Towards On-Demand Semantic Event Processing for Stream Reasoning

Daniel de Leng and Fredrik Heintz
Department of Computer and Information Science
Linköping University, Sweden
{daniel.de.leng, fredrik.heintz}@liu.se

***Abstract*—The ability to automatically, on-demand, apply pattern matching over streams of information to infer the occurrence of events is an important fusion functionality. Existing event detection approaches require explicit configuration of what events to detect and what streams to use as input. This paper discusses on-demand semantic event processing, and extends the semantic information integration approach used in the stream processing middleware framework DyKnow to incorporate this new feature. By supporting on-demand semantic event processing, systems can automatically configure what events to detect and what streams to use as input for the event detection. This can also include the detection of lower-level events as well as processing of streams. The semantic stream query language C-SPARQL is used to specify events, which can be seen as transformations over streams. Since semantic streams consist of RDF triples, we suggest a method to convert between RDF streams and DyKnow streams. DyKnow is integrated in the Robot Operating System (ROS) and used for example in collaborative unmanned aircraft systems missions.** [1]

I. Introduction

Modern fusion systems are incremental in nature. The information available is often in the form of sequences of time-stamped values, *streams*, and it is necessary to react to events in a timely manner. For the system to detect a high-level event it is usually necessary to combine and fuse information from diverse sources and on many different abstraction levels. This needs to be done incrementally due to the vast amounts of information that can be made available in a short time period.

The incremental reasoning over stream data is called *stream reasoning* [1], [2]. Stream reasoning is necessary to support important functionality such as situation awareness, execution monitoring, and planning [3]. Stream reasoning can be used to determine the truth value of logical formulas, or infer the occurrence of high-level events from low-level information. Event processing is a well-established research area with many existing approaches to detecting events in streams [4], [5]. The Semantic Web community is also working on stream reasoning over streams of time-stamped RDF triples [6]–[9].

Using existing event processing approaches it is necessary to specify exactly what events to detect and what inputs to use. In this work we present a solution to the problem of automatically configuring a system on-demand to detect a given high-level event. The solution is recursive in nature so that the detection of one event may trigger the detection of other events as well as other types of processing of streams. This allows the system to declare its interest in an event without explicitly configuring all the processing needed to detect instances of the event.

[1]This work is partially supported by grants from the National Graduate School in Computer Science, Sweden (CUGS), the Swedish Aeronautics Research Council (NFFP6), the Swedish Foundation for Strategic Research (SSF) project CUAS, the Swedish Research Council (VR) Linnaeus Center CADICS, the ELLIIT Excellence Center at Linköping-Lund for Information Technology, and the Center for Industrial Information Technology CENIIT.

Consider for example a UAV monitoring its speed in order to detect that it is flying too fast. Rather than subscribing directly to a speed sensor providing fluctuating values, it is much more convenient to declare an interest in occurrences of a high-speed event. This event could for example be defined as the average speed of the UAV over the last five seconds exceeds a maximum speed threshold.

In this paper, we present an extension to our earlier work on the semantic integration of streams and transformations over streams to also include event processing. Our contribution allows a system to automatically determine the necessary information for detecting a particular event. This could include the detection of other events that in turn require further information to be collected. This functionality is very useful both for autonomous systems and command and control applications which need to detect important events and react to them.

The remainder of this paper is organised as follows. In Section II, previous work towards semantic matching and DyKnow is covered. In Section III, we clarify what we mean with the term 'event', and in Section IV a method is proposed for the semantic integration of events. Finally, in Section V we describe how semantic matching can be used to support on-demand semantic event processing in the context of C-SPARQL and DyKnow, before concluding the paper in Section VI.

II. Previous Work

One major problem for high-level symbolic reasoning with low-level quantitative information is the difficulty associated with mapping information in streams to symbols. One way of achieving this mapping is by referring directly to the streams containing the desired information. However, this approach suffers from a number of draw-backs. When the number of streams grows this approach scales poorly. Furthermore, human error in the referencing of streams is a reasonable

concern, especially as the number of streams increases. The effects of a broken or incorrect mapping can be severe.

To address these problems we have developed a semantic matching approach using semantic web technologies [10]–[12]. We use an ontology to specify a common vocabulary with which both streams and available transformations over streams are annotated. Given an ontological concept, semantic matching returns a stream specification that can be used to construct a stream containing the desired information. A stream specification can be quite complex, involving the filtering, merging and synchronisation of numerous streams, as well as the performing of transformations on streams in order to produce higher-level information. Streams can therefore be semantically annotated by their creators and found based on this annotation. Additionally, if no stream containing the desired information can be found, the annotated transformations can be used to automatically generate a new stream containing the missing information. This functionality has been integrated in the stream-based knowledge processing middleware framework DyKnow [13]–[15].

