Semantic Web 0 (0) 1
IOS Press

Realizing Cascading Stream Reasoning with Streaming MASSIF

Pieter Bonte [a], Riccardo Tommasini [b], Emanuele Della Valle [b], Filip De Turck [a] and Femke Ongenae [a]
[a] *Ghent University - imec, IDLab, Department of Information Technology,*
Technologiepark-Zwijnaarde 15, Ghent, Belgium
E-mail: pieters.bonte@ugent.be, filip.deturck@ugent.be, femke.ongenae@ugent.be
[b] *Politecnico di Milano, DEIB,*
Piazza Leonardo da Vinci, 32, Milan, Italy
E-mail: riccardo.tommasini@polimi.it, emanuele.dellavalle@polimi.it

Abstract. To perform meaningful analysis over multiple streams of heterogeneous data, stream processing needs to support expressive reasoning capabilities to infer implicit facts and temporal reasoning to capture temporal dependencies. However, current stream reasoning approaches cannot perform the required reasoning expressivity while detecting time dependencies over high frequency data streams. Cascading Reasoning was meant to solve the problem of expressive reasoning over high frequency streams by introducing a hierarchical approach consisting of multiple layers. Each layer minimizes the processed data and increases the complexity of the data processing. However, the original Cascading Reasoning vision was never fully realized. Therefore, we propose a renewed and more generalized vision on Cascading Reasoning, serving as a blueprint for existing and future hierarchical approaches. Furthermore, we introduce Streaming MASSIF, a new Cascading Reasoning approach, performing expressive reasoning and complex event processing over high velocity streams. We show that our approach is able to handle high velocity streams up to hundreds of events per second, in combination with expressive reasoning and complex event processing.

Keywords: Stream Reasoning, Complex Event Processing, Description Logic Reasoning, Cascading Reasoning

1. Introduction

RDF Stream Processing (RSP) focuses on processing heterogeneous, high velocity data streams [43]. However, supporting expressive reasoning and temporal relatedness over these streams is still hard to achieve [21,19,32]. In this paper, we address this issue by presenting a layered approach that combines high velocity processing of data streams with expressive reasoning and complex event processing capabilities.

Due to the rise of the Internet of Things (IoT) and the popularity of Social Media, huge amounts of frequently changing data are continuously produced [6,10]. This data can be considered as unbounded streams. Many of those streams should be combined and integrated with background knowledge before they can be processed [18]. Combining streams and integrating background knowledge introduces more context and ensures more accurate results. Semantic Web technologies proved to be an ideal tool to fulfill these requirements [10,30,36]. Expressive reasoning and Complex Event Processing (CEP) techniques, allow to extract implicit facts in the streams, enabling meaningful analysis [19,39,41]. For example, a high traffic street can have many interpretations (depending on the type of street) and requires a rich background to model accurately.

RDF Stream processors [9,4,27] tackle the problem of combining various streams, integrating background knowledge and processing the data. However, they do not integrate expressive reasoning in their processing.

Existing work on expressive reasoning, such as Description Logic (DL) reasoning, has focused on static [38] or slowly changing [34] data. The problem of performing expressive reasoning over high veloc-

ity streams is however still not resolved [21]. Furthermore, temporal DL tend to become easily undecidable [29], making it even harder to perform temporal reasoning over high velocity streams. Through the use of CEP engines, temporal dependencies can be defined in various patterns. However, CEP engines struggle to integrate complex domains which makes it difficult to define complex patterns [41].

The Cascading Reasoning vision introduced by Stuckenschmidt, et al. [39], allows to perform expressive reasoning over high velocity streams by making a trade-off between complexity and data frequency. The vision presents various layers of processing, each with different complexities. However, the vision was never fully realized. Therefore, we propose a renewed and more general vision, serving as a blueprint for existing and future hierarchical approaches.

To tackle the challenges of performing expressive reasoning and detecting temporal dependencies over high velocity streams, we introduce Streaming MASSIF, a layered approach, based on our renewed Cascading Reasoning proposal.

Our approach combines RSP, expressive DL reasoning and CEP. We seamlessly combine DL and CEP enabling the definition of patterns using high-level concepts. This enables to use complex domain models within CEP and integrates a temporal notion in DL. The integration of RSP tackles the high velocity aspect of the streams. Furthermore, we introduce a query language that bridges the gap between stream processing, expressive reasoning and complex event processing.

We show that Streaming MASSIF is able to handle expressive reasoning and complex event processing over high velocity streams, up to hundreds of events per second.

The paper is structured as follows: Section 2 introduces an illustrative use case. Section 3 describes all the required background knowledge to understand the remainder of the paper. Section 4 details the original cascading reasoning vision, while Section 5 contributes our renewed and more generalized vision on cascading reasoning. Section 6 introduces a new query language that combines the various layers in cascading reasoning and describes the architecture of the system. Section 7 details the evaluation of our platform. The comparison to similar systems is described in Section 8. Section 9 discusses the results and how our platform compares to the state of the art. The conclusion and our outlook and direction for future work is elaborated in Section 10.

2. Motivating Example

In the Smart City of Aarhus [1], sensors have been integrated in multiple aspects of the city: traffic sensors to measure the traffic density, sensors to capture the occupation of parking spots and pollution sensors to measure the pollution values over the city. Since more and more employees have flexible working hours, we would like to notify them when it is a good time to go home. More specifically, that is when traffic near their offices starts decreasing. This notification should only be considered if the office allows flexible working hours. To achieve these kinds of notifications, we need to be able to:

1. *Combine various data streams*: To make meaningful analysis we need to combine streams from various sensors.
2. *Integrate background knowledge*: Since the sensory data typically only describes the sensor readings, we need to be able to link additional data, e.g. the type of measurement linked to the sensor, the location of the sensor, etc.
3. *Integrate complex domain knowledge*: For example, if we want to detect decreasing traffic near offices with flexible working hours, we first need to define what a flexible office is.
4. *Detect temporal dependencies*: To detect decreasing levels of traffic, we need to detect a temporal relation between low traffic observations and high traffic observations. More specifically, we need to detect when high traffic observations are followed by low traffic observations within a certain amount of time. Furthermore, we need to be able to filter out only those traffic updates going from high to low occurring in the same location.

Example 2.1. *In the ontology used to model our domain, we assign each Office various Policies. Based on these Policies an Office can be considered a FlexibleOffice or not:*

$$NoFixedHoursOffice \equiv Office \sqcap \exists hasPolicy.FlexibleHours,$$

$$NoFixedHoursOffice \sqsubseteq FlexibleOffice,$$

$$StartEarlyOffice \equiv Office \sqcap \exists hasPolicy.StartEarly,$$

$$StartEarlyOffice \sqsubseteq FlexibleOffice,$$

$$StopEarlyOffice \equiv Office \sqcap \exists hasPolicy.StopEarly,$$

$$StopEarlyOffice \sqsubseteq FlexibleOffice$$

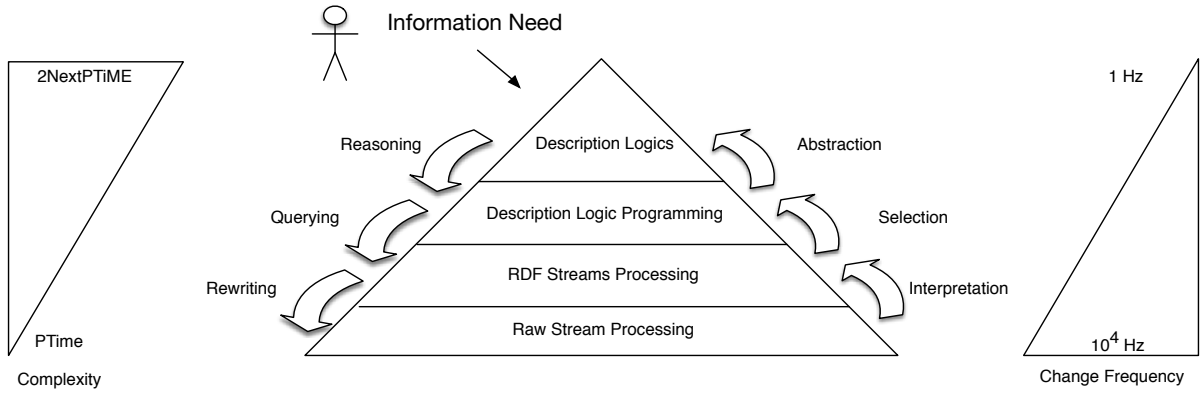

Fig. 1. Cascading Reasoning.

To model the observations that capture the various sensor readings across the city, we use the SSN Ontology [15].

Example 2.2. *We first model observations near flexible offices and then we model observations near flexible offices that also capture congestion levels:*

$$FlexibleOfficeObservation \equiv Observation$$
$$\sqcap (\exists observedFeature.(\exists isLocationOf.FlexibleOffice))$$
$$CongestionFOObservation \equiv FlexibleOfficeObservation$$
$$\sqcap \exists observedProperty.CongestionLevel$$

We can now model for each type of street, which is located near a flexible office, when it should be considered congested. With the congestion level defined as the number of detected vehicles divided by the street length (in meters):

$$HighTrafficMainRoadNearFlexibleOffice \equiv$$
$$CongestionFOObservation$$
$$\sqcap \exists observedProperty.MainRoad$$
$$\sqcap \exists hasValue > 0.025,$$
$$LowTrafficMainRoadNearFlexibleOffice \equiv$$
$$CongestionFOObservation$$
$$\sqcap \exists observedProperty.MainRoad$$
$$\sqcap \exists hasValue < 0.01,$$

Note that similar constructions can be made for different types of streets and that all these constructs are also subclasses of the concepts HighTrafficObservation or LowTrafficObservation.

Although this is a simplified version of the problem, it illustrates the challenges associated with it.

3. Background

In this section, we introduce the necessary knowledge to understand the content of the paper. First we introduce the original cascading reasoning vision and all the frameworks it contains and then we introduce Metric Temporal Logics.

3.1. The original Cascading Reasoning Vision

In particular, we provide details about Stuckenschmidt et al.'s vision of **Cascading Reasoning** [39] and the all the frameworks that it involves, i.e., *Raw Stream Processing*, *RDF Stream Processing (RSP)*, *Logic Programming (LP)* and *Description Logics (DL)*. The original vision is depicted in Figure 1.

