Semantic Web 1 (2009) 1–5
IOS Press

Stream Reasoning and Complex Event Processing in ETALIS

Editor(s): Martin Raubal, ETH Zurich, Switzerland
Solicited review(s): Carsten Keßler, University of Münster, Germany; Florian Probst, SAP Research, Germany; one anonymous reviewer

Darko Anicic [a], Sebastian Rudolph [b], Paul Fodor [c], Nenad Stojanovic [a]
[a] *FZI Research Center for Information Technology, Karlsruhe, Germany*
E-mail: darko.anicic@fzi.de, nenad.stojanovic@fzi.de
[b] *Karlsruhe Institute of Technology, Germany*
E-mail: rudolph@kit.edu
[c] *Stony Brook University, NY, USA*
E-mail: pfodor@cs.stonybrook.edu

Abstract. Addressing dynamics and notifications in the Semantic Web realm has recently become an important area of research. Run time data is continuously generated by multiple social networks, sensor networks, various on-line services and so forth. How to get advantage of this continuously arriving data (events) remains a challenge – that is, how to integrate heterogeneous event streams, combine them with background knowledge (e.g., an ontology), and perform event processing and stream reasoning. In this paper we describe ETALIS – a system which enables specification and monitoring of changes in near real time. Changes can be specified as complex event patterns, and ETALIS can detect them in real time. Moreover the system can perform reasoning over streaming events with respect to background knowledge. ETALIS implements two languages for specification of event patterns: ETALIS Language for Events, and Event Processing SPARQL. ETALIS has various applicabilities in capturing changes in semantic networks, broadcasting notifications to interested parties, and creating further changes (based on processing of the temporal, static, or slowly evolving knowledge).

Keywords: Complex Event Processing, Stream Reasoning, ETALIS Language for Events, EP-SPARQL

1. Introduction

The amount of semantically annotated data and related ontologies in the Web is rapidly growing. More and more of this data is *real time* information where fast delivery is crucial. For instance, consider a sensor-based traffic management system, that needs to assess traffic situations in real time, and act accordingly (e.g., to issue traffic warnings, speed limit modifications etc.). Such a system has to cope with a high volume of continuously generated *events* (sensor readings), and correlate them with respect to *background knowledge* (a domain knowledge related to traffic management).

Semantic Web tools utilize various techniques to understand the meaning of information, and further, to *reason* about that information. With rapidly changing information this capability is however not sufficient. A conclusion, that was derived a minute ago, may not be valid right *now*. Hence, instead of a reasoning service over static data, there is a requirement to enable reasoning over rapidly changing (streaming) data.
Stream Reasoning. While reasoning systems year-over-year improve in reasoning over a static (or slowly evolving) knowledge, reasoning over *streaming* knowledge remains a challenge. For instance, ontological knowledge can efficiently represent a domain of interest (of an application), and help in harnessing the semantics of information. However, the task of reason-

ing over streaming data (e.g., RDF triples) with respect to a background ontology constitutes a new challenge known as *Stream Reasoning* [16].

Complex Event Processing (CEP) is a set of methods and techniques for real time information processing. CEP is concerned with detection of near real time situations (complex events) that are of a particular business interest. A *complex event* can be perceived as a composition of various more simple events (e.g., sensor readings, elementary changes, updates etc.) that happens satisfying different *temporal*, and *causal* relationships.

Today's CEP, however, suffers from two limitations: it can provide on-the-fly analysis of streams of events, but cannot combine streams with *background knowledge* (e.g., domain ontologies, required to describe context in which events are composed); and it does not support *reasoning* tasks over events and that knowledge. CEP should be extended by classic semantic techniques for knowledge management, thereby creating Semantic-based Complex Event Processing (SCEP).

Semantic Complex Event Processing. In state-of-the-art CEP systems [2,9,7,14,12] complex patterns are detected only by examining *temporal* and *causal* relations between events. Yet the question is whether examining these relations is (necessarily) sufficient to detect real time situations of interest. Complex events are used to trigger *time critical actions* or *decisions*, and hence ensure appropriate response to certain situations. The question is whether complex events, detected by current CEP systems, are expressive enough to capture complex situations in all their aspects. For example, consider a sequence "event a is followed by event b in the last 10 seconds". How likely is to use such a complex event for triggering *critical* business decisions? For some applications, this and similar patterns are expressive enough; however, for *semantic-rich* applications they are certainly not. In such applications, time critical actions are not only triggered by events. Instead, additional *background knowledge* is required to describe context in which complex situations are detected. This knowledge usually captures a *domain of interest* related to business critical actions and decisions.

