Ontology-Based Introspection in Support of Stream Reasoning

Daniel de Leng and **Fredrik Heintz**
Department of Computer and Information Science
Linköping University, 581 83 Linköping, Sweden
{daniel.de.leng, fredrik.heintz}@liu.se

Abstract

Building complex systems such as autonomous robots usually require the integration of a wide variety of components including high-level reasoning functionalities. One important challenge is integrating the information in a system by setting up the data flow between the components. This paper extends our earlier work on semantic matching with support for adaptive on-demand semantic information integration based on ontology-based introspection. We take two important standpoints. First, we consider streams of information, to handle the fact that information often becomes continually and incrementally available. Second, we explicitly represent the semantics of the components and the information that can be provided by them in an ontology. Based on the ontology our custom-made stream configuration planner automatically sets up the stream processing needed to generate the streams of information requested. Furthermore, subscribers are notified when properties of a stream changes, which allows them to adapt accordingly. Since the ontology represents both the system's information about the world and its internal stream processing many other powerful forms of introspection are also made possible. The proposed semantic matching functionality is part of the DyKnow stream reasoning framework and has been integrated in the Robot Operating System (ROS).

1 Introduction

Building complex systems such as autonomous robots usually require the integration of a wide variety of components including high-level reasoning functionalities. This integration is usually done ad-hoc for each particular system. A large part of the integration effort is to make sure that each component has the information it needs in the form it needs it and when it needs it by setting up the data flow between components. Since most of this information becomes incrementally available at run-time it is natural to model the flow of information as a set of *streams*. As the number of sensors and other sources of streams increases there is a growing need for incremental reasoning over streams to draw relevant conclusions and react to new situations with minimal delays. We call such reasoning *stream reasoning*. Reasoning over incrementally available information is needed to support important functionalities such as situation awareness, execution monitoring, and planning.

When handling a large number of streams, it can be difficult to keep track of the semantics of individual streams and how they relate. The same information query further requires different configurations for different systems. Such a manual task leaves ample room for programmer errors, such as misspelling stream names, incorrect stream configurations and misunderstanding the semantics of stream content. Furthermore, if the indefinite continuation of a stream cannot be guaranteed, manual reconfiguration may be necessary at run-time, further increasing the risk for errors.

In this paper we extend earlier work on semantic matching (Heintz and de Leng 2013) where we introduced support for generating indirectly-available streams based on features. The extension focuses on ontology-based introspection for supporting adaptive on-demand semantic information integration. The basis for our approach is an ontology which represents the relevant concepts in the application domain, the stream processing capabilities of the system and the information currently generated by the system in terms of the application-dependent concepts. Relevant concepts are for example objects, sorts and features which the system wants to reason about. Semantic matching uses the ontology to compute a specification of the stream processing needed to generate the requested streams of information. It is for example possible to request the speed of a particular object, which requires generating a stream of GPS-coordinates of that object which are then filtered in order to generate a stream containing the estimated speed of the object. Figure 1 shows an overview of the approach. The semantic matching is done by the Semantics Manager (Sec. 4) and the stream processing is done by the Stream Processing Engine (Sec. 3).

Semantic matching allows for the automatic generation of indirectly-available streams, the handling of cases where there exist multiple applicable streams, support for coping with the loss of a stream, and introspection of the set of available and potential streams. We have for example used semantic matching to support metric temporal logical (MTL) reasoning (Koymans 1990) over streams for collaborative unmanned aircraft missions. Our work also extends the stream processing capabilities of our framework. In particular, this includes ontology-based introspection to support domain-specific reasoning at multiple levels of abstraction.

The proposed semantic matching functionality is integrated with the DyKnow stream reasoning framework (Heintz and Doherty 2004; Heintz 2009; Heintz, Kvarnström, and Doherty 2010; Heintz 2013) which pro-

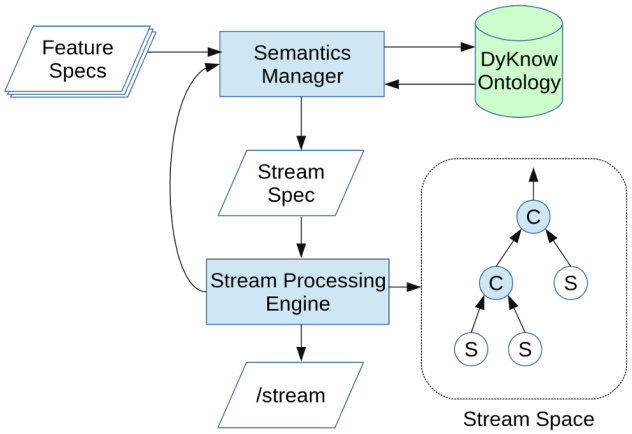

Figure 1: High-level overview of our approach.

vides functionality for processing streams of information and has been integrated in the Robot Operating System (ROS) (Quigley et al. 2009). DyKnow is related to both Data Stream Management Systems and Complex Event Processing (Cugola and Margara 2012). The approach is general and can be used with other stream processing systems.