3.1.1. Raw Stream Processing

This application domain comprises the bottom layer of the Cascading Reasoning pyramid and refers to those systems capable of processing large amounts of information in a timely fashion.

Raw Stream processing or **Information Flow Processing** (IFP) [16] describes how to timely process unbounded sequences of information, also called streams. IFP systems are divided into Data Stream Management Systems (DSMS) and Complex Event Processing (CEP) engines.

DSMSs extend traditional Data Base Management Systems to answer continuous queries that are registered and endlessly evaluated over time. Whereas a standard language for continuous queries does not

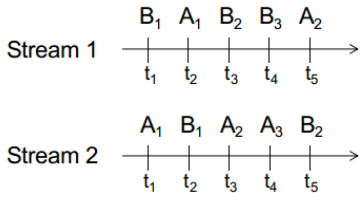

Fig. 2. Two example streams to illustrate the various event operators.

exist, the continuous query language (CQL) [5] is a prominent formalization adopted by most of the existing DSMSs query languages.

CEP Engines [28] are able to capture time dependencies between events. Complex events can be defined through event patterns consisting of various event operators. Examples of these event operators are the time-aware extensions of boolean operators (AND, OR) and the sequencing of events (SEQ). In the following, we present a list of the most prominent CEP operators, guards and modifiers:

- *AND* is a binary operator: A AND B matches if both A and B occur in the stream and turns true when the latest of the two occurs in the stream. In Figure 2, A AND B matches at t_2 in both Stream 1 and Stream 2.
- *OR* is a binary operator: A OR B matches if either A or B occurs in the stream. In Figure 2, A OR B matches at t_1 in both Stream 1 and Stream 2.
- *SEQ* is a binary operator that takes temporal dependencies into account. A SEQ B matches when B occurs after A, in the time-domain. In Figure 2, A SEQ B matches at t_3 in Stream 1 and at t_2 in Stream 2.
- *NOT* is a unary operator: NOT A matches when A is not present in the stream. NOT A matches at t_1 in Stream 1 and t_2 at Stream 2.
- *WITHIN* is a guard that limits the scope of the pattern within the time domain. A SEQ A WITHIN 2s matches in Figure 2 at t_3 in Stream 2 and not in Stream 1.
- *EVERY* is a modifier that forces the re-evaluation of a pattern once it has matched. EVERY A SEQ B matches at t_3 in Stream 1 and at t_2 & t_5 in Stream 2 for (A_2, B_2) and (A_3, B_2).

Example 3.1. *We can define a decreasing traffic observation as every high traffic observation followed by a low traffic observation within a certain amount of time, with the following event pattern:*
DecreasingTraffic = EVERY HighTraffic SEQ LowTraffic WITHIN 10m
However, it is not straightforward in CEP to define what a HighTraffic or LowTraffic exactly is.

For a comprehensive list of operators, we point the reader to Luckham [28]. Note that more advanced temporal relations exist, such as the ones presented in Allen's interval algebra [3].

3.1.2. RDF Stream Processing

RDF Stream Processing (RSP) [43] is an extension of IFP that can cope with heterogeneous data streams by exploiting semantic technologies. Resource Description Framework (RDF) streams are semantically annotated data streams encoded in RDF. RSP-QL [20] is a recent query language formalization that unifies the semantics of the existing approaches with a special emphasis on the operational semantics. The majority of the work on Stream Reasoning is focused in the area of RSP [21]. Therefore, in the following, we introduce some of RSP-QL definitions that are relevant to understand the next sections:

Definition 3.1. An **RDF Stream** S is a potentially infinite multiset of pairs (G_i, t_i), with G_i an RDF Graph and t_i a timestamp:
$$S = (g_1, t_1), (g_2, t_2), (g_3, t_3), (g_4, t_4), ...$$

Since a stream S is typically unbounded, a window is defined upon the stream in which the processing takes place.

Definition 3.2. A **Window** $W(S)$ is a multiset of RDF graphs extracted from a stream S. A **time-based window** is defined through two time instances o and c that are respectively the opening and closing time instants of each window: $W^{(o,c]}(S) = \{(g,t)|(g,t) \in S \wedge t \in (o,c]\}$.

Note that physical windows, based on the number of triples in the window, also exist [9].

Definition 3.3. A **time-based sliding window** $\mathbb{W}$ consumes a stream S and produces a time-varying graph $\overline{G}_{\mathbb{W}}$. $\mathbb{W}$ operates according to the parameters (α, β, t^0): it starts operating at t^0, it has a window width (α) and sliding parameter (β).

We now introduce the concepts of time-varying graphs and instantaneous graphs. The former captures the evolution of the graph over time, while the latter represents the content of a graph at a fixed time instant.

Definition 3.4. A **time-varying graph** $\overline{G}_{\mathbb{W}}$ is a function that selects an RDF graph for all time instants $t \in T$ where $\mathbb{W}$ is defined:
$\overline{G}_{\mathbb{W}} : T \rightarrow \{G | G_i \textit{ an RDF graph}\}$.
The RDF graph identified by the time-varying graph , at the time instant t, is called an **instantaneous graph** $\overline{G}_{\mathbb{W}}(t)$.

A dataset used within RSP-QL is defined as:

Definition 3.5. An **RSP-QL dataset** *SDS* is a set consisting of an (optional) default graph and n named graphs describing the static background data and m named time-varying graphs resulting from applying time-based sliding windows over $o \leq m$ streams, with $m, n \geq 0$.

Example 3.2. *In our example the SDS is defined as:*

$$SDS = \{G_0 = G_{sensors}, (w_1, \mathbb{W}_1(S_{traffic_1})), (w_2, \mathbb{W}_2(S_{traffic_2})), ...(w_n, \mathbb{W}_n(S_{traffic_n}))\}$$

$G_{sensors}$ describe the domain knowledge and the static data about the sensors such as their kinds, their locations, etc. $S_{traffic_i}$ describes the traffic observations and is windowed in $\mathbb{W}_i$. w_i is the window name.

To be able to query the *SDS* dataset, we define an RSP-QL query:

Definition 3.6. An **RSP-QL query** Q is defined as *(SE, SDS, ET, QF)* where:

- *SE* is an RSP-QL algebraic expression
- *SDS* is an RSP-QL dataset
- *ET* is a sequence of time instants on which the evaluation of the query occurs
- *QF* is the Query Form (e.g. Select or Construct)

3.1.3. Description Logic Programming

The reasoning application domain consists of the top two layers of the Cascading Reasoning pyramid. It refers to systems capable of deriving implicit knowledge from the input data combined with rules and domain models. The first reasoning layer in the original Cascading Reasoning vision was Logic Programs.

Logic Programs (LP) are sets of rules of the form head $\leftarrow$ body that can be read as *head "if" body*. The original vision of Cascading Reasoning referred to a specific fragment of LP, called Description Logic Programs (DLP) [23], which consists of the intersection between Description Logics and those LPs also expressible in First Order Logics. DLP can be seen of an ontological sub-language of DL that can be encoded in rules.

3.1.4. Description Logics

Description Logics (DL) [24] are the decidable fragment of First Order Logics and the second reasoning layer in Cascading Reasoning. DLs have formal semantics, ideal for many powerful reasoning tasks. DL defines *concepts* to represent classes of individuals and *roles* to represent binary relations between the individuals. Concrete roles (or data properties) are roles with datatype literals in the second argument. We call the concepts assigned to an individual, the types of the individual.

A Terminological Box (TBox) $\mathcal{T}$, is a finite set of concept (C) and role (R) inclusion axioms. An Assertion Box (ABox) $\mathcal{A}$ is a finite set of concept or role assertions. A *Knowledge base* $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ combines $\mathcal{T}$ and $\mathcal{A}$. $\mathcal{I}$ is an *interpretation* for $\mathcal{K}$ and $\mathcal{I}$ is a model of $\mathcal{K}$, if it satisfies all concept and role inclusions of $\mathcal{T}$ and all concept and role assertions of $\mathcal{A}$. This can be written as $\mathcal{I} \models \mathcal{K}$.

Example 3.3. *In Example 2.1 and 2.2 we already modeled the TBox. Lets consider a minimal ABox $\mathcal{A}$ describing the office, the road and their property:*

$$Office(office), hasPolicy(office, pol1), StopEarly(pol1), MainRoad(road), CongestionLevel(prop), propertyOf(prop, road), isLocationOf(road, office)$$

The observation capturing the current congestion level can be modeled as:

$$Observation(obs_i), observedProperty(obs_i, prop), hasValue(obs_i, 0.03)$$

By applying reasoning, we can infer from $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ that:

$$\mathcal{K} \models FlexibleOffice(office),$$
$$\mathcal{K} \models FlexibleOfficeObservation(obs1),$$
$$\mathcal{K} \models CongestionFOObservation(obs1),$$
$$\mathcal{K} \models HighTrafficMainRoadNearFlexibleOffice(obs1),$$
$$\mathcal{K} \models HighTrafficObservation(obs1)$$

3.2. Metric Temporal Logic

Metric Temporal Logic (MTL) [25] is a formalism designed to model real-time systems and reason about them using $\mathbb{R}$ as a time domain. It exploits the following modal operators:

- Diamond operators, i.e. refer to something happening sometimes in the past $\diamondminus_\varrho$ and sometimes in the future $\diamondplus_\varrho$.
- Box operators, i.e. always in the past $\boxminus_\varrho$ and in the future $\boxplus_\varrho$.
- Connectives, i.e. with respect to the future like *since* something happened $\mathcal{S}_\varrho$ or with respect to the past like *until* something happens $\mathcal{U}_\varrho$,

ϱ is a non-empty subset of $\mathbb{R}$, more specifically an interval $[a,b]$ such that $a,b \in \mathbb{R} b \geq a$.

MTL is undecidable if punctual operators ($\diamondminus_{[1,1]}$) are allowed and EXPSPACE-complete otherwise.