State-of-the-art CEP systems [2,9,7,14,12] provide integration of event streams with databases. However *explicitly* represented data from databases are not always sufficient. To match two events in a pattern, we often need to prove *semantic* relations between them; and to do that, we need to process background knowledge, describing relations between events. From this knowledge we can discover, not only explicit semantics, but also to derive *implicit* relationships. Being able to evaluate the knowledge on-the-fly, we can for example *enrich* recorded events with the background information; detect more *complex situations*; give certain intelligent *recommendations* in real time; or accomplish complex event *classification*, *clustering*, and *filtering*.

This paper presents an open source system called *ETALIS* [6]. The system is capable of effectively detecting *complex events* over streaming data. Moreover, ETALIS can evaluate *domain knowledge* on-the-fly, thereby proving *semantic relations* among events and *reasoning* about them. This important feature has been recognised recently in various related approaches [8,10,15]. In contrast to these, ETALIS follows a completely deductive rule-based paradigm, thereby providing an effective solution for CEP and Stream Reasoning.

2. ETALIS

2.1. Conceptual Architecture

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 and so forth. Simple events are combined into complex events depending on their temporal, causal and semantic relations.

The task of Complex Event Processing and Stream Reasoning in ETALIS is depicted in Figure 1, and it can be described as follows. Within some dynamic setting, events from multiple event sources take place (see "Events" in Figure 1). Those *atomic events* are instantaneous. 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 ETALIS in the order of their occurrence.

Further on, ETALIS features a set of *complex event descriptions* (denoted as "*Event Patterns*") by means of which *"Complex Events"* can be specified as temporal constellations of atomic events (see Figure 1). The complex events, thus defined, can in turn be used to compose even more complex events. As opposed to atomic events, those complex events are not consid-

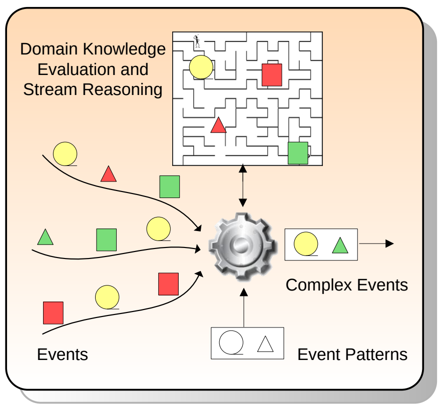


Fig. 1. Illustration of ETALIS' Conceptual Architecture

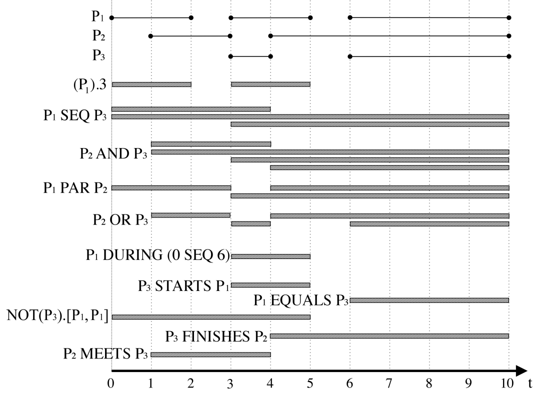


Fig. 2. Language for Event Processing - Composition Operators

ered instantaneous but are endowed with a time *interval* denoting when the event started and when it ended.

Finally, when detecting complex events, ETALIS may consult *background knowledge*. For instance, consider a traffic management system that detects areas with slow traffic (in real time). Such an area is detected when events, denoting *slow* traffic in a particular area, subsequently occur within a certain time span. What is a "slow" traffic, and what is a "particular" area for different events, roads, and road subsections is specified as background (domain) knowledge. ETALIS can evaluate the background knowledge on the fly (when certain events occur), possibly inferring new *implicit* knowledge. This knowledge is derived as a logical consequence from deductive rules, thereby providing the *Stream Reasoning* capability as illustrated with the upper part of Figure 1. In Section 2.4 we further detail the internal process of Complex Event Processing and Stream Reasoning in ETALIS.

2.2. *ETALIS Language for Events*

ETALIS implements a *rule-based* language for events defined in [5]. It is called *ETALIS Language for Events* (ELE). In this subsection we briefly review the language capabilities thereby reflecting capabilities of the system itself.

Figure 2 demonstrates various ways of constructing complex event patterns in the ETALIS Language for Events. Moreover, the figure informally presents the semantics of the language (a formal semantics can be found in [5]).

Let us assume that instances of three complex events, P_1, P_2, P_3, are occurring in time intervals as shown in Figure 2. 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_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 $(0$ 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.
- 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 [1].

- 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.

2.3. An Example of an ELE Application

It is worthwhile to demonstrate how ETALIS can be used in practice. Let us consider a sensor-based traffic management system mention before. The system monitors continuously generated traffic events, and diagnoses areas with slow traffic (bottleneck areas).