DyKnow is tasked with handling and manipulating streams and was designed as a tool for bridging the sense-reasoning gap [16] that exists between crisp high-level symbolic knowledge and the noisy and incomplete quantitative low-level data originating for example from sensors. DyKnow has been integrated in the Robot Operating System (ROS) [17] and used for example in collaborative unmanned aircraft systems missions. Its high-level architecture is shown in Fig 1. It shows the three main components being the stream processing module, the semantic integration module and the stream reasoning module. The stream processing module contains functionality for manipulating streams through filtering, merging or synchronisation. It also contains a library of transformations that can be performed on streams to generate new streams, and it can instantiate these transformations when needed. The semantic integration module maintains the DyKnow ontology that describes a common language with which streams and transformation can be semantically annotated. The stream reasoning engine can evaluate metric temporal logic (MTL) formulas over streams, where MTL is first-order logic extended with temporal operators. Because the focus in this paper is on stream reasoning in the context of event processing, this last module is outside the scope of this paper.

In previous work [10], we introduced semantic matching as a way to find the relevant streams given a metric temporal logic formula. Semantic matching was later extended to also handle the case of indirectly-available streams [11], [12] by extending the semantic integration to also include transformations. Streams and transformations are annotated with the concepts in a common language represented by an ontology using the Stream Semantics Language (SSL). The metric temporal logic formulas use the common language to refer to features, objects and sorts. Here a *feature* represents a property or relation for which the value may change over time. These properties and relations describe *objects* in the environment. An object can for example be a UAV or a road. Finally, *sorts* are collections

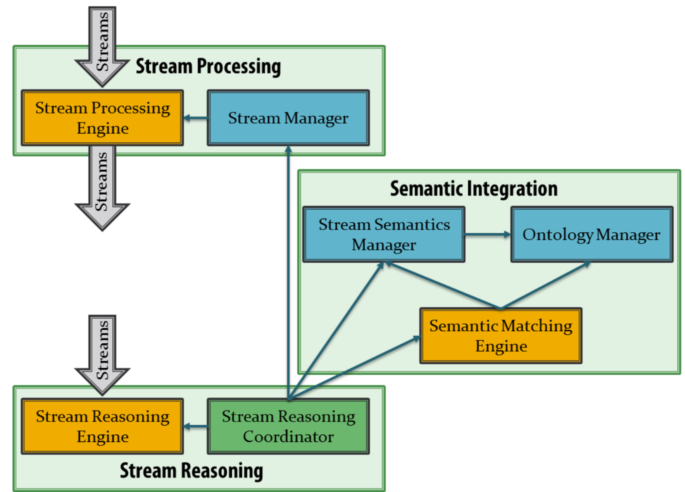

Fig. 1: High-level overview of DyKnow

of objects based on some commonality, e.g. being a vehicle. Ontologies allow us to specify hierarchies of concepts. We can thus represent knowledge such as a UAV being a kind of vehicle. The same is true for features. Aside from representing hierarchies, an ontology can also be used to encode other static knowledge. One example presented in previous work [11] is the encoding of unit conversion information.

The extended semantic integration approach is in line with recent work on semantic modeling of sensors [18], [19] and on semantic annotation of observations for the Semantic Sensor Web [20]–[22]. An interested approach is a publish/subscribe model for a sensor network based on semantic matching [20]. The matching is done by creating an ontology for each sensor based on its characteristics and an ontology for the requested service. If the sensor and service ontologies align, then the sensor provides relevant data for the service. This is a complex approach which requires significant semantic modeling and reasoning to match sensors to services. Our approach is more direct and avoids most of the overhead. The presented approach also bears some similarity to work by Whitehouse et al. [23] as both use stream-based reasoning and are inspired by semantic web services, and RoboEarth [24], which made it possible for robots to share plans and models by utilising semantic web technologies, making this knowledge available on demand from a centralised server.

III. Event Processing for Stream Reasoning

As has recently been pointed out by Ziafati et al. [25]: “DyKnow does not support event-processing, dynamic subscription and on-demand querying mechanisms presented in our work.” This lack of explicit event processing support is a potential shortcoming that we wish to address. When we talk about events, there are many equally valid interpretations we could use. Ziafati et al. [25] refer to an event as “a sensory information item such as the robot’s location at a time or the types and positions of objects recognised from a picture.” An alternative and more general definition [26] states

that an event is "anything that happens, or is contemplated as happening." Considering that DyKnow allows for stream-generating transformations over streams, does this constitute event processing support? One may reasonably argue that every piece of information being generated constitutes an event, or take a more coarse approach to events. Clearly, it is important to clarify what we mean when discussing events in this paper. Our interpretation for the purpose of this paper makes use of the implementation-independent *stream space* concept.

Definition 1 (Stream): A *stream* is a sequence of samples $\langle t_a, t_v, \vec{v} \rangle$, where t_a denotes the time at which the sample became available (available time), t_v denotes the time for which the sample is relevant (valid time), and $\vec{v}$ denotes the values contained within the sample. A total ordering $<$ is assumed over the samples in a given stream for the available time t_a.

Definition 2 (Stream space): A *stream space* $S \subseteq S^*$ is a collection of named streams, where S^* denotes the set of all possible named streams called the *stream universe*.