DatalogMTL [14] is a tractable fragment of MTL, which extends the Horn[1] fragment of MTL. It allows to define rules over a continuous temporal domain, e.g. $\mathbb{R}$, that consist of: (i) terms τ, i.e. individual variables or constants, and (ii) predicates of arbitrary arities (i.e. number of involved terms), of the form:

$$A^+ \leftarrow A_1 \wedge \cdots \wedge A_k \quad \text{or} \quad \bot \leftarrow A_1 \wedge \cdots \wedge A_k,$$

where $k \geq 1$, each A_i is either an inequality between two terms $\tau \neq \tau'$ or defined by the grammar:

$$A ::= P(\tau_1, \ldots, \tau_m) \mid \top \mid \boxminus_\varrho A \mid \boxplus_\varrho A \mid \diamondminus_\varrho A \mid \diamondplus_\varrho A \mid A \,\mathcal{U}_\varrho\, A' \mid A \,\mathcal{S}_\varrho\, A' \quad (1)$$

where $\boxplus, \boxminus, \diamondplus, \diamondminus, \mathcal{S}, \mathcal{U}$ are defined as in MTL. The atoms $A_1, \ldots, A_k$ constitute the *body* of the rule, while A^+ or $\bot$ its *head*. As usual for Datalog, every variable in the head of a rule also occurs in its body. Ranges ϱ in the temporal operators can be punctual $[r,r]$, to model the instantaneous occurrence of predicates. We focus on the non-recursive fragment of DatalogMTL, since query answering has a data complexity of AC^0[2].

4. Rethinking the Cascading Reasoning Vision

In this section, we identify why the current vision on Cascading Reasoning could never be fully realized and we propose a renovated and more general vision.

[1] i.e. the 'non-deterministic' operators $\diamondplus, \diamondminus, \mathcal{U}, \mathcal{S}$ are forbidden in formula's heads.

[2] For the formal semantics and the relative proofs we refer to [14].

4.1. Cascading Reasoning Limitations

Since Stuckenschmidt et al.'s vision of Cascading Reasoning was proposed, several new approaches populated the Stream Reasoning state of the art [21]. Although some of these new solutions also adopt a hierarchical architecture [33,35,12,4,13], none of them fully realize Stuckenschmidt et al.'s vision.

A first difference regards the reasoning techniques involved in the hierarchy. The initial scope of reasoning frameworks was focused mainly on DL and DLP. Recently, Temporal Logics, non-monotonic Logic Programs and technique for reasoning about time were proposed beside the traditional Stream Reasoning research areas. In the future, we also imagine the integration of on-line machine learning application, which already showed appealing results, and the combination of deductive and inductive reasoning [8,7].

In the original cascading reasoning pyramid, the role of RSP was limited to streaming data integration. Although this is utterly meaningful in combination with DL reasoning, data integration is a much more general problem to investigate when data are continuously changing. Moreover, RSP, but also stream processing, can support reasoning tasks (e.g. RSP under entailment or query rewriting).

Last but not least, the original cascading reasoning pyramid lacked the descriptive analytics aspects typical of Steam Processing (i.e. DSMS). Indeed, a common DSMS use-case is a decision-support application that requires to compute analytical queries.

We summarize these limitations as (i) *narrow scope of reasoning frameworks*; (ii) *RSP-centric data integration*; (iii) *lack of descriptive analytics aspects*.

4.2. Generalized Cascading Reasoning

Our proposed generalized cascading stream reasoning pyramid is depicted in Figure 3. As in the original vision, it aims at presenting the trade-off between expressiveness and rate of changes in the data. Practically, it forms a blueprint for existing and future hierarchical approaches.

We now detail each of the layers.

4.2.1. Stream Processing

At the lowest level, the data streams are processed. Different processing techniques can be used accordingly to the levels above, e.g. which information integration technique is used (if any). This layer can implement stream processing techniques like DSMSs and

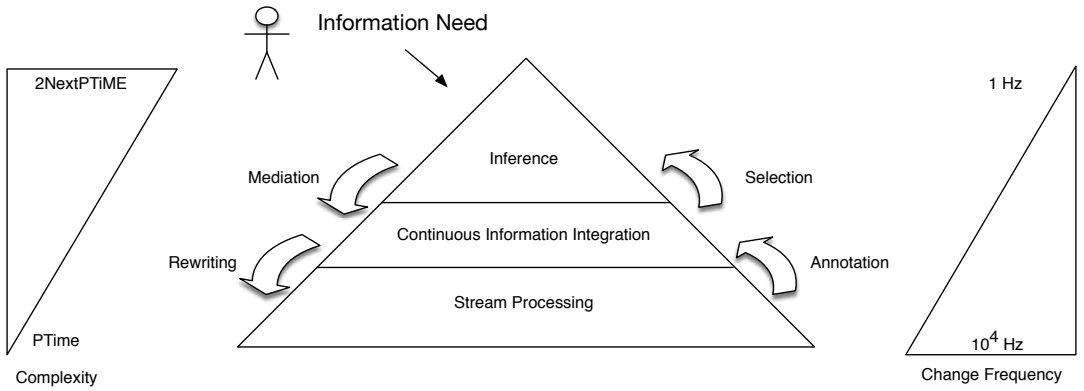

Fig. 3. A generalization of Cascading Reasoning.

CEPs or use RSP when dealing with semantically annotated data. Moreover, this level can also solve part of the analytic needs, since it is able to compute descriptive analysis of the streaming data.

4.2.2. *Continuous Information Integration*

In order to achieve a high-level view on the streaming data, we need an information integration layer that offers an homogeneous view over the streams.

The Continuous Information Integration layer combines data from heterogeneous streams into a common semantic space by the means of mapping assertions that populate a conceptual model. Two approaches are then possible to access the data: (i) Data *Annotation* (a.k.a. data materialization), i.e. data are transformed into a new format closer to the information need (ii) Query *Rewriting* (a.k.a. data visualization), i.e., the information need is rewritten into sub tasks that are closer to each of the original data formats.

4.2.3. *Inference*

In a cascading approach, an information need (IN) is formulated accordingly to a high-level view on the data. To enable efficient IN resolution, we need an inference layer that *mediates* the IN with domain-specific knowledge to the lower layers. Computational tasks at this level have a high complexity. This reduces the volume of data this level can actually process. Therefore, it is necessary to *select*, from the lower layers, the relevant parts of the streams that this layer has to interpret to infer hidden data. Possible inference implementations range from expressive reasoning, such as DL, Answer Set Programming (ASP), MTL or CEP to machine learning techniques such as Bayesian Networks (BN) or Hidden Markov Models (HMM).

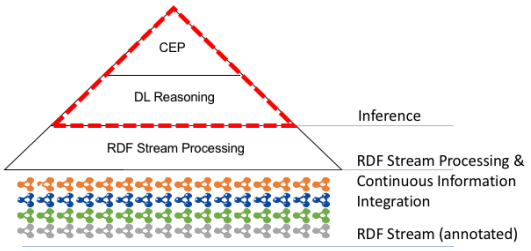

Fig. 4. Hierarchical layers of the new Cascading Reasoning approach.

Indeed, the original vision – which consists of raw stream processing, RSP, DL, and logic programming – fits this more general view: the raw stream processing is contained in our Stream Processing layer, RSP is contained in the continuous information integration layer and DL & logic programming are part of the inference layer.

5. A New Approach for Generalized Cascading Reasoning

Now that we have generalized cascading reasoning, we propose a new cascading reasoning approach consisting of a combination of CEP and DL as inference methods and RSP for continuous information integration.

5.1. *Layers Design*

In the following sections, we design a stream reasoning architecture that fulfills the requirements of our

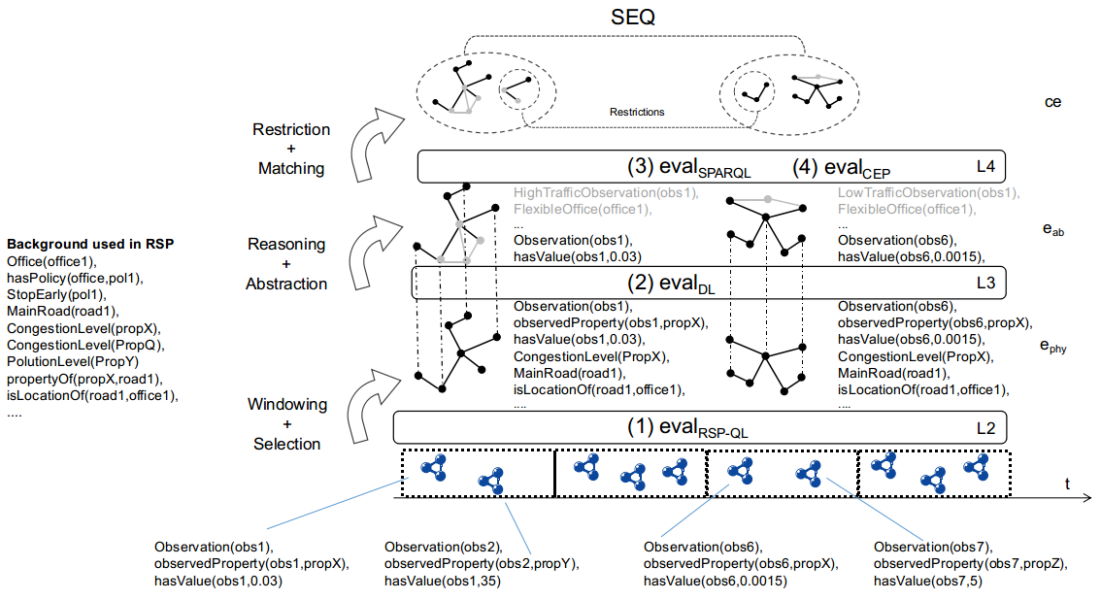

Fig. 5. Processing steps of the new Cascading Reasoning Approach.

motivating example and fits our proposal for cascading reasoning. As depicted in Figure 4, our approach consists of two layers that performs four tasks, starting from the bottom: (i) an RSP layer selects the parts of the streams that are relevant. (ii) It also integrates data from different streaming and static sources. (iii) An Inference layer enriches the output of the previous layer by deriving implicit data using DL reasoning. It also performs temporal reasoning via CEP on the inferred abstractions. We now discuss each of the layers in more detail.

5.1.1. RDF Stream Processing Layer

The RSP layer receives RDF streams (as defined in Definition 3.1) as input and answers continuous queries written in RSP-QL (see Definition 3.6). A given RSP-QL query Q is evaluated against a RSP-QL dataset *SDS* (as defined in Definition 3.5). The result of the defined queries are forwarded to the next layer. Therefore, we fix the Query Form to the CONSTRUCT query form.

5.1.2. Continuous Information Integration Layer

As we previously mentioned, we assume that data streams arrive directly encoded as RDF streams. This assumption allows us to perform stream processing and continuous information integration in the RSP layer by means of a common vocabulary.