For example, a bottleneck area is detected when two events, denoting slow traffic in the same area, subsequently occur within 30 minutes. Rule (1) detects such a situation.

$$\begin{array}{l}\texttt{bottleneckArea}(Area) \leftarrow \\ \quad \big(\texttt{trafficEvent}(Rd, S_1, N_1, W_1)\ \text{SEQ} \\ \quad \texttt{trafficEvent}(Rd, S_2, N_2, W_2)\big).30min \\ \quad \text{WHERE}\ \{ \\ \qquad \texttt{slowTraffic}(Rd, S_1), \\ \qquad \texttt{slowTraffic}(Rd, S_2) \\ \qquad \texttt{areaCheck}(Area, N_1, W_1) \\ \qquad \texttt{areaCheck}(Area, N_2, W_2)\}. \end{array} \tag{1}$$

`trafficEvent` carries information about a public road (*Rd*) for which the event is relevant; current traffic speed (S_i); and geographic location (N_i, W_i) of its source sensor. Apart from the temporal condition (denoted with SEQ operator and the 30-minute time window), traffic events need to satisfy other conditions too. First, they need to be considered for the same road (i.e., the two traffic events are joined on the same attribute, *Rd*). Second, they need to denote slow traffic and belong to the same area (see WHERE clause in rule (1)). We develop a simple knowledgebase (written as Prolog-style rules (3)-(4)), to enable evaluation of these conditions.

Let us define speed thresholds for particular roads, e.g., on the road rd_1, traffic under 40 kph is considered as slow (see facts (2)).

$$\begin{array}{l}\texttt{threshold}(rd_1, 40). \\ \texttt{threshold}(rd_2, 30). \\ \texttt{threshold}(rd_3, 50). \\ \quad .... \end{array} \tag{2}$$

Rule (3) gets information about speed from two traffic events (S_1, S_2), and evaluates to `true` if the speed is below the threshold for a road *Rd*.

$$\texttt{slowTraffic}(Rd, S) : -\texttt{threshold}(Rd, X), S < X. \tag{3}$$

Further on, we define traffic areas as rectangles[1] represented as four point coordinates.

$$\begin{array}{l}\texttt{area}(a_1, 4042, 4045, 7358, 7361). \\ \texttt{area}(a_2, 4045, 4048, 7361, 7363). \\ \texttt{area}(a_3, 4048, 4051, 7363, 7365). \\ \quad .... \end{array} \tag{4}$$

Rule (5), for given coordinates of an event sensor, retrieves a traffic area. In order to belong to the same area, two events must be matched by the same $Area$ attribute.

$$\begin{array}{l}\texttt{areaCheck}(Area, N, W) : - \\ \quad \texttt{area}(Area, X_1, X_2, Y_1, Y_2), \\ \quad X_1 < N, N < X_2, Y_1 < W, W < Y_2!. \end{array} \tag{5}$$

Now, when a `trafficEvent` occurs, followed by another occurrence of the same event, ETALIS will check the time window constraint. If the constraint is satisfied, ETALIS will check whether the traffic is slow (by evaluating rule (3)), and whether both events come from the same area (rule (5)), in which case a `bottleneckArea` event is triggered.

In this simple example, we have demonstrated how to combine CEP capabilities with evaluation of background knowledge, thereby providing an effective on-the-fly situation assessment. The example also demonstrates how to apply *temporal* and *spatial* processing over continuously arriving events.

2.4. Internals of processing in ETALIS

In this subsection we give more details about internal processing in ETALIS, i.e., how events specified in ELE can be detected at run time. Our approach to

[1] Other geometric shapes can be represented by rules too.

do SCEP is based on deductive (logic) rules. Such an approach enables us not only to do event processing, but also to process a domain knowledge in an event-driven fashion. To achieve this goal, ETALIS is developed as a deductive system. However, deductive systems are rather suited for a *request-response* paradigm of computation. That is, for a given *request*, a deductive system will evaluate available knowledge and *respond* with an answer. In our case, this means that a deductive system needs to check whether a complex event 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 event is not detected until the next time the same request is processed (though it can become satisfied in-between the two checks). In event processing, this is not a desirable behaviour. Complex events need to be detected as soon as they happen. Therefore, to overcome this difficulty, we have proposed *event-driven backward chaining* (EDBC) rules in [5].

