interest. Atomic events are expressed as ground atoms (i.e. predicates followed by arguments which are terms not containing variables). Intuitively, the arguments of a ground atom describing an atomic event denote information items (i.e. event data) that provide additional information about that event.

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

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

Thereby, $\texttt{pr}$ is a predicate name with arity n, t_i denote terms, t is a term of type boolean, q is a nonnegative rational number, and BIN is one of the binary operators SEQ,

[5] Our prototype, ETALIS, is an open source project, available at: `http://code.google.com/p/etalis`

AND, PAR, OR, EQUALS, MEETS, DURING, STARTS, or FINISHES. As a side condition, in every expression p WHERE t, all variables occurring in t must also occur in pattern p.

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

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

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

After introducing the formal syntax of our formalism, we will give some examples to provide some intuitive understanding before proceeding with the formal semantics in the next section. Adhering to a stock market scenario, one instantaneous event (not requiring further specification) might be $\mathtt{market\_closes}()$. Other events with additional information associated via arguments would be $\mathtt{bankrupt}(lehman)$ or $\mathtt{buys}(citigroup, wachovia)$. Within patterns, variables instead of constants may occur as arguments, whence we can write $\mathtt{bankrupt}(X)$ as a pattern matching all bankruptcy events irrespective of the victim. "Artificial" time-point events can be defined by just providing the according timestamp.

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

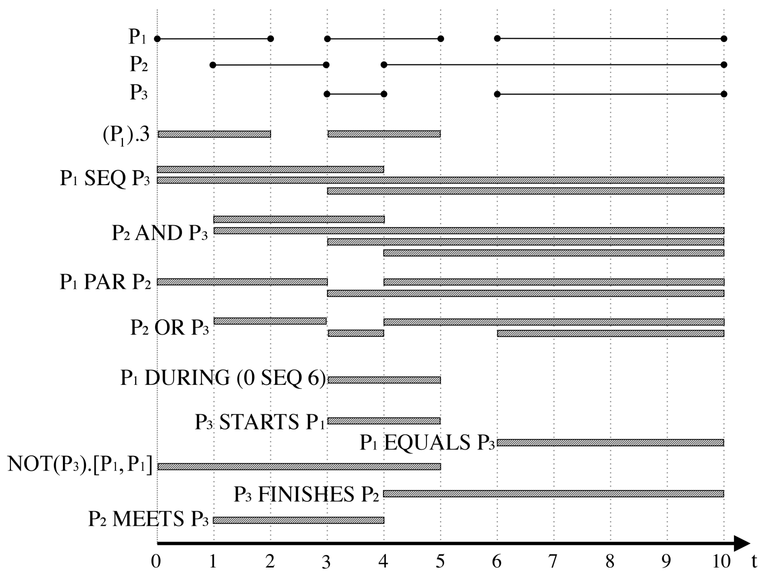

Fig. 1. Language for Event Processing - Composition Operators

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.
- 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; thereby 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_2 and P_3 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 $(0 \text{ SEQ } 6)$ 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_1 EQUALS P_3 is triggered when the two events occur exactly at the same time interval.
- 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 adopt one from [8].
- 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_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 [3]. The set can also be used for rich temporal reasoning.

2.2 Declarative Semantics

We define the declarative formal semantics of our language for event processing in a model-theoretic way.

Note that we assume a fixed interpretation of the occurring function symbols, i.e. for every function symbol f of arity n, we presume a predefined function $f^* : Con^n \to Con$. 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 extension of μ to terms defined in the usual way:

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

In addition to the set of rules $\mathcal{R}$, we fix an *event stream*. The event stream is formalized as a mapping $\epsilon : Ground \to 2^{\mathbb{Q}^+}$ from ground predicates into sets of nonnegative rational numbers. It thereby indicates at what time instants what elementary events occur. As a side condition, we require ϵ to be free of accumulation points, i.e. for every $q \in \mathbb{Q}^+$, the set $\{q' \in \mathbb{Q}^+ \mid q' < q \text{ and } q' \in \epsilon(g) \text{ for some } g \in Ground\}$ is finite.