Notably, we do not consider the annotation task (a.k.a. data materialization task) as part of the approach. If the data are not natively RDF streams, approaches like TripleWave [31], which rely on mapping techniques such as RML [22] and R2RML[3], can be utilized.

Example 5.1. *(cont'd) Since we're only interested in traffic observation that can be considered as high traffic observations, we select only the congestion level observations in the stream with a value above 0.03 or below 0.01. However, to determine that an observation is in fact a congestion level, we need to integrate with static background data describing the sensors. We also extract the information regarding the office near the location where the observation comes from, so we can determine later if these are flexible offices or not. Listing 1 shows a query Q that selects the relevant portion of the stream.*

In Figure 5 this query will select Observation(obs1) and Observation(obs6) from the stream. It will also add some additional data to the event, such as information regarding the road and the offices that can be used in the next layer for the expressive reasoning step.

Listing 1: Example of RSP-QL Query

```
CONSTRUCT {
 ?obs_X a ssn:Observation.
```

[3] https://www.w3.org/TR/r2rml/

```
 ?obs_X ssn:observedBy ?sensor_X.
 ?obs_X ssn:observedProperty ?property_X.
 ?property_X a CongestionLevel.
 ?obs_X hasValue ?value.
 ?property_X isPropertyOf ?foi.
 ?foi isLocationOf ?loc.
 ?loc hasPolicy ?pol. }
FROM NAMED WINDOW :traffic
  [RANGE 5m, SLIDE 1m] ON STREAM :Traffic
WHERE {
 ?property_X a CongestionLevel.
 ?property_X isPropertyOf ?foi.
 ?foi isLocationOf ?loc.
 ?loc hasPolicy ?pol.
 WINDOW ?w {
   ?obs_X a ssn:Observation.
   ?obs_X ssn:observedBy ?sensor_X.
   ?obs_X ssn:observedProperty ?property_X.
   ?obs_X hasValue ?value.
   FILTER(?value > 0.03 || ?value < 0.01) }
 }
```

5.1.3. The Inference Layer

The Inference layer of our architecture consists of two sub-layers: (i) *Description Logics*: since we want to infer information not explicitly available in the streams; and (ii) *Temporal Logics*: because we aim at deducting information based on temporal relations between the data. In the following, we explain how we link those sub-layers together.

First we need to make a distinction between physical events and abstract events:

Definition 5.1. A **physical event** e_{phy} is an event that occurs directly in the input stream S or is a result of the RSP layer. Note that in the latter, the event may also include background data. A collections of physical events is defined as E_{phy}.

Example 5.2. *(cont'd) In Figure 5, multiple physical events are depicted in a stream. Four physical events are detailed. One of these events is the following physical event that describes the first observation in the stream:*

$$Observation(obs1),$$
$$observedProperty(obs1, propX),$$
$$hasValue(obs1, 0.03)$$

In the RSP layer its enriched with the following information:

$$CongestionLevel(propX),$$
$$Office(office1),$$
$$MainRoad(road1),$$
$$isLocationOf(road1, office1),$$
$$StopEarly(pol1), hasPolicy(office1, pol1)$$

From these physical events we derive abstract events:

Definition 5.2. An **abstract event** e_{ab} consists of one or more physical events e_{phy} and hides their low-level details. An abstracted event e_{ab} can be inferred under an entailment Σ from a collection of physical events E_{phy} iff $\exists e_i \in E_{phy} : (\mathcal{T}, \mathcal{A}^+) \models C(e_i)$ with $C \in \mathcal{T}$ and $\mathcal{K} = (\mathcal{T}, \mathcal{A}^+)$, with $\mathcal{K}$ the knowledge based used in the reasoning process. $\mathcal{T}$ is the TBox and $\mathcal{A}^+$ the ABox, with $\mathcal{A}^+ = \mathcal{A} \cup E_{phy}$. We can now define the abstracted event as the triple $e_{ab} = (C, E'_{phy}, t)$, with E'_{phy} the collections of physical events in E_{phy} that lead to infer $C(e_i)$ and t the processing time at which the first physical event in E'_{phy} was produced. E_{ab} represents a collection of abstracted events. This is the case when multiple abstracted events can be abstracted.

Example 5.3. *(cont'd) The physical events can now be abstracted according to the defined ontology in Example 2.2. Through reasoning we obtain that:*

$$HighTrafficStreet(obs1),$$
$$HighTrafficMainRoadNearFlexibleOffice(obs1),$$
$$FlexibleOffice(office1)$$

This results in the following abstracted events with e_{phy} the physical event and t_i the time e_{phy} is produced.

$$(HighTrafficStreet, e_{phy}, t_i),$$
$$(HighTrafficMainRoadNearFlexibleOffice, e_{phy}, t_i),$$
$$(FlexibleOffice, e_{phy}, t_i)$$

We now want to identify temporal dependencies between the abstracted events provided by the DL sublayer. This reasoning task is suitable for temporal logics such as MTL. However, as we stated above, MTL is generally undecidable and intractable in most of the cases (see Section 3). Therefore, we identified

CEP operators	A_{now}	B_{now}
A AND B	$A \wedge \diamondminus_{(-\infty,0]} B$	$B \wedge \diamondplus_{[0,\infty)} A$
	$A \wedge \diamondplus_{[0,\infty)} B$	$B \wedge \diamondminus_{(-\infty,0]} A$
A SEQ B	$A \wedge \diamondplus_{(0,\infty)} B \wedge$ $(\neg A)\, \mathcal{U}_{(0,\infty)}\ A \wedge$ $(\neg B)\, \mathcal{U}_{(0,\infty)}\ B$	$B \wedge \diamondminus_{(-\infty,0)} A \wedge$ $(\neg A)\, \mathcal{U}_{(0,\infty)}\ A \wedge$ $(\neg B)\, \mathcal{U}_{(0,\infty)}\ B$
Every (A) SEQ B	$A \wedge (\neg A)\, \mathcal{U}_{(0,\infty)}\ B$ $A \wedge \diamondplus_{(0,\infty)} B \wedge$ $(\neg B)\, \mathcal{U}_{(0,\infty)}\ B$	$B \wedge (\neg B)\, \mathcal{S}_{(-\infty,0)}\ A$ $B \wedge \diamondminus_{(-\infty,0)} A \wedge$ $(\neg B)\, \mathcal{U}_{(-\infty,0)}\ B$
A SEQ Every (B)	$A \wedge \diamondplus_{[0,\infty]} B \wedge$ $(\neg A)\, \mathcal{U}_{(0,\infty)}\ A$	$B \wedge \diamondminus_{(-\infty,0)} A \wedge$ $(\neg A)\, \mathcal{U}_{(0,\infty)}$ $(\diamondminus_{(-\infty,0)} A)$
Every(A) SEQ Every(B)	$A \wedge \diamondminus_{(-\infty,0)} B$	$B \wedge \diamondplus_{(0,\infty)} A$

Table 1

Semantics of "AND" and "SEQ" combined with the modifier "Every".

non-recursive DatalogMTL as a proper fragment that serves our purpose (see Section 3). Moreover, considering the relation between Temporal Logics and CEP languages [17], a CEP engine can be used to efficiently evaluate the non-recursive DatalogMTL formulas that capture the semantics of CEP operators. Table 1 shows the equivalences between the CEP operators and non-recursive DatalogMTL formulas. For binary CEP operators we provide two DatalogMTL formulas, one form the viewpoint of A and one from the viewpoint of B. Thus, one that considers as "current" the time instant at which A occurs (A_{now}) and one considering the time instant at which B occurs (B_{now}).

Given these equivalences, it is possible to compute this fragment of MTL using a CEP engine. Therefore, we can detect temporal dependencies between the abstracted events provided by the DL sub-layer by defining event patterns.

Definition 5.3. An **event pattern** EP is a statement of the form

$$[\triangledown](E_1 \wedge \cdots \wedge E_k) | (E_1 \vee \cdots \vee E_k)[\triangle]$$

with E_i either (i) an event type or (ii) a complex event using one of the following operators: AND, OR, NOT, SEQ or (iii) another event pattern (recursively). $\triangledown$ is an optional modifier, e.g. EVERY and $\triangle$ is an optional guard, e.g. WITHIN.

We use these patterns to instantiate complex events that represent inferred information.

Definition 5.4. A **complex event** ce definition is a triple $ce = (h, p, R)$ with

- h the complex event type,
- p is the pattern defined using operators, modifiers and guards,
- R is a set of restrictions. Restrictions can be defined on the event values, e.g. event A(speed=45) has the property *speed* with a value of 45, one can restrict to speed values above a certain threshold. Other restrictions can be defined over events, e.g. if each event type has a location A(location=loc1) and B(location=loc1) then we can impose the restriction that event A and B should have the same location.

h is instantiated when p and R are satisfied.

The set of abstracted events (i.e. the collection of triples $(C_{\mathcal{E}}, e_{phy}, t)$) is used in the event pattern matching. More specifically each type $C_{\mathcal{E}}$ is checked if it matches the event types within the pattern. Additionally, the *restrictions* $R = (CR_{\mathcal{E}}, q_{SPARQL})$ can be defined on each event type in an event pattern. $CR_{\mathcal{E}}$ is an event type (i.e. defined in $\mathcal{E}$) and q_{SPARQL} is a SPARQL query. The SPARQL query is evaluated over each e_{phy} contained in the abstracted event $(e_{ab} = (C_{\mathcal{E}}, e_{phy}, t))$ where $C_{\mathcal{E}} == CR_{\mathcal{E}}$. Restrictions over multiple events in the event pattern can be achieved by creating multiple restrictions R with the same variable names in the q_{SPARQL}. The variable bindings are extracted and used for joining the events. This is shown in the restrictions of Example 5.4 through the use of the reoccurring variable name "?loc".

Example 5.4. *(cont'd) To detect the decreasing traffic, we need to monitor for high amount of traffic near flexible offices followed by low amounts of traffic near the same flexible offices within a certain time-range. This can be done by defining the complex event definition triple: $ce = (CE_{\mathcal{E}}, p, R)$ with*

- *$CE_{\mathcal{E}}$ the complex event type DecreasingTraffic.*
- *p the pattern describing EVERY HighTrafficAbstraction SEQ LowTrafficAbstraction WITHIN 10m*
- *R is a set of restrictions of the form $(CR_{\mathcal{E}}, q_{SPARQL})$ consisting of*
  * *$(HighTrafficAbstraction,\ q_1)$ with $q_1 =$*

```
Select * WHERE {
 ?o ssniot:hasLocation ?loc.}
```

* $(LowTrafficAbstraction, q_2)$ *with* $q_2 =$

```
Select * WHERE {
 ?o2 ssniot:hasLocation ?loc.}
```

Note that the restrictions state that high and low traffic events need to have the same location. The value in ?loc will be used to restricts the complex events, since its the only variable with the same name in q_1 *and* q_2.