The remainder of this paper is organized as follows. Section 2 starts off by putting the presented ideas in the context of similar and related efforts. In Section 3, we give an introduction to the underlying stream processing framework. This is a prelude to Section 4, which describes the details of our approach, where we also highlight functionality of interest made possible as the result of semantic matching. The paper concludes in Section 5 by providing a discussion of the introduced concepts and future work.

2 Related Work

Our approach is in line with recent work on semantic modeling of sensors (Goodwin and Russomanno 2009; Russomanno, Kothari, and Thomas 2005) and work on semantic annotation of observations for the Semantic Sensor Web (Bröring et al. 2011; Sheth, Henson, and Sahoo 2008; Botts et al. 2008). An interested approach is a publish/subscribe model for a sensor network based on semantic matching (Bröring et al. 2011). 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. Our approach also bears some similarity to the work by (Whitehouse, Zhao, and Liu 2006) as both use stream-based reasoning and are inspired by semantic web services. One major difference is that we represent the domain using an ontology while they use a logic-based markup language that supports 'is-a' statements.

In the robotic domain, the discussed problem is sometimes called *self-configuration* and is closely related to *task allocation*. The work by Tang and Parker (Tang and Parker 2005) on ASyMTRe is an example of a system geared towards the automatic self-configuration of robot resources in order to execute a certain task. Similar work was performed by Lundh, Karlsson and Saffiotti (Lundh, Karlsson, and Saffiotti 2008) related to the Ecology of Physically Embedded Intelligent Systems (Saffiotti et al. 2008), also called the PEIS-ecology. Lundh et al. developed a formalisation of the configuration problem, where configurations can be regarded as graphs of functionalities (vertices) and channels (edges), where configurations have a cost measure. This is similar to considering actors and streams respectively. A functionality is described by its name, preconditions, post-conditions, inputs, outputs and cost. Given a high-level goal described as a task, a configuration planner is used to configure a collection of robots towards the execution of the task. Some major differences between the work by Lundh et al. and the work on semantic information integration with DyKnow is that the descriptions of transformations are done semantically with the help of an ontology. Further, DyKnow makes use of streams of incrementally available information rather than shared tuples as used by channels. The configuration planner presented by Lundh et al. assumes full knowledge of the participating agents' capabilities and acts as an authority outside of the individuals agents, whereas we assumes full autonomy of agents and make no assumptions on the knowledge of agents' capabilities. Configuration planning further shares some similarities with efforts in the area of knowledge-based planning, where the focus is not on the actions to be performed but on the internal knowledge state.

In a broader context, the presented ideas are in line with a broader trend that moves away from the *how* and towards the *what*. Content-centric networks (CCN) seek to allow users to simply specify what data resource they are interested in, and lets the network handle the localisation and retrieval of that data resource. In the database community, the problem of self-configuration is somewhat similar to the handling of distributed data sources such as ontologies. The local-as-view and global-as-view approaches (Lenzerini 2002) both seek to provide a single interface that performs any necessary query rewriting and optimisation.

The approach presented here extends previous work by (Heintz and Dragisic 2012; Heintz and de Leng 2013; Heintz 2013) where the annotation was done in a separate XML-based language. This is a significant improvement since now both the system's information about the world and its internal stream processing are represented in a single ontology. This allows many powerful forms of introspective reasoning of which semantic matching is one.

3 Stream Processing with DyKnow

Stream processing is the basis for our approach to semantic information integration. It is used for generating streams by for example importing, synchronizing and transforming streams. A stream is a named sequences of incrementally-available time-stamped *samples* each containing a set of named values. Streams are generated by *stream processing engines* based on declarative specifications.