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

Fig. 2. Definition of extensional interpretation of event patterns. We use $p_{(x)}$ for patterns, $q_{(x)}$ for rational numbers, $t_{(x)}$ for terms and pr for event predicates.

Now, we define an interpretation $\mathcal{I} : \mathit{Ground} \to 2^{\mathbb{Q}^+\times\mathbb{Q}^+}$ as a mapping from the ground atoms to sets of pairs of nonnegative rationals, such that $q_1 \leq q_2$ for every $\langle q_1,q_2\rangle \in \mathcal{I}(g)$ for all $g \in \mathit{Ground}$.

Given an event stream ϵ, an interpretation $\mathcal{I}$ is called a *model* for a rule set $\mathcal{R}$ – written as $\mathcal{I} \models_\epsilon \mathcal{R}$ – if the following conditions are satisfied:

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

Given an interpretation $\mathcal{I}$ and some $q \in \mathbb{Q}^+$, we let $\mathcal{I}|_q$ denote the interpretation defined via $\mathcal{I}|_q(g) = \mathcal{I}(g) \cap \{\langle q1,q2\rangle \mid q2 - q1 \leq q\}$.

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

A model $\mathcal{I}$ is called *minimal* if there is no other model preferred to $\mathcal{I}$. It is easy to show that for every event stream ϵ and rule set $\mathcal{R}$ there is a unique minimal model $\mathcal{I}^{\epsilon,\mathcal{R}}$.

Finally, given an atom a and two rational numbers q_1, q_2, we say that the event $a^{[q_1,q_2]}$ is a *consequence* of the event stream ϵ and the rule base $\mathcal{R}$ (written $\epsilon,\mathcal{R} \models a^{[q_1,q_2]}$), if $\langle q_1,q_2\rangle \in \mathcal{I}_\mu^{\epsilon,\mathcal{R}}(a)$ 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.[6] This justifies to take

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

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.

3 A Deductive Rule-based Approach for Complex Event Processing

In Section 1 we have numbered few advantages of CEP approaches based on logic. The majority of state-of-the-art in CEP is however not based on logic rules [1, 6, 14]; hence these advantages can be seen as features going beyond the state-of-the-art. In this section we review the features once again, justifying the design principles of our proposed language.

Expressive, formal, and declarative semantics. In the previous section we have defined *formal* and *declarative* semantics of an event processing language. CEP is a real-time processing, involving very often multi-threading and distributed processing. In such an environment, *declarative* semantics guarantees *predictability* and *repeatability* of results produced by an event processing system. This is not case in procedural CEP languages where, e.g., for the same input stream and the same set of event patterns, the system may produce two different results. Our proposed semantics is also *expressive* enough to capture all thirteen Allen's temporal relationships [3].

To evaluate expressivity of our language in practice, we have implemented *Fast Flower Delivery* use case[7] from [9]. The use case describes distribution of flowers from multiple stores (in a large city). The distribution is handled by a group of drivers who need to accomplish their assignments in a timely fashion. In the remaining part of this section we will use some of the use case patterns to demonstrate capabilities of our language.

Database and rule queries. Database information may serve in *enriching* an event with additional data. For instance, whenever a customer purchases flowers an event $request$ is triggered. A delivery request event (dlv_req) consists of the initial event $request$, *enriched* by additional information. This information is taken from a database relation $store\_info$, and can be pulled by a matching store ID ($StrID$). The relation contains, e.g., information about the store location, minimum accepted driver's rank, and the bidding preferences (see [9]).