5.2. Unified Evaluation Functions

In the following, we explain, by means of Figure 5, how to combine the different layers into a single evaluation framework. At the lowest level, we have the evaluation of the RSP layer. Let's consider an RSP-QL query Q. The evaluation of Q over dataset *SDS* is defined as:

$$\Omega(t) = eval(SDS(G), SE, t), \text{ with } t \in ET$$

where ET represents all the time instances where SDS is defined and Ω is a time-varying multiset of solution mappings that maps time T to the set of solution mappings multisets [20]:

$$\Omega : T \rightarrow \{\omega | \omega \text{ is a multiset of solution mappings}\}$$

We consider CONSTRUCTS query form only; therefore the solution mappings still need to be substituted in a graph template defined in the query[4].

$$G_\Omega(t) = \sigma(G_{template}, \Omega(t))$$

With σ the substitution function and $G_{template}$ the graph template defined in Q. The solution $G_{\Omega(t)}$, for each $t \in ET$ is a subset of the data in *SDS* and is sent to the next layer in the cascading reasoner for further processing. We can define the evaluation of the RSP layer as

$$eval_{RSP-QL}(SDS, Q) = G_\Omega(t), \forall t \in ET$$

Each time the RSP layer produces results, they are sent to the DL layer as a set of physical event $E_{phy} = G_\Omega(t)$. The DL layer converts the physical events E_{phy} to a set of abstracted events E_{ab} under a certain entailment Σ.

[4]As defined in the SPARQL 1.1. specification: https://www.w3.org/TR/sparql11-query/#construct

$E_{ab} = \{C_\mathcal{E}(e_i) | \exists e_i \in E_{phy} : (\mathcal{T}^+, \mathcal{A}^+) \models C_\mathcal{E}(e_i) \wedge C_\mathcal{E} \in \mathcal{E}$ with $\mathcal{T}^+ = \mathcal{T} \cup \mathcal{E}$ and $\mathcal{A}^+ = \mathcal{A} \cup E_{phy}\}$. The $eval_{DL}$ reasoning step is defined as

$$eval_{DL}(E_{phy}, \mathcal{E}, \mathcal{O}, \Sigma) = E_{ab},$$

where E_{ab} is the set of abstracted events and the quadruple $< E_{phy}, \mathcal{E}, \mathcal{O}, \Sigma >$ comprises:

- E_{phy} – a set of one or more selected physical events contained in $G_{\Omega(t)}$.
- $\mathcal{O}$ – the ontology describing the domain knowledge. $\mathcal{O} = (\mathcal{T}, \mathcal{A})$ with $\mathcal{T}$ the TBox and $\mathcal{A}$ the ABox describing $\mathcal{O}$.
- $\mathcal{E}$ – an ontology TBox that bridges the domain ontology $\mathcal{O}$ and the physical events E_{phy}. It describes formally the abstraction based on $\mathcal{O}$. Only the concepts in $\mathcal{E}$ will be considered as abstracted events.
- Σ – the entailment regime under which the reasoner has to extract the abstract events from E_{phy}.

Finally, we define the result of the evaluation of the CEP layer as a set of abstract events:

$$eval_{CEP+}(CE, E_{ab}) = \{(CE_\mathcal{E}, \bigcup e_{phy}, t)\}$$

with $CE_\mathcal{E}$ the complex event type of the complex event $ce \in CE$ that matched, e_{phy} the physical events in E_{ab} that cause the patterns to trigger and t the processing time at which the patterned triggered. In the resulting complex event, the union of the underlying physical events is taken and the complex event type is assigned.

Since complex events are still physically represented as RDF graphs, in order to evaluate restrictions we can simply extend $eval_{CEP}$ with $eval_{SPARQL}$ that evaluates the restrictions describes as SPARQL queries.

To ensure termination, we restrict to non-recursive pattern definitions, i.e. $\forall p \in CE, \nexists E \in p : CE_\mathcal{E} == E$. The complex event type is thus not allowed in the definition of the pattern.

5.3. Summary

To conclude, we described a stream reasoning stack that is able to a) select the relevant portions of the stream using RSP, b) abstract the selected RDF graphs using expressive reasoning techniques and selecting only those that match the expected abstractions and c) apply complex event processing over these abstractions to detect temporal dependencies.

6. Realizing Cascading Reasoning

In this section, we explain how we realized the proposed cascading reasoning system. In the following, we present a Domain Specific Language (DSL) and the architecture of Streaming MASSIF, i.e. a cascading stream reasoner that fully implements the semantics explained in Section 5.

6.1. A Domain Specific Language for Cascading Stream Reasoning

In this section, we introduce a DSL that allows users to formulate information needs by solvable using the proposed cascading reasoning approach.

In order to explain the DSL, we provide an example of information need and we explain how each part of the query maps to the different evaluations functions described in Section 5.2.

Listing 2 describes the grammar of the proposed query language. Note that for conciseness reasons, we did not incorporate the following sub-grammars:

1. DLDescription: The definition of the abstract event types ($C_{\mathcal{E}}$) is based on the Manchester syntax. For more information regarding this syntax we refer the reader to the Manchester W3C page[5].
2. BGP: In the definition of the complex events, one can define Basic Graph Pattern (BGP) for restricting the validity of the events. We did not incorporate the explanation of the syntax of BGP in this proposal.
3. RSPQL: For targeting the RSP module, we utilize RSP-QL. The full syntax of RSP-QL has not been incorporated in our syntax proposal, more information regarding RSP-QL can be found in Dell'Aglio et al. [20].

As defined in Listing 2, an information need comprises multiple namespaces (*NameSpace*), multiple event declarations (*EventDecl*) and an optional *RSPQL* declaration. Figure 6 a) shows an information need from the example use-case. We now explain how this DSL targets each module of the cascading stream reasoner.

6.1.1. DSL Fragment for the RSP Layer

From line 19 in Figure 6 a), the RSP-QL syntax is used for selecting the relevant events from various streams. Note that there is no query form defined, since we restrict the use to the construct query form. The construct query template is generated from the BGP in the WHERE clause. Note that the definition of the RSP-QL clause is optional in the language. In the absence of the RSP-QL clause all streaming data is directly processed by the next layer (i.e. the abstraction layer). In this case, each event in the stream is processed one by one.

6.1.2. DSL Fragment for the DL Sub-Layer

An information need typically requires to define multiple events. An event declaration (*EventDecl*) starts with the declaration of a *NAMED EVENT*, a name for the event (*EventName*) and either the definition of an abstract event (*AbstractEvent*) or and complex event (*ComplexEvent*). The abstract event definition start with the *'AS'* keyword to indicate how the event name should be interpreted, followed by a declaration in Manchester DL syntax. This is shown in Figure 6 a) on line 6 to 9. We chose the Manchester Syntax[6] for the definition of these events since its very concise and expressive.

6.1.3. DSL Fragment for the CEP Sub-Layer

Besides the *AbstractEvents*, the *EventDecl* clause can also define complex events (*ComplexEvents*). These are declared with the 'MATCH' keyword, followed by a modifier (*Modifier*), an event pattern (*EventPattern*), a guard (*Guard*) and an optional restriction clause (*IFClause*). The *EventPattern* is constructed from various abstract events and event operators (*EventOperators*). Figure 6 a) shows an example event pattern defined over high and low traffic abstractions on line 11 to 13.

The restrictions (*IFClause*) are declared using the 'IF' keyword, followed by the abstract event name used in the pattern that needs to be restricted. The restriction itself is defined in a BGP. Both filter and join restrictions can be modeled in this manner. An example on how to define join restrictions over multiple events can be found in Figure 6 a) on line 14 to 16. The restriction states that the high and low traffic abstractions should occur in the same location. Note that the variable name 'loc' is the same in both the restrictions.

We can also define restrictions to filter individual events:

Example 6.1. *Listing 3 shows a filter restriction example over the high and low traffic abstractions that*

[5] https://www.w3.org/TR/owl2-manchester-syntax/

[6] https://www.w3.org/TR/owl2-manchester-syntax/

Listing 2 Syntax of the Streaming MASSIF DSL

```
DSL –> NameSpace* EventDecl* RSPQL?
EventDecl –> 'NAMED EVENT' EventName (AbstractEvent | ComplexEvent)
AbstractEvent –> 'AS' DLDescription
ComplexEvent –> 'MATCH' (Modifier)? EventPattern (Guard)? (IFClause)?
EventPattern –> EventPattern EventOperator EventPattern | AbstractEvent | 'NOT' EventPattern
IFClause –> 'IF' '{' ('EVENT' AbstractEvent '{' BGP '}')* '}'
EventOperator –> 'AND' | 'OR' | 'SEQ'
Modifier –> 'EVERY' | 'FIRST' | 'LAST'
Guard –> 'WHITIN' Num '(' TIMEUNIT ')'
TIMEUNIT –> 's' | 'm' | 'h' | 'd'
EventName –> String
Num –> [0-9]+
NameSpace –> SPARQL PREFIX SYNTAX
DLDescription –> MANCHESTER SYNTAX
BGP –> SPARQL BGP SYNTAX
RSPQL –> RSP-QL SYNTAX
```

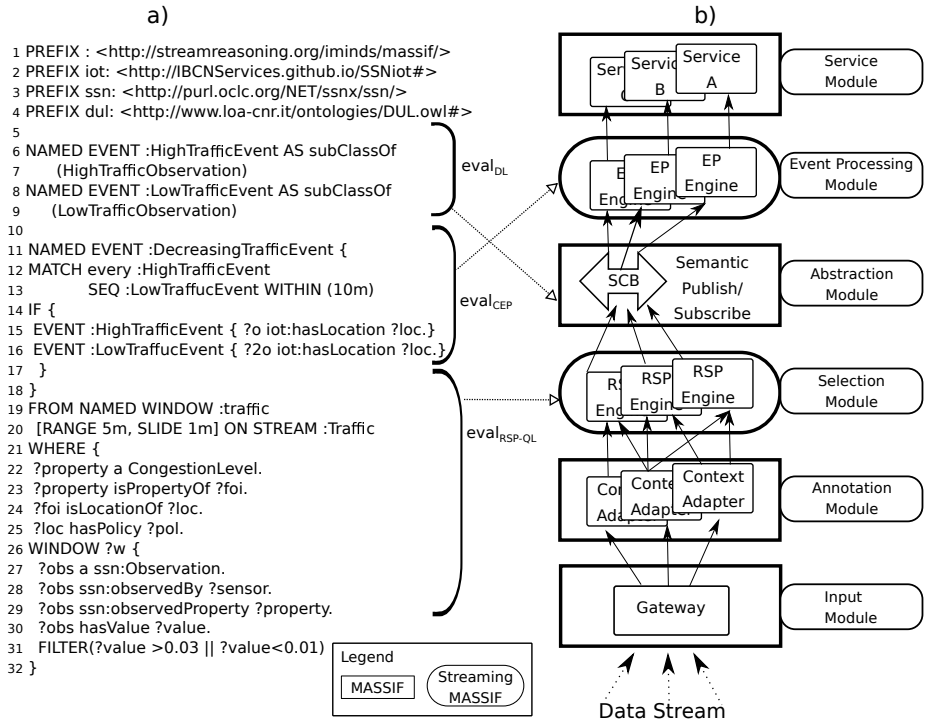

Fig. 6. a) Example of the Streaming MASSIF DSL. b) Streaming MASSIF architecture.

restricts observations to be after 3 o'clock in the afternoon.