EDBC rules represent a basic mechanism in ETALIS that "converts" the request-response computation into an *event-driven* processing. It is a mechanism which enables a deductive system to derive a complex event at the moment it really occurs (not at the moment when a request is posed). The notable property of these rules is that they are *event-driven*, i.e., a rule will be evaluated when an event, that matches the rule's head, occurs. In such a situation, a firing rule will insert a goal into the memory. The purpose of the goal is to denote that a certain event happened, and that the system "waits" for another appropriate event in order to produce a more complex goal. For example, let us consider pattern rule (6). When event a occurs, there will be an EDBC rule which will insert a goal stating that the system waits for event b to happen in order to produce intermediate event ie_1. Later, when event b occurs the system will insert a goal stating that intermediate event ie_1 occurred, and the system waits for event c to happen, in order to produce event c. We see that pattern rule (6) is split into binary rules: $ie_1 \leftarrow a \text{ SEQ } b$, and $e \leftarrow ie_1 \text{ SEQ } c$. ETALIS automatically compiles user defined pattern rules into binary rules. We refer to this compilation process as *binarization*.

The binarization eases internal processes in ETALIS for three reasons. First, it is easier to implement an event operator when events are considered on a "two by two" basis. Second, binarization increases the possibility for *sharing* among (complex) events and intermediate events (when the granularity of intermediate patterns is reduced). Third, the binarization facilitates the *management* of rules. Each new use of an event (in a pattern) amounts to appending one or more rules to an existing rule set. What is important is that we never need to *modify* the existing rule set[2].

ETALIS is a rule-based deductive system that acts as an event-driven engine. Figure 3 shows basic operational steps that are undertaken in ETALIS. Rectangles in the diagram are used to depict certain processes in ETALIS, while ovals represent either (external/internal) inputs to these processes, or (external/internal) outputs from them.

The system diagram starts by user-written *ETALIS CEP rules* as input. These rules specify complex event patterns according to ELE (see Section 2.2). ETALIS validates these rules with respect to the language grammar, and parses them[3]. As a result, ETALIS produces rules in an *internal format*, ready for the process of *binarization* (see Figure 3).

$$e \leftarrow a \text{ SEQ } b \text{ SEQ } c. \tag{6}$$

The *ETALIS Compiler* compiles binary rules into EDBC rules, i.e., executable rules (written in Prolog). These rules may be accompanied with background knowledge to describe the domain of interest (as discussed in Section 1 and Section 2.1). Domain knowledge is also expected to be expressed in Prolog (although, in Section 3, we will describe an extension of ETALIS that accepts RDFS ontologies as background knowledge too). Compiled rules, together with the domain knowledge, are then executed by a standard Prolog system (e.g., SWI, YAP, XSB etc.). EDBC rules are triggered by events from *Event streams* (see Figure 3). As a result EDBC rules continuously derive complex events as soon as they happen. Let us briefly explain the oval on the right hand side of Figure 3. Apart from pattern rules, detection of complex events also depends on *consumption policies*. Other important matters in ETALIS are *garbage collection*, and additional *algebra for reasoning* about time intervals, see Figure 3.

In event processing, consumption policies (or event contexts [11]) deal with an issue of *selecting* particular events occurrences when there are more than one event

[2]This property holds, even when patterns with negations are added.

[3]"parser.P" and "etalis.P" are source files that implement the corresponding functionality (see Figure 3) in our open source implementation [6].