The execution of a stream processing system can be described by a sequence of stream spaces $S^{t_0} \Rightarrow S^{t_1} \Rightarrow \ldots \Rightarrow S^{t_n}$. Here S^t represents the streams at time-point t which means that every sample in every stream in S^t must have an available time $t_a \leq t$. In this paper, an event denotes a set of stream space transition sequences. If one such transition sequence occurs, the event is said to occur.

Definition 3 (Event): An *event* is described by a set of stream space transition sequences, $S^{t_0} \Rightarrow S^{t_1} \Rightarrow \ldots \Rightarrow S^{t_n}$ where $t_i < t_{i+1}$. If any transition sequence in the set occurs, the event is said to occur.

An event occurrence is usually associated with additional information that may for example relate to (some of the) values that fulfilled a set of constraints or a pattern from which the set of stream space transitions is derived. By performing event checks at a specified interval, an *event stream* is generated for event e by a *stream generator*. Every *event instance* in this event stream represents the fact that the constraints for e are fulfilled and thus e occurred. Such an event stream can then be used for more complex events. The suggested definitions have the desirable property of being able to model events according to both event definitions described earlier by [25] and [26], so we reject neither.

The set of stream space transition sequences gives a complete description of the pattern or constraints describing an event. However, this complete description is often hard or even impossible to provide due to the size of the resulting corresponding set of stream space transition sequences. Thankfully there exist many languages that let the user specify these constraints succinctly, allowing us to specify these sets of stream space transition sequences indirectly. Due to space limitations we only consider a few of them.

A. Overview of C-SPARQL

Continuous SPARQL (C-SPARQL) [9], [27], [28] is an extension of SPARQL [29] for RDF data streams as part of the LarKC project [30]. It allows for aggregation over windows, which is not supported by SPARQL. The authors of C-SPARQL characterise stream reasoning as "logical reasoning in real time on gigantic and inevitably noisy data streams in order to support the decision process of extremely large numbers of concurrent users." [2]. C-SPARQL facilitates stream reasoning in this definition.

Due to its roots in SPARQL, C-SPARQL is closely tied to ontologies. The RDF data streams used consist of time-stamped RDF triples, i.e. $(\langle subj_i, pred_i, obj_i \rangle, \tau_i)$. It combines these dynamic streams with static ontologies. The language is popular in the semantic web community and has been used for example with Twitter feeds.

B. Overview of EP-SPARQL in ETALIS

ETALIS [7] is an open source system that can efficiently detect complex events over data streams. Its focus is on semantic complex event processing. An event is defined as "something that occurs, happens or changes the current state of affairs." Thus, in the context of ETALIS, both low-level sensor information streams as well as high-level complex event patterns can be considered events, which fits our own definition for this paper.

ETALIS implements two languages. The ETALIS Language for Events (ELE) is a language for constructing complex event patterns. It supports a number of causal and temporal relations between events and is used to define Prolog-style rules. The second language is called Event Processing SPARQL (EP-SPARQL) [8], which is an extension to SPARQL for the purpose of handling streams in Semantic Web applications.

EP-SPARQL extends SPARQL by adding causal and temporal expressivity. For example, it supports methods such as `getDURATION()` which can be used to filter based on the duration of an event pattern. This is not supported by C-SPARQL. In contrast, C-SPARQL supports windows and aggregation, which are not explicitly supported in EP-SPARQL.

C. Overview of SASE

SASE [4], [5], [31], is a complex event processing language designed for the context of RFID event streams. It was later extended with the Kleene closure to *SASE+* [32], [33]. Unlike C-SPARQL and EP-SPARQL, SASE does not explicitly support Semantic Web streams.

SASE was designed based on perceived shortcomings in existing languages for stream processing. Some examples are the handling of non-occurrence of events and support for sliding windows. Like ETALIS, it considers event streams for processing. An event in the context of SASE is defined as "an instantaneous and atomic (i.e., happens completely or not at all) occurrence of interest at a point in time." SASE supports causal relations, including those that consider the non-occurrence of events. While SASE does not consider Semantic Web technologies, it may be possible to provide semantic annotations to relate symbols in SASE statements to an ontology.

IV. Semantic Integration of Events

While semantic integration techniques have in previous work been considered for streams and transformations, this is not the case for events. Semantically integrated events are events for which the semantic description is known, be it explicitly through the actual definition of the event or implicitly through the usage of a semantic annotation. Event processing can be regarded as a kind of transformation over streams that results in a stream describing event instances. It is already possible to semantically integrate stream transformations [11] by relating their input and output features to the ontology. This makes it possible for a program to interpret what information is necessary in order to perform a transformation, and what the resulting information is. A similar approach can be used for event processing.

In this section, we discuss the semantic integration of events using the C-SPARQL language as an example. C-SPARQL is of special interest due to its basis in semantic web technologies. It allows us to both provide explicit and implicit semantic meaning to event processors. Our goal is to provide a general method for semantic integration that allows a program to find the relevant information necessary for event processing, which could be adopted for other languages as well. To this end, we first discuss how to semantically annotate events using C-SPARQL. This is then followed up by considering abstract events, or *event templates*.