Note that the SPARQL FILTER clause is optional. When the defined BGP does not match the underlying event, i.e. no results are returned, the event is filtered out and not considered in the complex event processing.

Example 6.2. *(cont'd) If we have an observation in the morning, e.g.:*
Observation(obsx),
...

```
1 NAMED EVENT :DecreasingTrafficEvent {
2 MATCH EVERY :HighTrafficEvent
3                 SEQ :LowTrafficEvent WITHIN (10m)
4 IF {
5  EVENT :HighTrafficEvent { ?o timeStamp ?time.
6           FILTER(hours(?time) > 15) }
7  EVENT :LowTrafficEvent { ?o2 timeStamp ?time2.
8           FILTER(hours(?time)>15) } }
9 }
```

Listing 3 DSL Event Clause

timeStamp(obsx,"2017-08-20T10:00:00"^^xsd:dateTime)

Executing the query from Example 6.1 does not produce any result because the observation was not made after 3 o'clock in the afternoon. Thus, it is dropped and not incorporated in the complex event processing.

6.2. Architecture

In the following, we describe the architecture of a system that implements our DSL and, realizes the cascading stream reasoning approach presented in Section 5.

Our work is based on the **MASSIF** platform, i.e. a layered, even-driven platform for data integration in the IoT domain [11] consisting of multiple modules. MASSIF facilitates the annotation of raw sensor data to semantic data and allows the development and deployment of modular semantic reasoning services which collaborate in order to allow scalable and efficient processing of the annotated data. Each one of the services fulfills a distinct reasoning task and operates on a different ontology model. The Semantic Communication Bus (SCB) facilitates collaboration between services. Services indicate in which types of data they are interested, referring to high-level ontology concepts, by registering filter rules, i.e., OWL axioms, on the SCB. The annotated data from the sensors or other sources and the derived data produced by the various services are pushed back on the SCB and forwarded to those services that have indicated interest in the published data. The SCB can coordinate the data on a high-level through the use of semantic reasoning.

Although MASSIF is an event-driven platform, it processes one event at a time and is, thus, not able to process streams nor capture temporal dependencies between events. However, its layered architecture and the ability to perform service composition over high-level concepts offer a good base to extend it into our cascading reasoning approach.

To realize our cascading stream reasoning approach, two addition modules have been added on the MASSIF platform, as depicted by the rounded blocks in Figure 6 b). We named the resulting platform Streaming MASSIF. Figure 6 also illustrates how the DSL targets each module. We now explain each module in more detail.

6.2.1. Selection Module

The **Selection Module** implements both the Stream Processing and the Continuous Information Integration Layer of the cascading reasoning approach and selects, through RSP, those parts of the RDF stream that are relevant. We utilized YASPER [42], i.e., an RSP engine recently developed, that fully implements RSP-QL [20] semantics and can consumes RSP-QL queries. YASPER, differently from C-SPARQL [9] or CQELS [26] consumes time-annotated graphs instead of time-annotated triples. The selected sub-graphs $G_\Omega(t)$, i.e. selected physical events e_{phy}, are forwarded to the next module. Note that multiple RSP engines can optionally run in parallel, for example to distribute the load of various queries. Furthermore, we utilize the Construct Quad [7] by the Jena Engine to separate the various selected events from each other. Otherwise, all the events are merged together and its not clear which triples belong to which event.

6.2.2. The Abstraction Module

The **Abstraction Module** implements the DL inference sub-layer. It receives the selected physical events from the *Selection Module* and infers them to abstracted events. The *Abstraction Module* consists of a semantic publish/subscribe mechanism and allows the subscription to abstracted events. Each service in the service module can subscribe to events by defining event descriptions in $\mathcal{E}$. When one of the physical events can be inferred to one of the descriptions in $\mathcal{E}$, the services that defined this description are looked-up and the abstracted event is forwarded to these services.

If an Event Processing clause is defined, it is first forwarded to the *Event Processing Module*.

Technically, the *Abstraction Module* operates on an OWL reasoner, i.e. the Hermit reasoner [38], which operates on $\mathcal{O} \cup \mathcal{E}$.[8] Each time physical events have been selected in the *Selection Module*, they are added to the ontology in the *Abstraction Module*. Through the use of reasoning, we check which inferred types

[7] https://jena.apache.org/documentation/query/construct-quad.html

[8] Note that due to the modularity of the platform, other reasoners can easily be plugged in.

of the individuals in the physical events are defined in $\mathcal{E}$. When the types are found, the abstracted events are constructed using the found types, the underlying physical event and the processing time. The abstracted event is then forwarded to those services that subscribed to the found types. Lastly, the events are removed from the ontology ABox. When new events have been selected by the underlying module, they are added to the ontology and the types of a new events can be checked.

6.2.3. The Event Processing Module

The **Event Processing Module** implements the temporal reasoning sub-layer. When an event processing clause has been defined, the *Event Processing Module* receives the abstracted events defined in the query. Each of the received abstracted events is checked if it matches an event pattern, through the use of the Esper CEP engine[9]. We choose Esper since it supports the declarative language EPL. Note that when multiple abstracted events are inserted at once, they are first ordered according to their timestamp. When filter restrictions have been defined, these restrictions are checked first before adding the event to the CEP engine. When a join-restriction has been detected (variables over multiple AbstractEvents with the same name), the bindings of the those variables are used within the CEP engine to perform the joins. When an event pattern matches, it is forwarded to the associated service.

6.2.4. The Remaining Modules

The MASSIF platform also consists of an **Input Module** that serves as the entry point of the platform and an **Annotation Module**, where raw data can be semantically annotated if necessary.

Finally, the **Service Module** receives the processed data and can perform additional analysis. Through the **Service Module** information need formulated using our DSL (Section 6.1) can be issued to Streaming Massif. Therefore, services can subscribe to all underlying modules with one query.

7. Evaluation

To evaluate Streaming MASSIF, we extended the City Bench benchmark [1] with expressive ontology concepts, as those described in Example 2.1 and 2.2. We also extended the ABox and added various offices located near the monitored streets, each with a set of random policies. Among these office policies are the possibility to start early, to stop early, having flexible work hours and the presence of childcare. To further increase the complexity we also added some complex roles which are used within the high and low traffic modeling, e.g.:

$$observedFeature \sqsubseteq observedProperty \circ isPropertyOf$$

For streaming the City Bench data, we utilized RSP Lab [10] and ran the streamers on a different node. The evaluation was conducted on a 16 core Intel Xeon E5520 @ 2.27GHz CPU with 12GB of RAM running on Ubuntu 16.04.

7.1. Test1: Increasing event rate

To test the scalability of the platform, we first artificially speed up the traffic streams to see how many events the platform can handle. Each stream in City Bench produces data every 5 minutes. We speed up the stream to produce multiple events per second. Figure 7 visualizes for each component, the number of processed events, and the processing time for a specific event rate. The RSP Query Time denotes the time to select the events within the window, the Abstraction Time denotes the time to abstract the received events from the RSP layer to high-level concepts and the CEP Time measures how long it takes for the pattern to match when the last event arrived that caused the pattern to match. On the x-axis we plotted the (rounded) actual event rate as they enter the platform. Note that each time the stream produces data, 5 observations are produced: the average speed, the vehicle count, the measured time, the estimated time and the congestion level. However, its not stated explicitly in the stream what kind of observation is transmitted. Integrating with background knowledge is, thus, required to filter out the congestion level observations. This is done in the RSP layer. We evaluated our results over 8 streams and calculated the averages over the first 120000 events. For easily calculating the processing time in each layer, we used a tumbling window (the sliding parameters is the same as the window width) of 2 seconds for each event rate. Using a tum-

[9] http://www.espertech.com/esper/

[10] https://github.com/streamreasoning/rsplab

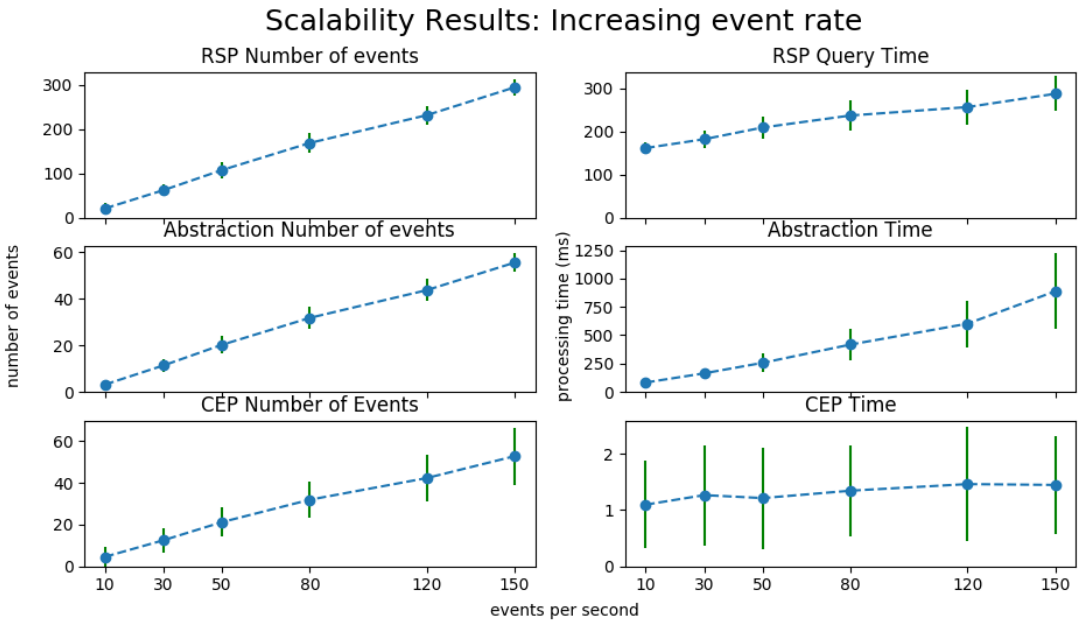

Fig. 7. The influence of increasing event rate on the platform.

bling window, each event only occurs onces and this simplifies the processing time calculations. To perform the evaluation, we used the example query from Figure 6 a).