$\texttt{dlv\_req}(ReqID, Loc_1, Loc_2, Time, MinRank, Pref) \leftarrow$
$\quad \texttt{req}(ReqID, Loc_1, Time, StrID) \text{ WHERE } \texttt{store\_info}(StrID, Loc_2, MinRank, Pref).$

In the above rule pattern, relation $store\_info$ contains *explicit* data. With no restriction, it could also contain a changing (updatable) data; or *implicit* knowledge derived by rules, possibly spanning over multiple relations, involving *recursions* and so forth.

Easy of programming. SQL-based syntax is predominant in today's CEP systems [1, 6, 14]. It is considered to be easy to understand, as many programmers today are familiar with database concepts. We propose a *rule-based syntax* and argue that it is convenient for CEP. We base our opinion gained on experience in implementation of Fast Flower

[7] Complete running implementation of the use case is published under name *ETALIS* on the Event Processing Technical Society web site: `http://www.ep-ts.com/content/view/79/`.

Delivery use case, as well as, on the implementation of our language. For example, let us consider the following simple pattern rules.

$\texttt{ce1}(Result) \leftarrow \texttt{e}(Name, Result)$ SEQ $\texttt{e}(Name, Result)$ WHERE $(Name =' a', Result = 1)$.
$\texttt{ce2}(Result) \leftarrow \texttt{ce1}(Result)$ AND $\texttt{ce1}(Result)$ WHERE $(Result = 1)$.

Their representation in an SQL-like language of Esper[8] based on [6] is shown below. As we see, complex events detect by the first pattern need to be *re-inserted* in a temporal stream of events $tmpE$ (using $insert$ in the first Esper statement). If complex event $ce2$ was further used in building a more complex event, we would to $insert$ instances of $ce2$ event in another temporal stream too. Consequently, very complex (nested) events in such a language can become easily unreadable. On the other hand, with a rule-based syntax it is easy to nest (complex) events. Also it is easy to pass data within events via *variable binding* which in total gives a more compact and clear syntax of the language.

```
<Query name= "ce1" text="
insert into tmpE(ceName, Result)
select 'ce1' as ceName, e1.Result as Result
from pattern [every ( +
    e1=e(e1.Name='a' and e1.Result=1) ->
    e2=e(e2.Name='a' and e2.Result=1) )]"/>

<Query name= "ce2" text="
select 'ce2' as Name, e1.Result as Result
from pattern [every ( +
    e1=tmpE(e1.ceName='ce1' and e1.Result=1) and
    e2=tmpE(e2.ceName='ce1' and e2.Result=1) )]"/>
```

Also it is worth mentioning that our prototype implementation consist of about 2500 lines of Prolog code (see Section 5), while Esper 3.3.0 has approx. 150000 lines of code. Hence rule-based declarative programming results in drastic *reduction* in code size.

Knowledge-based complex event processing. Events and event pattern rules represent *temporal* knowledge, based on which it is possible to derivate more complex dynamic matters. Apart from this knowledge, there may exists *static* (or evolving) knowledge (i.e., facts, rules and ontologies, constituting the *domain* knowledge). We have already seen how static data can be used for event enrichment. More importantly, to detect complex events we can also consult additional (*contextual*) knowledge, e.g., to prove *semantic* relations among matched events (not only temporal relations). While detecting complex events incrementally (at run time), our formalism may additionally evaluate the static knowledge (expressed as Prolog rules and facts) to enhance the detection. In this section we give an example where combining events with static knowledge may be beneficial in practice.

The following rule detects a $route$ event, i.e., a sequence of a delivery assignment event (dlv_assgn) followed by a driver's location event ($gpsToRegion$).

$\texttt{route}() \leftarrow \texttt{dlv\_assgn}(DrvID, Loc_1)$ SEQ $\texttt{gpsToRegion}(DrvID, Loc_2)$ WHERE $\texttt{reachable}(Loc_1, Loc_2)$.

To check whether the route is possible, rules defining *reachability* between two location are evaluated.

[8] Esper is a CEP system: `http://esper.codehaus.org/`.