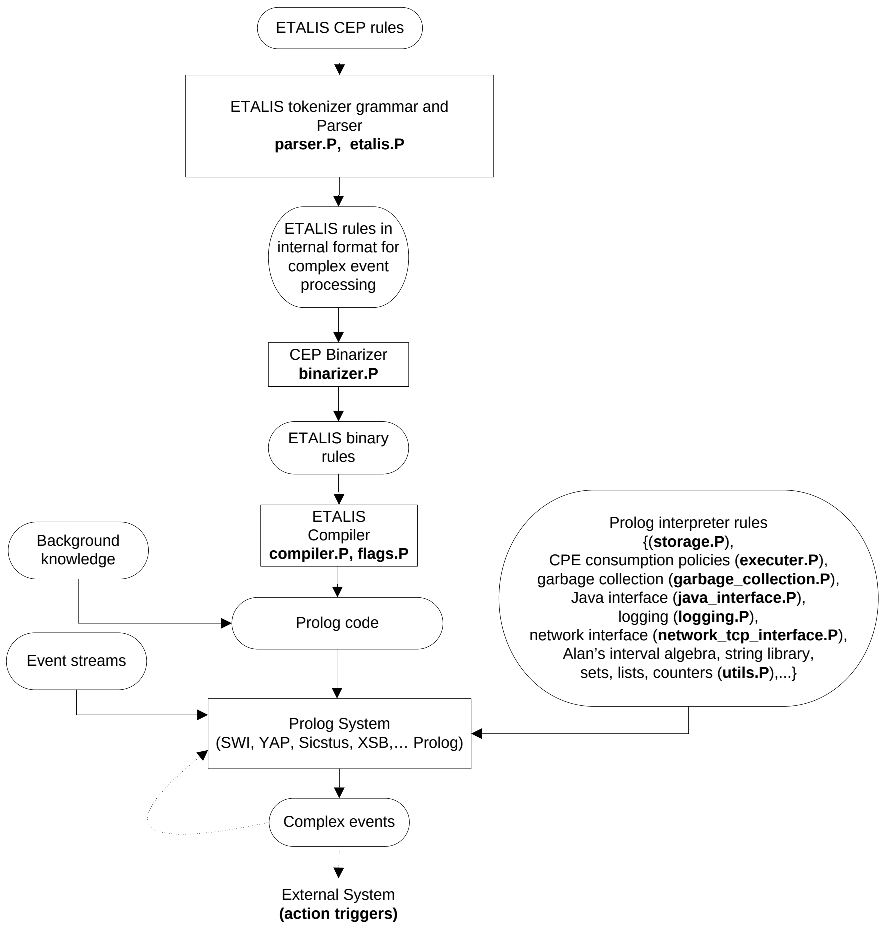

ETALIS CEP rules
ETALIS tokenizer grammar and Parser **parser.P, etalis.P**
ETALIS rules in internal format for complex event processing
CEP Binarizer **binarizer.P**
ETALIS binary rules
ETALIS Compiler **compiler.P, flags.P**
Background knowledge
Prolog code
Event streams
Prolog interpreter rules {(**storage.P**), CPE consumption policies (**executer.P**), garbage collection (**garbage_collection.P**), Java interface (**java_interface.P**), logging (**logging.P**), network interface (**network_tcp_interface.P**), Alan's interval algebra, string library, sets, lists, counters (**utils.P**),...}
Prolog System (SWI, YAP, Sicstus, XSB,... Prolog)
Complex events
External System **(action triggers)**

Fig. 3. System Diagram: ETALIS Language for Events

instance applicable and *consuming* events after they have been used in patterns. We have implemented three widely used consumption policies: *recent*, *chronological*, and *unrestricted* policy.

ETALIS also features two memory management techniques to *prune* outdated events. The first technique modifies the binarization step by pushing time constraints[4]. The technique ensures that time constraints are checked during the incremental process of events detection. This enables ETALIS to refrain from detecting intermediary (sub-complex) events when time constraints are violated (i.e., time windows have expired). Our second solution for garbage collection is to prune expired events by using periodic events, generated by the system. This technique does not check the constraints at each step during the incremental event detection. Instead, events are pruned periodically as system events are triggered.

As an algebra for reasoning about time intervals we have implemented Allen's temporal relationships [3]. Using this algebra, the system can also reason about intervals of detected complex events (e.g., to discover whether one complex event occurred during another complex event, whether one complex event starts/finishes another event, and so forth).

Finally, it is worth noting that detected complex events are fed back into the system, either to produce more complex events, or to trigger external actions in timely fashion. Typically, this situation happens when *iterative* event patterns are processed. Recursion is in the system diagram denoted by the backward (dashed) edge, see Figure 3.

[4]users are encouraged to write patterns with certain time window constraints

3. Stream Reasoning with EP-SPARQL

To enable ETALIS to be used in real time Semantic Web applications we have developed Event Processing SPARQL (EP-SPARQL) language [4]. This extension enables a user to specify complex event patterns in a SPARQL-like language which are continuously evaluated. Event streams are expected to be represented as timestamped RDF triples [4], and background knowledge can be specified as an RDFS ontology.

Syntactically, EP-SPARQL extends SPARQL by binary operators SEQ, EQUALS, OPTIONALSEQ, and EQUALSOPTIONAL. The operators are used to combine graph patterns in the same way as UNION and OPTIONAL in the pure SPARQL. Intuitively, all those operators act like a (left, right or full) join, but they do so in a selective way depending on how the constituents are *temporally* interrelated: P_1 SEQ P_2 joins P_1 and P_2 only if P_2 occurs strictly after P_1, whereas P_1 EQUALS P_2 performs the join if P_1 and P_2 are exactly simultaneous. OPTIONALSEQ and EQUALSOPTIONAL are temporal-sensitive variants of OPTIONAL.

Moreover, we added the function `getDURATION()` to be used inside filter expressions. This function yields a literal of type `xsd:duration` giving the length of the time interval associated to the graph pattern the FILTER condition is placed in. Likewise, we added functions `getSTARTTIME()` and `getENDTIME()` to retrieve the time stamps (of type `xsd:dateTime`) of the start and end of the currently described interval.

3.1. An Example of an EP-SPARQL Application

We provide an example application with EP-SPARQL, further concerning the traffic management system.