A. Semantic Annotation of Events

The semantic annotation of streams or transformations makes it possible to use the common language defined by an ontology when describing these streams or transformations. This allows a system, given some semantic concept, to identify which streams or transformations are semantically annotated with this concept. This process is called *semantic matching*. In order to expose events for the purpose of semantic matching, we can provide a similar annotation using the ontology as a common language. In the case of C-SPARQL this is especially interesting because the semantic annotation is inherent to the language itself.

Before discussing how to use semantic annotations for C-SPARQL statements, we first take a look at ontologies. Recall that we consider features, sorts and objects. In an ontology, we can distinguish between concepts and individuals. Concepts represent sorts whereas objects are represented by individuals. Unary features can be represented using properties for individuals. Some of these properties, such as colour, can be stored in the ontology as they remain relatively static. Others, like speed, are represented by streams instead as they can rapidly change as time passes. An example of a feature would be speed, whereas a sort might be UAV and an object of that sort might be uav1. In the DyKnow ontology, speed is described with the URI http://www.ida.liu.se/dyknow/speed. For the purpose of semantic annotations, however, we assume a common namespace. Consequently the short-hand notation is used instead of the URI.

Listing 1: Formal grammar for SSL

```
decl           : stream_decl | source_decl | compunit_decl ;
stream_decl    : 'stream' NAME 'contains' feature_list
                 for_part? ;
source_decl    : 'source' NAME 'provides' field_feature ;
compunit_decl  : 'compunit' NAME 'transforms'
                 field_features 'to' field_feature ;
field_features : field_feature (COMMA field_feature)* ;
field_feature  : NAME COLON NAME unit_list? ;
feature_list   : feature (COMMA feature)* ;
feature        : NAME LP feature_args RP EQ
                 NAME unit_list? ;
feature_args   : feature_arg (COMMA feature_arg)* ;
feature_arg    : NAME alias? ;
for_part       : 'for' entity (COMMA entity)* ;
entity         : sort | object ;
unit_list      : ( OPEN unit (COMMA unit)* CLOSE )
               | 'no_unit' ;
unit           : NAME power? ;
power          : ('+' | '-')? NUMBER ;
alias          : 'as' NAME ;
object         : entity_full ;
sort           : sort_type entity_full ;
entity_full    : NAME EQ NAME ;
sort_type      : 'some' | 'every' ;
NAME           : ( 'a'..'z'|'A'..'Z'|'0'..'9')+ ;
NUMBER         : ('0'..'9')+ ;
```

Listing 2: Example SSL statements for streams

```
stream s1 contains altitude(uav1) = alt [ft]
stream s2 contains altitude(uav1) = alt [ft] for uav1 = id
stream s3 contains speed(UAV) = spd [mi.h-1]
    for every UAV = id
stream s4 contains xydist(UAV as arg1, UAV as arg2) = dist
    for every arg1 = id1, arg2 = id2
```

We can semantically annotate streams and transformations using the Semantic Specification Language (SSL) described in [12]. SSL annotations describe the features contained within streams using an ontology serving as a common language. Similarly, transformations can be described by their input and output features. The advantage of semantically annotating streams is that it becomes possible to find streams based on the features they contain. The grammar is shown in Listing 1, followed by some examples in Listing 2.

The first example statement states that stream s1 contains information on the altitude of object uav1 in the field named 'alt'. This is different from the second statement, which states that stream s2 contains the same information as stream s1 with the difference that this is only the case when the field named 'id' has the value 'uav1'. The last two statements make use of sorts that are specified in the object ontology. Stream s3 contains information on the speed for all objects in sort UAV, where the speed information is presented in the field named 'spd' for the UAV object referred to in the field named 'id'. We can see a similar construct in the semantic stream specification for stream s4. However, here we encounter some ambiguity as the sort UAV occurs twice. This is resolved by using an alias, in this case 'arg1' and 'arg2'. Note that the annotations include units of measurement so that unit conversion can be applied if necessary. They are represented as multiplications of units of measurement associated with their powers.

The combination of the semantic matching procedure, ontology, and semantic annotations of streams form foundation for automatically finding streams based on features of interest. To integrate SASE as an event processing language, the next step would be to provide a semantic annotation of the terms used in that language. An alternative would be to assume every term directly corresponds to a feature in the ontology. In the case of C-SPARQL, most of this work is already done for us. C-SPARQL is based on semantic web technologies, and therefore considers RDF triples where the object, predicate and subject are URIs. Consider the following example.

Listing 3: Example C-SPARQL statement

```
REGISTER STREAM HighSpeedEvent COMPUTE EVERY 1s AS
PREFIX dyknow: <http://www.ida.liu.se/dyknow/>
CONSTRUCT {?uav dyknow:altitude ?avgSpeed}
FROM <http://www.ida.liu.se/dyknow/ontology.rdf>
FROM STREAM <http://www.ida.liu.se/dyknow/s59.trdf>
    [RANGE 5s STEP 1s]
WHERE {
    ?uav a dyknow:UAV .
    ?uav dyknow:speed ?spd .
}
AGGREGATE {(?avgSpeed, AVG, {?spd} )
    FILTER (?avgSpeed > 100)}
```

Listing 3 filters altitudes pertaining to UAV objects for which the average speed over the provided window is a value greater than 100. The result is a stream of RDF triples at one-second intervals. From the statement itself, the desired information can be inferred. The speed feature assumes as its subject ?uav. From the statement it can also be inferred that ?uav is intended to be of sort UAV. When the query is executed the resulting stream consists of UAV objects and their corresponding average speed over a five-second window.