3.1 Representing Information Flows

Streams are regarded as fundamental entities in DyKnow. For any given system, we call the set of active streams the *stream space* $S \subseteq S^*$, where S^* is the set of all possible streams; the *stream universe*. A sample is represented as a tuple $\langle t_a, t_v, \vec{v} \rangle$, where t_a represents the time the sample became available, t_v represents the time for which the sample is valid, and $\vec{v}$ represents a vector of values. A special kind of stream is the *constant stream*, which only contains one sample. The execution of an information flow processing system is described by a series of *stream space transitions* $S^{t_0} \Rightarrow S^{t_1} \Rightarrow \cdots \Rightarrow S^{t_n}$. Here S^t represents a stream space at time t such that every sample in every stream in S has an available time $t_a \leq t$.

Transformations in this context are stream-generating functions that take streams as arguments. They are associated with an identifying label and a specification determining the instantiation procedure. This abstracts away the implementation of transformations from the stream processing functionality. Transformations are related to the combination of implementation and parameters of their corresponding implementations. This means that for a given implementation there might exist multiple transformations, each using different parameters for the implementation.

When a transformation is instantiated, the instance is called a *computational unit*. This instantiation is performed by the stream processing engine. A computational unit is associated with a number of input and output streams. It is able to replace input and output streams at will. A computational unit with zero input streams is called a *source*. An example of a source is a sensor interface that takes raw sensor data and streams this data. Conversely, computational units with zero transmitters are called *sinks*. An example of a sink is a storage or a unit that is used to control the agent hosting the system, such as an unmanned aerial vehicle (UAV).

DyKnow's stream processing engine as shown in Figure 1 is responsible for manipulating the stream space based on declarative specifications, and thereby plays a key role as the foundation for the stream reasoning framework.

3.2 Configurations in DyKnow

A configuration represents the state of the stream processing system in terms of computational units and the streams connecting them. The configuration can be changed through the use of declarative stream specifications. An example of a stream specification is shown in Listing 1, and describes a configuration for producing a stream of locations for detected humans.

Listing 1: Configuration specification format

```
<?xml version="1.0" encoding="UTF-8"?>
<spec:specification
   xmlns:xsi=
      "http://www.w3.org/2001/XMLSchema-
          instance"
   xsi:schemaLocation=
      "http://www.dyknow.eu/ontology#
          Specification
      http://www.dyknow.eu/config.xsd"
   xmlns:spec=
      "http://www.dyknow.eu/ontology#
          Specification">
   <spec:insertions>
      <spec:cu name="result"
         type="project2Dto3D">
         <spec:cu type="fusionRGBIR">
            <spec:cu type="rgbCam" />
            <spec:cu type="irCam" />
         </spec:cu>
         <spec:cu type="GPSto3D">
            <spec:cu type="gps" />
         </spec:cu>
      </spec:cu>
   </spec:insertions>
   <spec:removals>
      <!-- Removals based on names of
          transformations and CUs -->
   </spec:removals>
</spec:specification>
```

The shown specification can be executed by the stream processing engine, which instantiates the declared computational units and connects them according to the specification. In the example shown here, we make use of an XML-based specification tree, where the children of every tree node represent the inputs for that computational unit. The cu tag is used to indicate a computational unit, which may be a source taking no input streams. A computational unit produces at most one stream, and this output stream can thus be used as input stream for other computational units. Indeed, only one computational unit explicitly defines the output stream name as result. When no explicit name is given, DyKnow assigns a unique name for internal bookkeeping. Note that every cu tag has a label associated with it. This label represents the transformation used to instantiate the computational unit, which is then given a unique name by DyKnow as well. As long as a transformation label is associated with an implementation and parameter settings, the stream processing engine is able to use this information to do the instantiation. In this toy example, the specification tree uses a GPS to infer coordinates, and combines this with RGB and infrared video data to provide the coordinates of some entities detected in the video data. Since DyKnow has been implemented in ROS, currently only Nodelet-based implementations are supported.

The result of the stream declaration is that the stream processing engine instantiates the necessary transformations and automatically assigns the necessary subscriptions for the result stream to be executed. Additionally, it uses its own /status stream to inform subscribers when it instantiates a transformation or starts a new stream, along with the specification used. This makes it possible for other components or even computational units to respond to changes to the stream space. This is illustrated in Figure 1, where the /status stream reports to the semantics manager. The stream space shows streams as arrows produced by computational units (C) and sources (S).

The described stream processing capability serves as a foundation for stream reasoning. It makes it possible to generate streams based on specifications over labels, abstracting away some of the low-level details. However, further improvements can be made by considering ontology-based introspection through semantic information integration.