$\texttt{reachable}(X, Y) \leftarrow \texttt{linked}(X, Y).$
$\texttt{reachable}(X, Z) \leftarrow \texttt{linked}(X, Y), \texttt{reachable}(Y, Z).$

Information about current connections (w.r.t traffic conditions, roads closed due to maintenance etc.) are encoded through the following links.

$\texttt{linked}(L_1, L_2)$
$\ldots$
$\texttt{linked}(L_{n-1}, L_n)$

We see that in order to detect event $route$, an occurrence of event $gpsToRegion$ needs to follow an occurrence of dlv_assgn. Additionally, the pattern will be matched only if the two events carry *reachable* locations (i.e., there exist a *semantic* relation, approved through the *background* knowledge).

Another example demonstrates use of rules for event *translation* and *classification*. Periodically sent drivers' gps events are translated to city region events (for convenience of store dispatchers).

$\texttt{gpsToRegion}(DrvID, Rg) \leftarrow \texttt{gps}(DrvID, coord(SNH, Lat, EWH, Long))$
$\quad \textsc{where}\ trsf\_rule(coord(SNH, Lat, EWH, Long), Rg).$

Prolog rules are used as background knowledge to classify gps coordinates into city regions.

$\texttt{trsf\_rule}(\texttt{coord}('N', X, 'W', Y), 'Manhattan') : -4042 < X, X < 4049, 7358 < Y, Y < 7370, !.$
$\ldots$
$\texttt{trsf\_rule}(\texttt{coord}('N', X, 'W', Y), 'StatenIsland') : -4034 < X, X < 4040, 7368 < Y, Y < 7399, !.$

In this section we arguable demonstrated powerful features of our formalism that go beyond (non-logic-based) state-of-the-art CEP systems [1, 6, 14]. In the following, we explain how complex events are detected using event-driven incremental computation, a feature that is the main difference of our work in comparison to other (logic-based) state-of-the-art CEP approaches [13, 16, 15, 7, 17].

4 Run-time Detection of Complex Events

This section describes how complex events, defined in Section 2.2, are computed at run-time. We explain the pattern matching procedure for a *sequence* of events. In principle the mechanism is similar for other operators too, which we omit due to space restrictions. For details about all operators, an interested reader is referred to [5].

Let us consider a sequence of events represented by rule (1) (e is detected when an event a[9] is followed by b, and followed by c). We can always represent the pattern (1) as $e \leftarrow ((a \ \textsc{seq}\ b) \ \textsc{seq}\ c)$. In general, rules (2) represent two equivalent rules.[10] We refer to this kind of "events coupling" as *binarization* of events. Effectively, in binarization we introduce *two-input* intermediate events (goals). For example, now we can rewrite rule (1) as $ie_1 \leftarrow a \ \textsc{seq}\ b$, and the $e \leftarrow ie_1 \ \textsc{seq}\ c$. 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

[9] More precisely, by "an event a" is meant an *instance* of the event a.
[10] If no parentheses are given, we assume all operators to be left-associative. While in some cases (e.g., SEQ) this is irrelevant, other operators such as PAR are not associative.

convenient to construct *event-driven* rules for three reasons. First, it is easier to implement an event operator when events are considered on "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. Each new use of an event (in a pattern) amounts to appending one or more rules to the existing rule set. However what is important for the management of rules, we don't need to *modify* existing rules when adding new ones (even when patterns with negations are added).

$$e \leftarrow a \text{ SEQ } b \text{ SEQ } c. \tag{1}$$

$$\begin{aligned} e &\leftarrow p_1 \text{ BIN } p_2 \text{ BIN } p_3 \ldots \text{ BIN } p_n. \\ e &\leftarrow (((p_1 \text{ BIN } p_2) \text{ BIN } p_3) \ldots \text{ BIN } p_n). \end{aligned} \tag{2}$$