When we analyse the C-SPARQL statement more carefully, we can observe that the statement explicitly mentions the URI of the stream of interest. In C-SPARQL the user has to explicitly state which stream contains the relevant information. This could potentially result in user errors and scales poorly with higher numbers of streams. By using our semantic matching this is done automatically. Indeed, the arguments for the semantic annotation of streams hold for queries as well. For a system to automatically be able to find the relevant streams of information in order to execute this C-SPARQL query, some kind of semantic annotation can be used. In the case of C-SPARQL queries, the queries themselves refer directly to ontological concepts and properties. The ability to find the necessary streams based on this information inherent to C-SPARQL queries removes the need to manually specify which streams are of relevance.

Consider the following approaches. Given a stream processing framework such as DyKnow where streams are annotated with the features they describe, the naive approach would be to let a C-SPARQL query engine subscribe to all streams. The query engine would be able to filter the information it needs in order to perform the query, albeit at a low throughput due to the potentially massive amounts of irrelevant data. A better approach would be to consider the feature annotations of the stream identifiers in the current stream space. In the case of Listing 3, we are only interested in the dyknow:speed feature. The namespace dyknow: indicates that this is a feature known in the DyKnow ontology, which specifies the common language used to annotate streams in the DyKnow framework. Features that do not have this namespace must therefore be part of static ontologies rather than streams. By filtering based on this annotation, DyKnow can then automatically create a stream containing only speed feature information. An even better approach, outside the scope of this paper, is to also consider the predicate object. For example, in the above C-SPARQL statement the feature speed references UAV objects rather than all Thing objects. This would further enhance performance.

The proposed method makes it possible to infer from a C-SPARQL query the streams containing relevant information for query execution. This takes away the need to explicitly specify which streams are considered during this process. While it is still possible to specify this information using the FROM STREAM keywords, this is no longer required.

B. Event Templates

An *event template* can be regarded as an abstract or incomplete event. In Listing 3, we saw a C-SPARQL statement that generated an event instance every time a UAV had an average speed value greater than 100 over the five-second window. Once provided, the query is executed. In the case of events, we are interested in storing event declarations for on-demand execution. An example is provided in Listing 4. Its intended meaning is as follows. If an object of the type specified by the argument ?type has an average speed value that is greater than the threshold for the provided window, then the HighSpeedEvent has occurred and both the object and its average speed are reported.

The idea behind event templates is that an event can be customised while retaining a reusable component. In the example provided, it does not matter what the type of ?o is when it comes to determining whether the HighSpeedEvent occurred. It is possible to request dyknow:HighSpeedEvent(?o, dyknow:Thing, "100"^^xsd:integer) if we want to generate a stream containing HighSpeedEvent instances over every sort. An identical approach can be taken when considering predicates. Since the event templates are described by the ontology, they are part of the common language and can be referenced as a result.

Listing 4: Example event template

```
PREFIX dyknow: <http://www.ida.liu.se/dyknow/>
REGISTER TEMPLATE dyknow:HighSpeedEvent(?o, ?type,
    ?threshold) COMPUTE EVERY 1s AS
CONSTRUCT {?o dyknow:speed ?avgSpeed}
FROM <http://www.ida.liu.se/dyknow/ontology.rdf>
WHERE {
    ?o a ?type
    ?o dyknow:speed ?spd .
}
AGGREGATE {(?avgSpeed, AVG, {?spd} )
    FILTER (?avgSpeed > ?threshold)}
WINDOW [RANGE 5s STEP 1s]
```

Event templates for C-SPARQL can be regarded as abstract queries from which well-formed C-SPARQL queries can be derived. This is done by filling in the arguments provided for an event template, preparing a stream containing relevant information, and filling this into the query by using the FROM STREAM keywords and utilising the WINDOW specification from the event template. The resulting C-SPARQL query can then be understood by a C-SPARQL engine.

The combination of the semantic integration of events with the ability to describe event templates makes it possible for a machine or human to request a stream of events according to a previously stored event template. In the next section, we discuss how this can be used for on-demand event processing.

V. On-Demand Event Processing

We have identified and suggested methods for the various components that, when combined, support on-demand semantic event processing. By supporting on-demand processing in the context of event processing, a system is able to automatically find and prepare the necessary data and transformations to produce a desired event stream. The intent was to provide methods general enough to be applicable to any stream processing framework adhering to certain criteria. In this section, we provide a case study specific to the DyKnow framework in the context of the scenario mentioned in the beginning of this paper.