The following EP-SPARQL query searches for roads for which two *slow traffic* events have been reported within the the *last hour*. For example, results from this query could be used to automatically modify a speed limit on a certain road (or its particular section).

```
PREFIX tr:  <http://traffic.example.org/data#>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?road ?speed WHERE
       { ?road   tr:  slowTrafficDue ?observ }
   SEQ { ?road   tr:  slowTrafficDue ?observ }
   AND { ?observ rdfs:subClassOf tr:SlowTraffCause}
   AND { ?observ tr:  speed ?speed }
FILTER ( getDURATION()<"P1H"^^xsd:duration)
```

Traffic can be slowed down due to various reasons. We define a simple RDFS knowledgebase to number few of them. The background knowledge will be evaluated when sensor observations (events) get reported. Only events reporting about `SlowTraffCause` will be selected.

Since (direct or indirect) subclasses of `SlowTraffCause` may also be relevant, ETALIS utilize a *reasoning* procedure to find out *subclass relationships*.

```
tr:Accident    rdfs:subClassOf tr:SlowTraffCause.
tr:GhostDriver rdfs:subClassOf tr:SlowTraffCause.
tr:BadWeather  rdfs:subClassOf tr:SlowTraffCause.
tr:Rain        rdfs:subClassOf tr:BadWeather.
tr:Snow        rdfs:subClassOf tr:BadWeather.
```

We assume that there exist various types of traffic observations. For example, `Observ_1` is a specific type of `tr:Accident`, and in general, there may exist more than one instance of each type (e.g., a traffic accident is classified as a head-on collision, side collision, rollover etc.). Additionally, for each type of an observation there may exist a suggested speed limit, and other relevant details (omitted here for simplicity reasons).

```
Observ_1
  rdf:type  tr:Accident ;
  tr:speed  "70"^^xsd:int .

Observ_2
  rdf:type  tr:GhostDriver ;
  tr:speed  "50"^^xsd:int .

Observ_3
  rdf:type  tr:Snow ;
  tr:speed  "40"^^xsd:int .
```

Finally, to enable detection of *indirect* observations (e.g., of `SlowTraffCause` class) we utilise the subclass relation rule (7).

$$\begin{aligned}&\texttt{rdf:type}(\texttt{A}, Y) :- \\ &\quad \texttt{rdfs:subClassOf}(X, Y), \texttt{rdf:type}(\texttt{A}, X).\end{aligned} \tag{7}$$

Note that, by using deductive rules (e.g., rule (7)), ETALIS can be used to *infer* implicit knowledge (i.e., not only explicitly stated knowledge). This powerful feature is beyond the state-of-the-art CEP systems [2, 9,7,14,12], and enables a more advanced processing over streaming data.

3.2. Internals of EP-SPARQL Implementation

EP-SPARQL is implemented as an extension to ELE (see Section 2.4). A system diagram of the EP-SPARQL extension is shown in Figure 4.

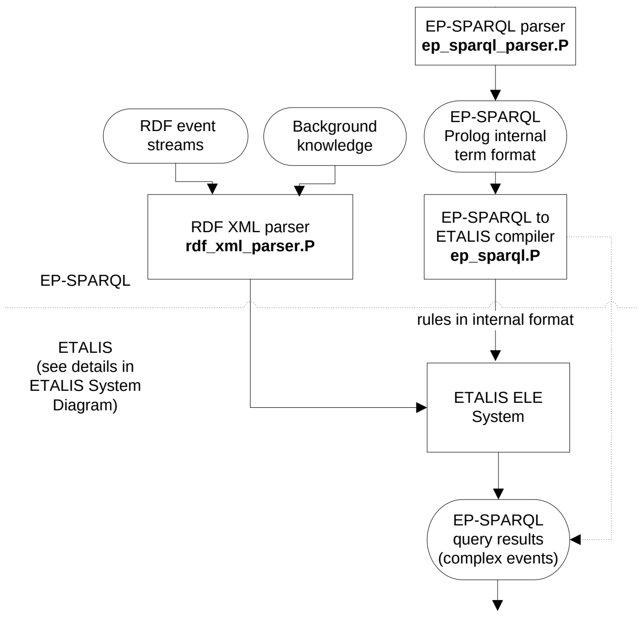


Fig. 4. System Diagram: EP-SPARQL

A user writes EP-SPARQL queries and deploys them into the engine. These queries act similarly as *continuous* queries in Database Stream Management Systems (DSMS), i.e., once registered, queries are continuously evaluated with respect to streaming data. In our implementation, the engine *incrementally* matches incoming data (events), thereby producing complex events as soon as they occur (see Section 2.4).