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

Algorithm 4.1 accepts as input a rule referring to a binary sequence $e_i \leftarrow a$ SEQ b, and produces event-driven backward chaining rules (i.e. executable rules) for the sequence pattern. The binarization step must precede the rule transformation. Rules, produced by Algorithm 4.1, belong to one of two different classes of rules. We refer to the first class as to *goal inserting rules*. The second class corresponds to *checking rules*. For example, rule (4) belonging to the first class inserts $goal(b(\_,\_), a(T_1, T_2), e1(\_,\_)$. The rule will fire when a occurs, and the meaning of the goal it inserts is as follows: "an event a has occurred at $[T_1, T_2]$,[11] and we are waiting for b to happen in order to detect ie_1". Obviously, the goal does not carry information about times for b and ie_1, as we don't know when they will occur. In general, the *second* event in a goal always denotes the event that has just occurred. The role of the *first* event is to specify what we are waiting for to detect an event that is on the *third* position.

Algorithm 4.1 Sequence.

Input: event binary goal $ie_1 \leftarrow a$ SEQ b.

Output: event-driven backward chaining rules for SEQ operator.

```
Each event binary goal $ie_1 \leftarrow a$ SEQ $b$. is converted into: {
  $a(T_1, T_2) : -for\_each(a, 1, [T_1, T_2]).$
  $a(1, T_1, T_2) : -assert(goal(b(\_,\_), a(T_1, T_2), e1(\_,\_))).$
  $b(T_3, T_4) : -for\_each(b, 1, [T_3, T_4]).$
  $b(1, T_3, T_4) : -goal(b(T_3, T_4), a(T_1, T_2), ie_1), T_2 < T_3,$
          $retract(goal(b(T_3, T_4), a(T_1, T_2), ie_1(\_,\_))), ie_1(T_1, T_4).$
}
```

Rule (5) 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 (5) will fire whenever b occurs. The rule checks whether

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

$goal(b(T_3, T_4), a(T_1, T_2), ie_1)$ already exists (i.e. a has previously happened), in which case the rule triggers ie_1 by calling $ie_1(T_1, T_4)$. The time occurrence of ie_1 (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 $ie_1(T_1, T_4)$, 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. notify me if an event is about to happen; for example it 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 etc.).

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[12]. In this way, goals are kept persistent as long as (but not longer) than needed.

$$\begin{aligned}&for\_each(Pred, N, L) :- ((FullPred = ..[Pred, N, L]),\\&event\_trigger(FullPred), (N1 is N + 1),\\&for\_each(Pred, N1, L)) \lor true.\end{aligned} \tag{3}$$

$$a(1, T_1, T_2) :- assert(goal(b(\_, \_), a(T_1, T_2), e1(\_, \_))). \tag{4}$$

$$\begin{aligned}&b(1, T_3, T_4) :- goal(b(T_3, T_4), a(T_1, T_2), ie_1), T_2 < T_3,\\&retract(goal(b(T_3, T_4), a(T_1, T_2), ie_1(\_, \_))), ie_1(T_1, T_4).\end{aligned} \tag{5}$$

Finally, we see that for each different event type (i.e. a and b in our case) we have one rule with a for_each predicate. It is defined by rule (3). 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 4.1 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 (4). In general case, there may be as many of these rules as usages of a particular event may be manifold in an event program (i.e. set of all event patterns).

Memory management. It is worth mentioning that we have implemented two memory management techniques to *prune* outdated events, and hence free up memory in our running implementation (see Section 5). The first technique modifies the binarization step by pushing the time constraints (set by pattern's time window information; users are always encouraged to write patterns with certain time constraints). The technique ensures that time window constraints are checked during the incremental event detection. Therefore unnecessary intermediary sub-complex events will not be generated if the time constraints are violated (i.e., time expired). Our second solution for garbage collection is to prune expired events (goals) by using periodic events, generated by the system. Similarly to the first technique, rules are defined with time window constraints

[12] Removing "consumed" goals is often needed for space reasons but might be omitted if events are required in a log for further processing or analyzing.

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. Instead, events (goals) are pruned periodically as system events are triggered[13].