A. Converting Between RDF Streams and Streams in DyKnow

There exists a disconnect between RDF streams and streams in the context of DyKnow. Semantic streams are supported by C-SPARQL and contain time-stamped RDF triples, i.e. $(\langle subj_i, pred_i, obj_i \rangle, \tau_i)$ for time-point τ. In contrast, streams in DyKnow contain samples consisting of an available time t_a, a valid time t_v, and a vector of values $\vec{v}$, i.e. $\langle t_a, t_v, \vec{v} \rangle$. Whereas the semantic annotation of RDF streams is inherent to the RDF triples in the stream, DyKnow streams can contain any value and are optionally semantically annotated with the features contained in the streams. Concretely, RDF streams are semantically annotated on the sample level, whereas DyKnow streams are semantically annotated on the stream level. This leads to some interesting consequences.

In the case of RDF streams, the semantic annotation is inherent to the RDF triples. This is efficient since it requires no additional annotation. Streams in DyKnow do need this additional annotation, but this allows for them to be described at a stream level. For example, a stream in DyKnow can be annotated to state that the stream contains the speed feature for every UAV object. This is not possible for RDF streams without adding a meta-level annotation. Additionally, we can state that the speed feature in a stream in DyKnow assumes km/h as its unit of measurement. In RDF triples this would involve a fourth value, although this may be resolved by adding new data types for every unit of measurement. One advantage of RDF streams aside from the inherent semantic annotation is that the stream may consist of various RDF triples with different predicates without the need for changing its semantic annotation. In DyKnow, this can be resolved by keeping separate streams and synchronising them when necessary.

Listing 5: Example SSL statements for streams, continued

```
stream s5 contains altitude(UAV) = alt ft,
    speed(UAV) = spd [mi.h−1] for every UAV = id
stream s6 contains altitude(uav1) = alt ft,
    speed(uav1) = spd [mi.h−1] for uav1 = id
stream s7 contains altitude(uav1) = alt ft,
    speed(uav1) = spd [mi.h−1]
```

It is clear that there exists some overlap between RDF streams and streams in DyKnow. It is indeed possible to convert from streams in DyKnow to RDF streams by utilising the semantic annotations, and vice-versa. As an example, consider a stream s5 with a semantic annotation specified in Listing 5. According to the semantic specification, this stream describes the altitude and speed features for all UAV objects in feet and miles per hour respectively. Streams s6 and s7 describe these features for a single UAV called uav1. The difference between the two is that every sample in s7 contains information on uav1 whereas this is only the case for samples in s6 if their corresponding 'id' value is uav1.

In all of the above cases, the intended result consists of triples with predicates dyknow:altitude and dyknow:speed. In the case of stream s5, the subject for a sample is stored in the 'id' field. This refers to an object in the DyKnow ontology, and can thus easily be converted to the appropriate URI. The type information of a feature is desribed by the ontology using the appropriate datatype property description. The time-points associated with the output RDF triples are the available time t_a, and the value acting as object in an RDF triple is the value contained within the stream in DyKnow. A resulting triple can for example look like $(\langle obj_i, pred_i, subj_i \rangle, \tau_i)$, where obj_i is dyknow:uav1, $pred_i$ is dyknow:altitude or dyknow:speed, and obj_i is "100"^^xsd:integer for $\tau = t_a$. A similar approach can be taken for the case of stream s6 by only considering those samples where 'id' is uav1. For stream s7 there is no explicit mention of uav1 in any of the values, but uav1 is inferred as object for the associated RDF triples through the semantic annotation.

The decribed method converts streams in DyKnow to an RDF stream that can be used by a C-SPARQL engine to execute queries. Such a continuous query can produce result tables or a new RDF stream [28]. In order to use this new stream in the DyKnow framework, it needs to be converted to a stream in the context of DyKnow. For example, consider again the C-SPARQL query in Listing 3 where we were interested in all UAV objects with an average speed of over 100. The result of this query is an RDF stream containing RDF triples ?uav dyknow:speed ?avgSpeed. In order to convert these triples to a format used by streams in DyKnow, we can use the 'id' value to refer to the RDF subject. A second value 'speed' can then be assigned the RDF object ?avgSpeed. A similar approach can be used when handling the case of tables produced by C-SPARQL queries.

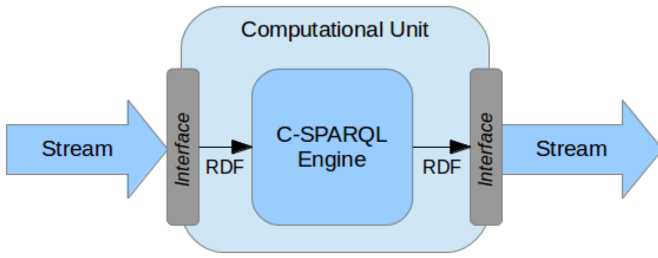

Fig. 2: C-SPARQL computational unit

With the proposed methods for converting between RDF streams and streams in DyKnow it is now possible to use a C-SPARQL engine to execute queries over information contained by semantically annotated streams in DyKnow. The resulting RDF streams can be converted to streams in the context of DyKnow so they can be used elsewhere.

B. Integrating C-SPARQL with Event Templates

Recall that we can consider event processors to be a type of stream generating transformations over streams. In the context of DyKnow, these transformations can be instantiated as *computational units*. If we consider a C-SPARQL engine to be a computational unit within DyKnow, we can leverage its ability to do event processing. The problem is then to integrate the engine so that it can handle streams in this framework. Fig 2 shows a diagram describing the integration architecture.