Since event streams and background knowledge are both represented in RDF, we use an RDF/XML parser to convert inputs into internal ETALIS format (see Figure 4). For event streams, the conversion is applied on-the-fly. It is a straight forward mapping that typically does not cause a significant overhead at run time. Background knowledge – expressed in a form of RDFS ontologies – is static knowledge, hence is converted into a Prolog program at design time. Similarly, we have also implemented a parser for the EP-SPARQL syntax and a compiler which produces EDBC rules out of EP-SPARQL expressions. All three inputs (EP-SPARQL queries, event streams and a domain ontology) are then fed into ETALIS, where the processing as described in Section 2.4 takes place.

4. Using ETALIS

ETALIS can be accessed in the following ways:

- interaction through the command line interface;
- access through a foreign language interface.

The command line interface is suitable for development, testing and deployment of an event-driven application. Since CEP tools belong to middleware systems – where they serve as a part of other complex systems – ETALIS is designed to be interfaced from other programming languages (e.g., Java, C and C#). This also enables ETALIS to be combined with existing programs and libraries. For more details on this topic, we refer the interested reader to [6]. Questions and issues related to use of ETALIS and its further development are discussed in a Google group for ETALIS[5]. Also, to see typical use of ETALIS, the interested reader is referred to various projects where ETALIS has been deployed[6,7,8,9,10].

5. Experimental Results

ETALIS is implemented in Prolog, and freely available from [6]. As mentioned, the system has already

[5]`http://groups.google.com/group/etalis`
[6]SYNERGY: `http://www.synergy-ist.eu/`
[7]PLAY: `http://www.play-project.eu/`
[8]ALERT: `http://www.alert-project.eu/`
[9]ARtSENSE: `http://www.artsense.eu/`
[10]ReFLEX: `http://www.reflexforsmes.eu/`

been used in few research and academic projects. In this section we present results from two experiments with ETALIS. The first test compares ETALIS to Esper 3.3.0[11] – an open source, and commercially used engine. The second test presents an example application, demonstrating CEP capabilities combined with background knowledge evaluation. To run tests we have implemented an event stream generator which creates time series data with probabilistic values. The presented tests were carried out on a workstation with Intel Core Quad CPU Q9400 2,66GHz, 8GB of RAM, running Windows Vista x64.

Test 1: Comparison results. Figure 5 shows experimental results we obtained for ELE: SEQ and AND operators (evaluated in rule (8)), NOT operator (used in rule (9)), and OR operator (combined with SEQ operator in rule (10)). The rule set was implemented, executed, and verified for both systems, ETALIS and Esper 3.3.0 (for the same input event stream). Figure 5 shows a dominance of ETALIS system. Esper is an engine primarily relying on state machines – a concept that is widely used today in CEP systems. ETALIS is based on deductive rules and inferencing. Nevertheless, the simplicity of the ETALIS algorithms [5], combined with fast Prolog engines[12], gives to ETALIS advantage over Esper engine.

$$d(Id, X, Y) :- a(Id, X) \text{ BIN } b(Id, Y) \text{ BIN } c(Id, Z). \quad (8)$$

$$d(Id, X, Y) :- \text{NOT}(c(Id, Z)).[a(Id, X), b(Id, Y)]. \quad (9)$$

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

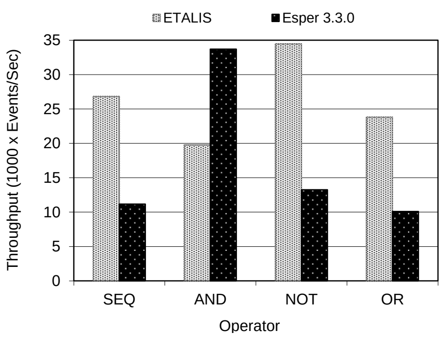


Fig. 5. Throughput – comparison results

Test 2: An example application. We developed an application using both, static RDF knowledge bases, and RDF event streams. The application implements a Goods Delivery System with traffic management capabilities in the city of Milan. The system comprises a set of delivery agents, that need to deliver manufactured products to consumers. Each of them has a list of locations that she needs to deliver goods to. While an agent is visiting a particular location, the system "*knows*" her next location, and "*listens*" to traffic-update events on that route/s. The routes are known thanks to a Milan ontology[13], which ETALIS uses as background knowledge to explore the city of Milan. If the agent requests the next route at the moment when the route is currently inaccessible, the system will find another route (on-the-fly calculating a transitive closure over the background ontology). The application has been implemented on the top of EP-SPARQL and ETALIS. Due to space limitations we cannot show patterns from the application here. Instead, we show in Figure 6 results obtained for 1 and 10 delivery agents, when visiting 20 locations. The time spent at a location is irrelevant for the test, hence it is ignored. We simulated situations where more than 50% of connections between visiting locations were inaccessible. In such a situation, the system needed to recalculate the transitive closure frequently.