5 Implementation and Experimental Results

As a proof of concept, we have provided a prototype implementation of the language. In this section, we present experimental results of our prototype in comparison to Esper 3.3.0[14], i.e., we compare a declarative implementation (written in Prolog) versus a procedural one (written in Java). Esper is an engine primarily relying on state machines, i.e. a different paradigm that is widely used in CEP systems.

The test cases presented here were carried out on a workstation with Intel Core Quad CPU Q9400 2,66GHz, 8GB of RAM, running Windows Vista x64. Since our prototype automatically compiles the user-defined complex event descriptions into Prolog rules, we used SWI Prolog version 5.6.64[15] and YAP Prolog version 5.1.3[16]. All tested engines ran in a single dedicated CPU core.

To run tests, we have implemented an event stream generator, which creates time series data with probabilistic values. Event streams are generated so that every event in a stream is used for detection of one or more complex events (except the test defined by rule (7)). The whole output generated from all tests is validated, so we have made sure that all tested systems produce the same, correct, results.

Figure 3 shows experimental results we obtained for the *sequence* operator (SEQ). In particular, Figure 3(a) shows the throughput measurements for a pattern that exhibits a sequence of three events and the join operation on their Id attribute, see rule (1). The X-axis shows the event throughput achieved by the three different CEP systems: Esper 3.3.0, and our prototype (P) running on SWI and YAP Prolog, denoted as P-SWI and P-YAP respectively). The Y-axis shows different sizes of event streams, used for detection of complex events, defined by rule (6). In this test, our system clearly outperforms Esper. The throughput achieved by YAP engine is more than twice as big as the one produced by Esper. In Figure 3(b) we have evaluated the sequence which (apart from the join operation) also contain a selection parameter K (see rule (7)). K varies selectivity of Y attribute, ranging from 10% till 100%. When 10%-50% of the input events are selected, Esper shows significant advantage over our system. Hence in the future we need to review our implementation so to select events as early as possible. When all events are taken into account (100% selectivity), our system running on YAP is slightly better than Esper. We did this test on a stream of 25K events. In Figure 3(c) we extended the tests (for 100%) to check out whether the system throughput will remain constant for bigger streams (e.g., 50K-100K).

$$d(Id, X, Y, Z) : -a(Id, X) \text{ SEQ } b(Id, Y) \text{ SEQ } c(Id, Z). \tag{6}$$

[13] Frequency of system events can be programmatically scheduled, depending on available system memory.
[14] Esper: `http://esper.codehaus.org`
[15] SWI Prolog `http://www.swi-prolog.org/`.
[16] YAP Prolog: `http://www.dcc.fc.up.pt/~vsc/Yap/`.

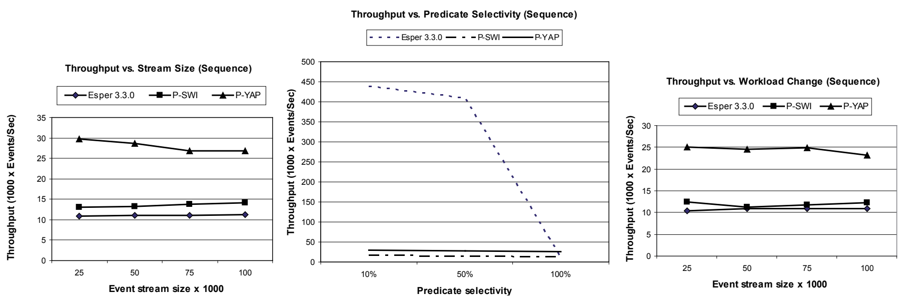

Fig. 3. Sequence - (a) Throughput (b) Throughput vs. Predicate Selectivity (c) Throughput vs. Workload Change