In the diagram the computational unit is represented as the light-coloured outer box with rounded edges. The arrows on either side represent the input and output streams associated with the computational unit. Both arrows are connected to thin bars representing interfaces. These interfaces convert between RDF streams and streams in the context of DyKnow using the approach discussed earlier. Internal to the computational unit there are arrows connecting the interfaces to the dark-coloured inner box with rounded edges, representing a C-SPARQL engine taking and producing data in the form of RDF streams.

In order to generate event streams on demand, a user or program first needs to request a desired event type. Given such an event type, DyKnow will access its ontology to determine whether it knows the desired event type. Recall that event templates are associated with event concepts in the ontology and that the templates are stored as a property of such a concept. Therefore, in order for DyKnow to understand the desired event, it needs to exist in its ontology.

If the desired event type is known, DyKnow can proceed to fetch the event template from the ontology and convert it to a C-SPARQL query by using the provided arguments, if any. For example, a HighSpeedEvent(?o, ?type, ?threshold) can be instantiated as HighSpeedEvent(dyknow:uav1, dyknow:UAV, ”100”^^xsd:integer) iff dyknow:HighSpeedEvent exists in the ontology. The resulting C-SPARQL query can then be used to determine which streams contain the desired features necessary for executing the query. Semantic matching is used to find these streams, after which DyKnow's stream processing capabilities are used to synchronise them. A C-SPARQL computational unit is subsequently instantiated and subscribes to the desired streams. The conversion interfaces convert the stream samples to RDF streams, which are used by the C-SPARQL engine to execute the provided C-SPARQL query. The results are then converted to a stream in the context of DyKnow so that they can be used elsewhere.

C. Scenario Revisited

Recall the scenario where a UAV is interested in events of type HighSpeedEvent concerning itself at a speed exceeding 100 km/h. It first checks its ontology to determine whether the HighSpeedEvent concept exists. If so, it fetches the associated event template from the ontology. The event template is represented with a arguments: HighSpeedEvent(?obj, ?type, ?threshold). Because our UAV is interested in its own high speed events with a threshold of 100, it can fill in these values as HighSpeedEvent(dyknow:uav1, dyknow:UAV, ”100”^^xsd:integer). From the resulting C-SPARQL event template, DyKnow is able to deduce that streams of relevance are those that contain dyknow:speed information specific to object dyknow:uav1. It uses its semantic matching functionality to find, leveraging semantic annotations, some matching stream containing this information. However, in this scenario it is unable to find such a stream. Thankfully the semantic matching procedure is able to find an alternative by constructing a new stream with speed information for uav1 by using an existing coordinate stream with coordinates of all UAV objects. It filters this stream for coordinate information on uav1. It then instantiates a computational unit from a transformation stored in its library that transforms coordinate features to speed features. The new coordinate stream is used as input for the newly created computational unit, producing a speed stream called dyknow_reserved35 concerning uav1 as its output.

With the speed information now available, DyKnow uses the C-SPARQL event template to construct a C-SPARQL query, filling in stream dyknow_reserved35 as input stream and automatically assigning stream dyknow_reserved36 as output stream. This yields a valid C-SPARQL query. DyKnow then instantiates a computational unit running a C-SPARQL engine. It provides the C-SPARQL query to the computational unit, which can then subscribe to stream dyknow_reserved35 and prepare to publish over stream dyknow_reserved36. Every sample that arrives over stream dyknow_reserved35 is converted to RDF triples by the computational unit's conversion interface. The RDF triples produced by the C-SPARQL engine are similarly converted to a format that can be handled by the DyKnow stream dyknow_reserved36.

The resulting event stream is used to send a sample every time the pattern of uav1 exceeding a speed of 100 km/h is detected. It can be used by other computational units within DyKnow for further processing. Where in this relatively simple scenario we used the on-demand functionality to find and generate the necessary sensor streams and transformations,

the same can be done with event streams, as all streams are conceptually the same.

VI. Conclusions and Future Work

In this paper, we have proposed an approach to on-demand semantic event processing for stream reasoning. Given a specification of a high-level event the approach will infer the processing needed to detect instances of the event. To achieve this the approach extends the semantic matching functionality used in the DyKnow middleware to handle events. To illustrate the approach we use C-SPARQL as a concrete event processing language. From a C-SPARQL event definition the extended semantic matching functionality generates a stream processing specification that DyKnow can execute. The result of the stream processing is a stream of RDF triples over which the C-SPARQL query can be evaluated.

This is an important functionality since it allows a system to automatically produce high-level event streams by transforming and fusing low-level information. The approach is recursive in nature so requesting the detection of one event may trigger the detection of other events as well as other types of processing of streams. This allows the system to declare its interest in an event without explicitly configuring all the processing needed on the many different abstraction levels to detect instances of the event.

The suggested approach is general and can be extended to other event processing languages such as SASE and EP-SPARQL. Another interesting area for future work is how to utilise events as context for stream reasoning in for example metric temporal logic, or use them for introspection by detecting changes in the stream space when e.g. a new stream becomes available.