The goal of the test was to show the usefulness of our formalism in a real use case scenario, as well as, to show that the application scales *linearly* with increase of number of agents (throughput for one agent is about 10 times higher than the throughput for 10 agents, see Figure 6 (a)). Similarly, Figure 6 (b) shows the memory consumption for the same test, demonstrating the linear space dependency w.r.t number of agents.

For more extensive ETALIS experiments, the interested reader is referred to [4] and [5]. We have also implemented a use case study from [13]. The implementation demonstrates how various common CEP operations can be implemented in ETALIS. The implementation is published by Event Processing Technical Society[14].

6. Conclusion

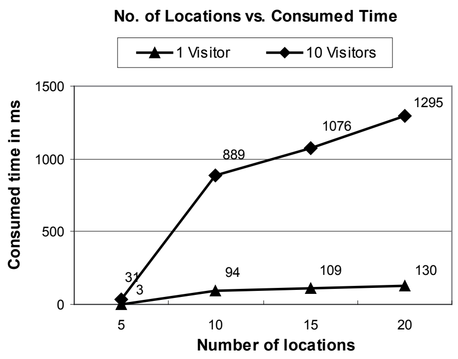
Addressing dynamics in the realm of the Semantic Web has recently become an important area of research. Real time processing of frequent changes has

[11]Esper: `http://esper.codehaus.org`

[12]relying on many optimizations from Logic Programming

[13]Milan ontology `http://www.larkc.eu/resources/published-data-sources/`

[14]`http://www.ep-ts.com/content/view/79/109/`



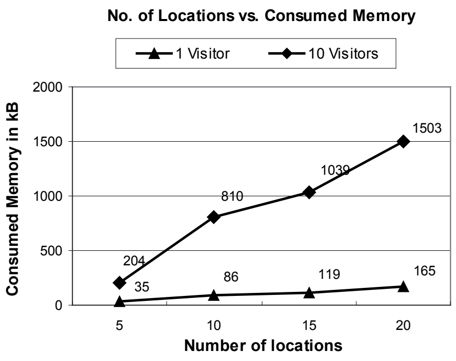


Fig. 6. Milan Sightseeing: (a) Delay caused by processing (b) Memory consumption

useful applications in many areas, including Web applications such as blogs and feeds, financial services, sensor networks, geospatial services, click stream analysis, etc. In this paper we have described ETALIS which is a system for *Complex Event Processing* and *Stream Reasoning*. The system can efficiently detect complex events in (near) real time, while evaluating background knowledge (e.g., an ontology). The knowledge is evaluated on-the-fly, either to capture the domain of interest (context), or to prove certain relations between matching events. ETALIS is an open source system.

References

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

[2] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In J. T.-L. Wang, editor, *Proceedings of the 28th ACM SIGMOD Conference*, SIGMOD'09, pages 147–160, New York, USA, 2008. ACM.

[3] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[4] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, editors, *Proceedings of the 20th International Conference on World Wide Web*, WWW'11, pages 635–644, New York, USA, 2011. ACM.

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

[6] D. Anicic, P. Fodor, R. Stühmer, and S. Rudolph. ETALIS home. `http://code.google.com/p/etalis`.

[7] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB Journal*, 15(2):121–142, 2006.

[8] D. F. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus. An execution environment for C-SPARQL queries. In I. Manolescu, S. Spaccapietra, J. Teubner, M. Kitsuregawa, A. Leger, F. Naumann, A. Ailamaki, and F. Ozcan, editors, *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT'10, pages 441–452, New York, USA, 2010. ACM.

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

[10] A. Bolles, M. Grawunder, and J. Jacobi. Streaming SPARQL - Extending SPARQL to process data streams. In S. Bechhofer, M. Hauswirth, J. Hoffmann, and M. Koubarakis, editors, *Proceedings of the 5th European Semantic Web Conference*, ESWC'08, pages 448–462, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings of 20th International Conference on Very Large Data Bases*, VLDB'94, pages 606–617, San Fransisco, USA, 1994. Morgan Kaufmann Publishers Inc.

[12] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In G. Weikum, J. Hellerstein, and M. Stonebraker, editors, *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research*, CIDR'03, 2003.

[13] O. Etzion and P. Niblett. *Event Processing in Action*. Manning Publications Co., Greenwich, CT, USA, 2010.

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

[15] D. Le-Phuoc, J. X. Parreira, M. Hausenblas, and M. Hauswirth. Unifying stream data and linked open data. Technical Report 2010-08-15, Digital Enterprise Research Institute, 2010.

[16] E. D. Valle, S. Ceri, F. v. Harmelen, and D. Fensel. It's a streaming world! Reasoning upon rapidly changing information. *IEEE Intelligent Systems*, 24:83–89, November 2009.