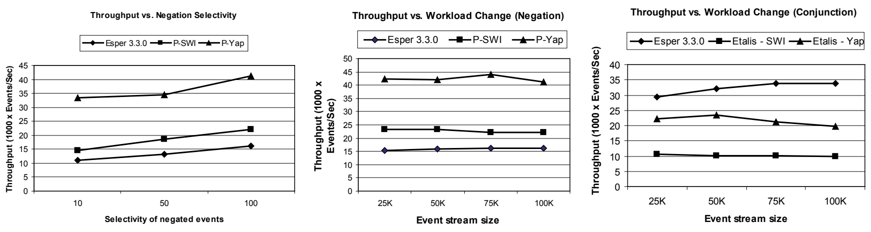

Fig. 4. Negation - (a) Throughput vs. Selectivity (b) Throughput vs. Workload Change (c) Conjunction - Throughput

Figure 4 presents experimental results for *negation* (NOT) and *conjunction* (AND). Figure 4(a) shows results obtained by evaluating a negated pattern from rule (8). The pattern is detected when an instance of a is followed by an occurrence of b with no c in between the two events. We have generated input event streams with different percentage of occurrences of events of type c (i.e., 10%-100%). We see that our prototype (either run by SWI or YAP) dominates over Esper. The test is computed on a stream of 25K. Figure 4(b) shows that the throughput does not go down even though we increased the stream size (e.g., 50K-100K). We have tested conjunction operator too. The pattern is specified by rule (9), and results are presented in Figure 4(c). Esper was faster in this test. Our algorithm for handling conjunction (see [5]) contains twice as many rules as the algorithm for sequence (i.e., two events in a conjunct may occur in any order). As a future work, we will try to improve the implementation of conjunction by corresponding rules in that algorithm.

$$c(Id, X, Y) :- a(Id, X) \text{ SEQ } b(Id, Y) \text{ WHERE } (Y < K). \tag{7}$$

$$d(Id, X, Y) :- a(Id, X) \text{ SEQ } b(Id, Y) \text{NOT} c(Id, Z). \tag{8}$$

$$d(Id, X, Y) :- a(Id, X) \text{ AND } b(Id, Y) \text{ AND } c(Id, Z). \tag{9}$$

$$d(Id, X, Y) :- a(Id, X) \text{ SEQ } (b(Id, Y) \text{ OR } c(Id, Y)). \tag{10}$$

$$\begin{array}{l} tc(X,Y) : -a(X,Y). \\ tc(X,Y) : -tc(X,Z) \text{ SEQ } a(Z,Y). \end{array} \tag{11}$$

Figure 5(a) shows results for *disjunction*, and evaluation of rule (10). In this test our system running on YAP was the most effective. The throughput for this test is similar to results for sequence (Figure 3(a)); this means that the presence of a disjunct does not ruin the performance of the sequence. We have also tested computation of the transitive closure (see rule (11)). The throughput change for different sizes of event streams are presented in Figure 5(b). Evaluation results were obtained under chronological consumption policy, see [5]. Our system on YAP was the fastest, however the difference between evaluations running on YAP and SWI was huge. Finally, Figure 5(c) compares the tested systems w.r.t event plan sharing. We have run an event program containing the same pattern (similar to rule (6)) multiplying the pattern 1, 8, and 16 times. The focus was on examining how well the systems can exhibit computation sharing among patterns. Our system run by YAP was not resistant on increase of pattern rules. However our prototype executed on SWI was still faster than Esper, see Figure 5(c).

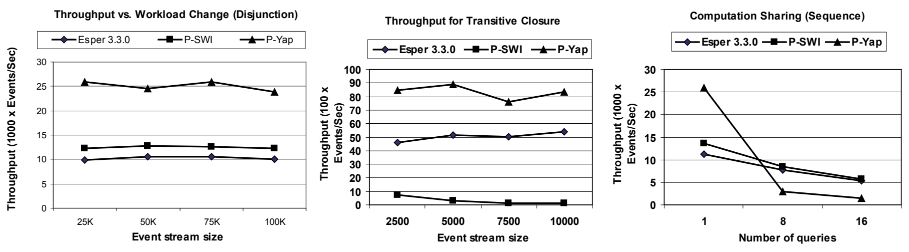

Fig. 5. (a) Disjunction-Throughput (b) Transitive Closure (c) Plan Sharing

.