References

[1] F. Heintz, J. Kvarnström, and P. Doherty, "Stream reasoning in DyKnow: A knowledge processing middleware system," in *Proc. 1st Intl Workshop Stream Reasoning*, 2009.

[2] D. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus, "Stream reasoning: Where we got so far," in *Proc. 4th workshop on new forms of reasoning for the Semantic Web: Scalable & dynamic*, 2010.

[3] F. Heintz, J. Kvarnström, and P. Doherty, "Stream-based middleware support for autonomous systems," in *Proc. ECAI*, 2010.

[4] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams," in *In SIGMOD*, 2006.

[5] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, "Efficient pattern matching over event streams," in *Proc. ACM SIGMOD international conference on Management of data*, 2008.

[6] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific American*, 2001.

[7] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic, "Stream reasoning and complex event processing in ETALIS," *Semantic Web Journal*, 2010.

[8] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic, "EP-SPARQL: a unified language for event processing and stream reasoning," in *Proc. WWW*, 2011.

[9] D. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus, "C-SPARQL: SPARQL for continuous querying," in *Proc. WWW*, 2009.

[10] F. Heintz and Z. Dragisic, "Semantic information integration for stream reasoning," in *Proc. Fusion*, 2012.

[11] F. Heintz and D. de Leng, "Semantic information integration with transformations for stream reasoning," in *Proc. Fusion*, 2013.

[12] D. de Leng, "Extending semantic matching in DyKnow to handle indirectly-available streams," Master's thesis, Utrecht University, 2013.

[13] F. Heintz and P. Doherty, "DyKnow: An approach to middleware for knowledge processing," *J. of Intelligent and Fuzzy Syst.*, vol. 15, no. 1, 2004.

[14] F. Heintz, "DyKnow: A stream-based knowledge processing middleware framework," Ph.D. dissertation, Linköpings universitet, 2009.

[15] ——, "Semantically grounded stream reasoning integrated with ROS," in *Proc. IROS*, 2013.

[16] F. Heintz, J. Kvarnström, and P. Doherty, "Bridging the sense-reasoning gap: DyKnow – stream-based middleware for knowledge processing," *J. of Advanced Engineering Informatics*, vol. 24, no. 1, pp. 14–26, 2010.

[17] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[18] J. Goodwin and D. Russomanno, "Ontology integration within a service-oriented architecture for expert system applications using sensor networks," *Expert Systems*, vol. 26, no. 5, pp. 409–432, 2009.

[19] D. Russomanno, C. Kothari, and O. Thomas, "Building a sensor ontology: A practical approach leveraging ISO and OGC models," in *Proc. the Int. Conf. on AI*, 2005.

[20] A. Bröring, P. Maué, K. Janowicz, D. Nüst, and C. Malewski, "Semantically-enabled sensor plug & play for the sensor web," *Sensors*, vol. 11, no. 8, pp. 7568–7605, 2011.

[21] A. Sheth, C. Henson, and S. Sahoo, "Semantic sensor web," *IEEE Internet Computing*, pp. 78–83, 2008.

[22] M. Botts, G. Percivall, C. Reed, and J. Davidson, "OGC® sensor web enablement: Overview and high level architecture," *GeoSensor networks*, pp. 175–190, 2008.

[23] K. Whitehouse, F. Zhao, and J. Liu, "Semantic streams: a framework for composable semantic interpretation of sensor data," in *Proc. EWSN*, 2006.

[24] M. Waibel, M. Beetz, J. Civera, R. D'Andrea, J. Elfring, D. Galvez-Lopez, K. Haussermann, R. Janssen, J. Montiel, A. Perzylo, B. Schiessle, M. Tenorth, O. Zweigle, and R. van de Molengraft, "RoboEarth," *Robotics Automation Magazine, IEEE*, vol. 18, no. 2, pp. 69–82, 2011.

[25] P. Ziafati, M. Dastani, J.-J. Meyer, and L. van der Torre, "Event-processing in autonomous robot programming," in *Proc. AAMAS*, 2013.

[26] D. Luckham and R. Schulte, "Event processing glossary version 2.0," 2011.

[27] D. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus, "Querying RDF streams with C-SPARQL," *ACM SIGMOD Record*, vol. 39, no. 1, 2010.

[28] D. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus, "An execution environment for C-SPARQL queries," in *Proc. EDBT*, 2010.

[29] E. Prud'hommeaux and A. Seaborne, "SPARQL query language for RDF," 2008.

[30] A. Cheptsov, "Final release of the LarKC platform," 2011.

[31] D. Gyllstrom, E. Wu, H. Chae, Y. Diao, P. Stahlberg, and G. Anderson, "SASE: Complex event processing over streams," in *Proc. CIDR*, 2007.

[32] Y. Diao, N. Immerman, and D. Gyllstrom, "SASE+: An agile language for Kleene closure over event streams."

[33] D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman, "On supporting Kleene closure over event streams." in *Proc. ICDE*, vol. 8, 2008.