From Figure 7, we can see that the biggest selection of events happens in the RSP layer, while less events are selected in the abstraction layer. Furthermore, the processing time in the Abstraction layer rises more quickly than in the other layers, what can be expected of a expressive reasoning process. However, we see that when abstracting even more than 50 events, the abstraction time is lower than 1 second. The total latency of the abstraction stays well below 2 seconds (the size of the window) and thus the system stays reactive even when processing 300 events per second.

7.2. *Test2: Increasing Window Size*

The performance of each layer is clearly dependent on the number of considered events. We investigated the processing time of each layer when the window size in the RSP layer increases. This forces the processing of an increasing number of events in each layer. Figure 8 visualizes the number of processed events and the processing time for each layer when the window size increases from 1 second to 100 seconds. We see a clear increase in the processing time of each layer. The Abstraction time increases exponentially, what can be expected of an expressive reasoning process. However, abstracting up to 100 events takes about 15seconds, still a lot faster then the time it takes for the window to slide.

7.3. *Test3: Comparison with MASSIF*

Since we extended the MASSIF platform to implement the adapted cascading reasoning vision, we also measured how fast the MASSIF platform could processes the event stream. Note however that the MASSIF platform needs to perform the abstraction on all the background data, consisting of all the information of all the sensors, the streets, the offices, etc. The whole background data contains more than 60000 statements. In the RSP layer we select the relevant portion from the stream, but also select the relevant data from the background knowledge. This eliminates the need for the Abstraction layer to contain the whole background knowledge, the TBox is most important there. Without this selection step, the abstraction of a single event in the MASSIF platform takes up to 20seconds. This shows that the layered approach is a lot more scalable.

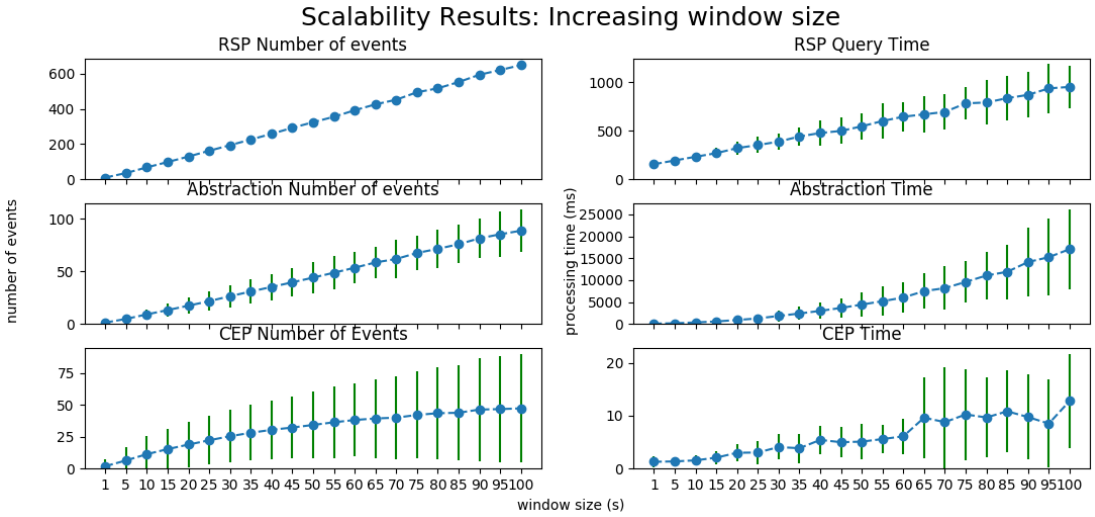

Fig. 8. The influence of increasing window size on the platform.

8. Related Work

In this section we elaborate on the related work in the literature and how our approach distinguishes from these previous approaches.

EP-SPARQL [4] is an RSP engine that focuses on event processing over basic graph patterns using Allen's Algebra. However, the reasoning expressivity is limited to RDFS and the definition of event patterns over basic graph patterns is complex compared to high-level events.

StreamRule [33] is a two-layered platform that combines RSP with ASP. However, there is no support for additional layers such as CEP and the two layers are not integrated in an unifying query language for easy usage.

Ali, et al.[2] proposed an IoT-enabled communication implemented on StreamRule that performs event-condition-actions rules in ASP. This allows to define action rules on specific events detected in the stream.

In the CityPulse project [37], the combination of RSP, CEP and expressive reasoning is presented. The combination of RSP and ASP (for the expressive reasoning) is supported by StreamRule. In order to handle CEP rules, the system can be extended programmatically. Streaming MASSIF, on the other hand, integrates CEP seamlessly by abstracting the events and using these abstractions for the event processing. This can all be defined in the query language, without the need to implement any code.

To the best of our knowledge, existing Semantic Complex Event Processing (SCEP) solutions focus on enriching events with semantic technologies.

Teymourian, et al. [41] proposed a knowledge-based CEP approach where events are enriched using external knowledge bases. The enrichment is defined using multiple SPARQL queries. However, the system is event-based, there is no support for streaming data and reasoning is only provided in the external knowledge base that is used for the event enrichment. Thus, no reasoning on the events themselves is possible.

Taylor, et al. [40] proposed a SCEP approach that allows to generalize query definition for CEP engines, enabling interoperability. This is done by defining the event processing operators as ontology concepts. These generalized queries can then be translated into a target language, for example in Event Processing Language (EPL). However, reasoning and streaming data is not taken into account.

Differently from SCEP our approach for cascading reasoning exploits CEP as an efficient alternative to perform temporal reasoning.

9. Discussion

In the evaluation, we can see that the *Abstraction Module* can easily become the bottleneck with a high number of events, therefore incremental reasoning techniques should be further researched. Currently, there exist no efficient expressive incremental reasoning techniques that also incorporate data prop-

Name	Stream Processing	Continuous Information Integration		Inference			Unifying QL
				Entailment	Selection	Mediation	
EP-SPARQL [4]	Etalis	RSP	Rewriting	RDFS & Allen Algebra	CEP	✓	✓
StreamRule [33]	CQELS	RSP	Annotation	ASP	Window	None	None
Ali et al.[2]	CQELS	RSP	Annotation	Action-Rules in ASP	Window	None	None
CityPulse [37]	CQELS	RSP	Annotation	ASP & CEP[1]	Window	None	None
Morph$_{stream}$ [12]	SneeQL	RSP	Rewriting	RDFS	Window	✓	✓
StreamQR [13]	CQELS	RSP	Rewriting	$\mathcal{ELHIO}$	Window	✓	✓
STARQL [35]	Exareem	RSP	Rewriting	DL-lite	Window	✓	✓
Streaming MASSIF	Yasper 1.0	RSP	Annotation	DL & Temporal[2]	Window	None	✓

Table 2
Overview of the related work and how they relate to our generalized Cascading Reasoning vision. 1: not integrated, 2: via CEP rules

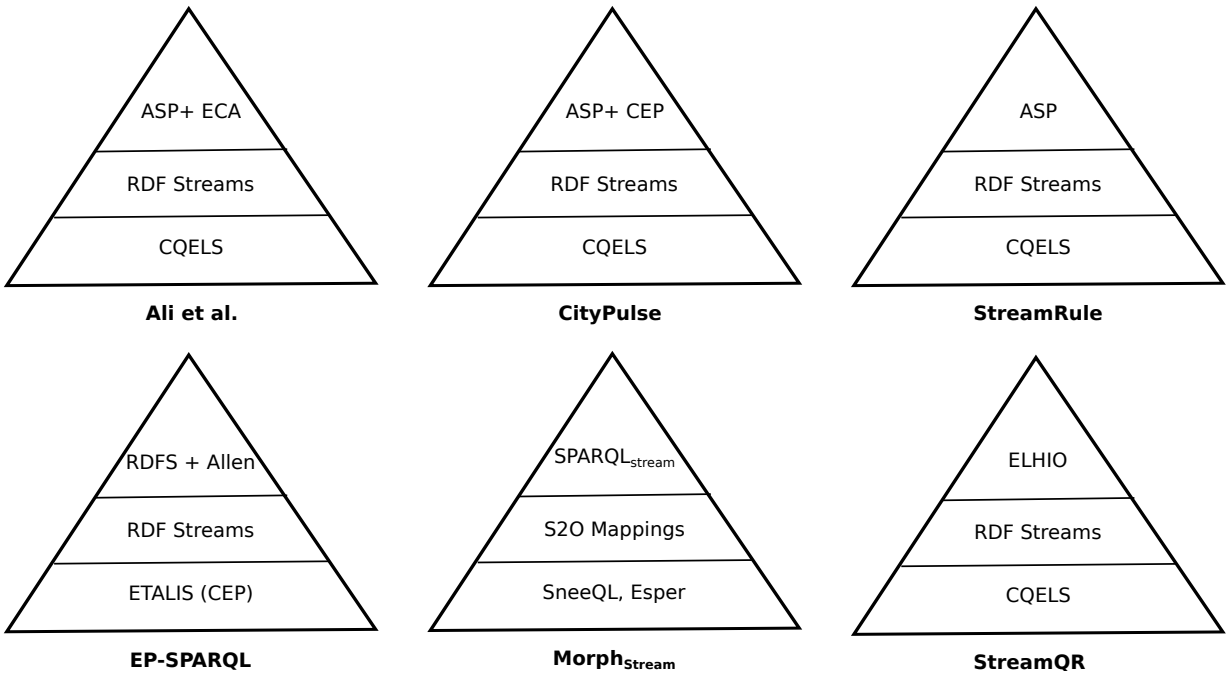

Fig. 9. Different approaches represented in the generalized Cascading Reasoning vision.

erty reasoning. We could easily perform the abstraction in parallel and load balance various events to increase the performance. This is possible in the cases that the events are independent of each other. When multiple physical events should be abstracted together, the query in the lower RSP layer could be adapted to link them together. This would allow to scale the abstraction module even more since the abstraction of a low number of events is still rather quick, i.e. less than half a second for 30 events.