At the end, let us mention that the cost of compilation of an event program (written in the proposed language) into Prolog rules is minor (typically few micro seconds).

In this section, we have provided measurement results of our running CEP engine. Even though there is a lot of room for improvements, preliminary results show that logic-based event processing has the capability to achieve significant performance. Working 15 months on this project, we have managed to develop a CEP language and a corresponding system that is competitive to a mature CEP engines such as Esper 3.3.0. Taking inference capability into account, logic-based CEP goes beyond the state-of-the art providing a powerful combination of *deductive* capabilities and *temporal* features, while at the same time exhibiting competitive run-time characteristics.

6 Conclusions and Future Work

We propose a language for Complex Event Processing based on deductive rules. The language comes with a clear declarative, formal semantics for complex event patterns. We have also provided a prototype implementation of the language, which allows for specification of complex events and their detection at occurrence time. Our approach clearly substantiates existing event-driven systems with declarative semantics and the power of knowledge-based event processing.

7 Acknowledgments

This work was supported by European Commission funded project SYNERGY (FP7-216089). We thank Jia Ding and Ahmed Khalil Hafsi for their help in the implementation and testing of ETALIS prototype.

References

1. J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, 2008.
2. J. J. Alferes, F. Banti, and A. Brogi. An event-condition-action logic programming language. In *JELIA 06*. Springer, 2006.
3. J. F. Allen. Maintaining knowledge about temporal intervals. In *Communications of the ACM 26, 11, 832-843*, 1983.
4. A. Alves. Extensions to logic programming inference engines to support cep. In *RuleML '09*, 2009.
5. D. Anicic, P. Fodor, S. Rudolph, R. Stühmer, N. Stojanovic, and R. Studer. Etalis: Rule-based reasoning in event processing. In S. Helmer, A. Poulovassilis, and F. Xhafa, editors, *Reasoning in Event-based Distributed Systems, Studies in Computational Intelligence series*. LNCS, Springer Verlag, 2010.
6. A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. In *VLDB Journal*, 2003.
7. F. Bry and M. Eckert. Rule-based composite event queries: The language xchangeeq and its semantics. In *RR*. Springer, 2007.
8. S. Chakravarthy and D. Mishra. Snoop: an expressive event specification language for active databases. In *Data Knowledge Engineering*. Elsevier Science Publishers B. V., 1994.
9. O. Etzion and P. Niblett. *Event Processing in Action*. Manning Publications Co., Greenwich, CT, USA, 2010.
10. C. L. Forgy. Rete: A fast algorithm for the many pattern/ many object pattern match problem. In *Artificial Intelligence*, 1982.
11. S. Gatziu and K. R. Dittrich. Samos: an active object-oriented database system. In *IEEE Bulletin of the TC on Data Engineering*, 1992.
12. P. Haley. Data-driven backward chaining. In *International Joint Conferences on Artificial Intelligence*. Milan, Italy, 1987.
13. R. Kowalski and M. Sergot. A logic-based calculus of events. In *New Generation Computing*. Ohmsha, 1986.
14. J. Krämer and B. Seeger. Semantics and implementation of continuous sliding window queries over data streams. In *ACM Trans. Database Syst.* ACM, 2009.
15. G. Lausen, B. Ludäscher, and W. May. On active deductive databases: The statelog approach. In *ILPS'97*, 1998.
16. R. Miller and M. Shanahan. The event calculus in classical logic - alternative axiomatisations. In *Electron. Trans. Artif. Intell.*, 1999.
17. A. Paschke, A. Kozlenkov, and H. Boley. A homogenous reaction rules language for complex event processing. In *International Workshop on Event Drive Architecture for Complex Event Process*. ACM, 2007.