Note that other reasoning approach exists, such as ASP, but we opted for DL since it's a web standard and widely adopted.

One of the limitations of the DSL is the fact that the user still manually needs to define a query over all the layers. The query mediation and query rewriting process is currently not researched yet.

Our Cascading Reasoning generalization can be used to compare different hierarchical stream reasoning approaches. It allows to distinguish the different parts of an approach, identifying the layers and the transition between them. Table 2 gives an overview of the various presented techniques, how they compare to Streaming MASSIF and how they fit in our generalized Cascading Reasoning vision. Furthermore, Fig-

ure 9 illustrates how the different approaches fill out the Cascading Reasoning pyramid.

We also introduce $\text{Morph}_{\text{Stream}}$ [12], StreamQR [13] and STARQL [35], at first sight, non-hierarchical RSP engines to demonstrate that these also fit our generalized Cascading Reasoning proposal. They can be split up in different techniques for Stream Processing, Continuous Information Integration and Inference.

10. Conclusion and Future Work

In this paper, we presented a renewed vision on Cascading Reasoning consisting of Stream Processing, Continuous Information Integration and Inference layers. We introduced a new layered approach based on this renewed vision, combining RSP, DL reasoning and CEP to enable expressive reasoning and event processing over high velocity streams. We described a query languages that combines these various layers, allowing easy querying of the whole reasoning stack without the need to write any code. Our approach can perform expressive reasoning and event processing over high velocity streams by selecting only the relevant events from the stream. However, when the RSP layer is not able to make this selection from the stream and huge number of events need to be abstracted, the platform might become slow.

In our future work, we try to tackle this issue by incorporating load balancing and caching techniques.

We will also investigate query mediation and rewriting to automatically construct the queries on the lower levels, based on the defined concepts on the highest layer. This will further simplify the query definition and bring Stream Reasoning closer to the masses.

Acknowledgment This research was funded by the VLAIO Strategic Fundamental Research (SBO) DiSSeCt.

References

[1] Muhammad Intizar Ali, Feng Gao, and Alessandra Mileo. Citybench: A configurable benchmark to evaluate rsp engines using smart city datasets. In *International Semantic Web Conference*, pages 374–389. Springer, 2015.

[2] Muhammad Intizar Ali, Naomi Ono, Mahedi Kaysar, Keith Griffin, and Alessandra Mileo. A Semantic Processing Framework for IoT-Enabled Communication Systems. *The Semantic Web-ISWC 2015*, pages 241–258, 2015.

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

[4] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. pages 635–644, 2011.

[5] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, 15(2):121–142, 2006.

[6] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[7] Marco Balduini, Irene Celino, Daniele Dell'Aglio, Emanuele Della Valle, Yi Huang, Tony Kyung-il Lee, Seon-Ho Kim, and Volker Tresp. Reality mining on micropost streams - deductive and inductive reasoning for personalized and location-based recommendations. *Semantic Web*, 5(5):341–356, 2014.

[8] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, Yi Huang, Volker Tresp, Achim Rettinger, and Hendrik Wermser. Deductive and inductive stream reasoning for semantic social media analytics. *IEEE Intelligent Systems*, 25(6):32–41, 2010.

[9] Davide Francesco Barbieri and et al. Querying RDF streams with C-SPARQL. *SIGMOD Record*, 2010.

[10] Payam Barnaghi, Wei Wang, Cory Henson, and Kerry Taylor. Semantics for the internet of things: early progress and back to the future. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 8(1):1–21, 2012.

[11] Pieter Bonte, Femke Ongenae, Femke De Backere, Jeroen Schaballie, Dörthe Arndt, Stijn Verstichel, Erik Mannens, Rik Van de Walle, and Filip De Turck. The massif platform: a modular and semantic platform for the development of flexible iot services. *KAIS*, 2016.

[12] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair J. G. Gray. *Enabling Ontology-Based Access to Streaming Data Sources*, pages 96–111. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[13] Jean-Paul Calbimonte, Jose Mora, and Oscar Corcho. Query rewriting in rdf stream processing. In *International Semantic Web Conference*, pages 486–502. Springer, 2016.

[14] Diego Calvanese, Elem Güzel Kalayci, Vladislav Ryzhikov, Guohui Xiao, and Michael Zakharyaschev. Metric temporal logic for ontology-based data access over log data. *CoRR*, 2017.

[15] Michael Compton, Payam Barnaghi, Luis Bermudez, RaúL GarcíA-Castro, Oscar Corcho, Simon Cox, John Graybeal, Manfred Hauswirth, Cory Henson, Arthur Herzog, et al. The ssn ontology of the w3c semantic sensor network incubator group. *Web semantics: science, services and agents on the World Wide Web*, 17:25–32, 2012.

[16] Gianpaolo Cugola and et al. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15, 2012.

[17] Gianpaolo Cugola and Alessandro Margara. TESLA: a formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS, Cambridge, United Kingdom*, 2010.

[18] Emanuele Della Valle and et al. Taming velocity and variety simultaneously in big data with stream reasoning: tutorial. In *Proceedings of DEBS*, 2016.

[19] Emanuele Della Valle, Stefan Schlobach, Markus Krötzsch, Alessandro Bozzon, Stefano Ceri, and Ian Horrocks. Order matters! harnessing a world of orderings for reasoning over massive data. *Semantic Web*, 4(2):219–231, 2013.

[20] Daniele Dell'Aglio, Emanuele Della Valle, Jean-Paul Calbimonte, and Óscar Corcho. RSP-QL semantics: A unifying query model to explain heterogeneity of RDF stream processing systems. *Int. J. Semantic Web Inf. Syst.*, 10(4):17–44, 2014.

[21] Daniele Dell'Aglio, Emanuele Della Valle, Frank van Harmelen, and Abraham Bernstein. Stream reasoning: A survey and outlook. *Data Science*, (Preprint):1–24, 2017.

[22] Anastasia Dimou, Miel Vander Sande, Pieter Colpaert, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In *Proceedings of the 7th Workshop on Linked Data on the Web*, April 2014.

[23] Benjamin N Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description logic programs: combining logic programs with description logic. In *Proceedings of the 12th international conference on World Wide Web*, pages 48–57. ACM, 2003.

[24] Roman Kontchakov and Michael Zakharyaschev. An introduction to description logics and query rewriting. In *Reasoning Web International Summer School*, pages 195–244. Springer, 2014.

[25] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[26] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *International Semantic Web Conference*, pages 370–388. Springer, 2011.

[27] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. *A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data*, pages 370–388. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[28] David Luckham. The power of events: An introduction to complex event processing in distributed enterprise systems. In *Rule Representation, Interchange and Reasoning on the Web, International Symposium, RuleML, Orlando, FL, USA*, 2008.

[29] Carsten Lutz, Frank Wolter, and Michael Zakharyashev. Temporal Description Logics: A Survey.

[30] Alessandro Margara, Jacopo Urbani, Frank Van Harmelen, and Henri Bal. Streaming the web: Reasoning over dynamic data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 25:24–44, 2014.

[31] Andrea Mauri, Jean-Paul Calbimonte, Daniele Dell'Aglio, Marco Balduini, Marco Brambilla, Emanuele Della Valle, and Karl Aberer. Triplewave: Spreading rdf streams on the web. In *International Semantic Web Conference*, pages 140–149. Springer, 2016.

[32] Alessandra Mileo. Web stream reasoning: From data streams to actionable knowledge. In *Reasoning Web International Summer School*, pages 75–87. Springer, 2015.

[33] Alessandra Mileo, Ahmed Abdelrahman, Sean Policarpio, and Manfred Hauswirth. Streamrule: a nonmonotonic stream reasoning system for the semantic web. In *International Conference on Web Reasoning and Rule Systems*, pages 247–252. Springer, 2013.

[34] Yavor Nenov and et al. Rdfox: A highly-scalable RDF store. In *ISWC 2015 , Proceedings, Part II*, pages 3–20, 2015.

[35] Özgür Lütfü Özçep, Ralf Möller, and Christian Neuenstadt. A stream-temporal query language for ontology based data access. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 183–194. Springer, 2014.

[36] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Context aware computing for the internet of things: A survey. *IEEE Communications Surveys & Tutorials*, 16(1):414–454, 2014.

[37] Dan Puiu, Payam Barnaghi, Ralf Tonjes, Daniel Kumper, Muhammad Intizar Ali, Alessandra Mileo, Josiane Xavier Parreira, Marten Fischer, Sefki Kolozali, Nazli Farajidavar, Feng Gao, Thorben Iggena, Thu-Le Pham, Cosmin-Septimiu Nechifor, Daniel Puschmann, and Joao Fernandes. CityPulse: Large Scale Data Analytics Framework for Smart Cities. *IEEE Access*, 4:1086–1108, 2016.

[38] Rob Shearer, Boris Motik, and Ian Horrocks. Hermit: A highly-efficient owl reasoner. In *OWLED*, volume 432, page 91, 2008.

[39] Heiner Stuckenschmidt and et al. Towards expressive stream reasoning. In *Semantic Challenges in Sensor Networks, 24.01. - 29.01.2010*, 2010.

[40] Kerry Taylor and et al. Ontology-driven complex event processing in heterogeneous sensor networks. In *The Semanic Web: Research and Applications - ESWC 2011, Proceedings, Part II*, pages 285–299, 2011.

[41] Kia Teymourian. *A Framework for Knowledge-Based Complex Event Processing*. PhD thesis, Free University of Berlin, 2014.

[42] Riccardo Tommasini and Emanuele Della Valle. Challenges & opportunities of rsp-ql implementations. 2017.

[43] Emanuele Della Valle, Daniele Dell'Aglio, and Alessandro Margara. Taming velocity and variety simultaneously in big data with stream reasoning: tutorial. In *DEBS*, pages 394–401. ACM, 2016.