4 Semantic Information Integration

Semantic information integration in the context of this paper is about allowing a component to specify what information it needs relative to an ontology describing the semantics of the information provided by a system. This allows the system to reason about its own information and what is required to generate particular information. It takes away the need to know exactly which streams contain what information, what information is currently being produced by the system, or which combination of transformations generates a specific kind of information. It greatly simplifies the architecture of components connected by streams to one where only the desired information needs to be described at a higher level of abstraction, and wherein the underlying system configures and adapts automatically. This is achieved through the use of ontologies and a technique called *semantic matching*. Both are maintained in our framework by the *semantics manager*. The advantages of this approach includes the automatic generation of indirectly-available streams, the handling of cases where there exist multiple applicable streams, providing support for coping with the loss of a stream, and providing support for the introspection of the space of available and potential streams.

4.1 Ontology for Configuration Modeling

Ontologies are used to describe concepts and relations between concepts. The Web Ontology Language (OWL) (McGuinness, Van Harmelen, and others 2004) was designed to describe such ontologies, and is closely related to Description Logic (DL). In efforts to further the Semantic Web (Berners-Lee, Hendler, and Lassila 2001), many ontologies have been created. However, to the best of our knowledge no ontology exists to describe the concepts related to streams and stream transformations. As such, in developing our own ontology to serve as a data model for our stream reasoning framework.

In order to describe the stream space, we developed the *DyKnow Ontology for Stream Space Modeling*[1]. Figure 2 shows the corresponding concept graph generated by Protégé. We use the prefix : (colon) to refer to concepts in this ontology. The ontology seeks to specify the general concepts related to streams. Some of these terms have been discussed in the previous section, and are formalized in the ontology. For example, by using an ontology we can also indicate that every stream has at least one sample, and that a computational unit is an instance of a transformation and has some input and output streams. This makes it possible to model and reason about sets of streams and changes. For example, we can assign an individual (object) to the :Stream concept to represent an existing stream. Similarly,

[1] http://www.dyknow.eu/ontology

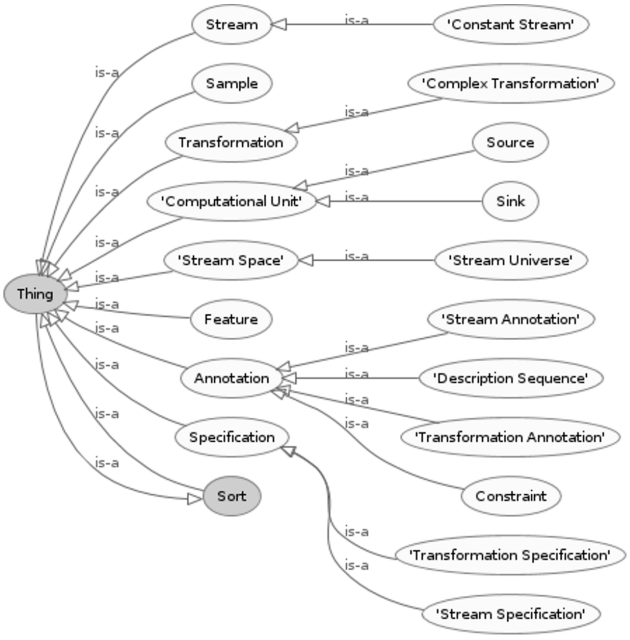

Figure 2: Protégé-generated concept graph of the application independent DyKnow Ontology for Stream Space Modeling

we can model the computational units and their input and output streams. By using a DL reasoner, we can then infer (through the inverse property between :hasInput and :isInput) which computational units a stream acts as input and output for. Concretely, by populating the ontology with information on the stream space, it can serve as a structured semantic model of the stream space that can be queried. As an example, consider again the specification presented in Listing 1. It assumes a number of transformations (indicated by spec:type) which are represented as individuals of dyknow:Transformation. The computational units generated as the result of executing this specification are also registered by the semantics manager and added to the ontology. The fusionRGBIR-type computational unit is connected to its input providers of types rgbCam and irCam. Listing 2 shows how the ontology registers this computational unit. The registration for the other computational units in the specification is done similarly.

Listing 2: Example CU in Turtle syntax

```
:dyknow_cu1 a :ComputationalUnit ;
    :hasInput :dyknow_stream1 ;
    :hasInput :dyknow_stream2 ;
    :hasOutput :dyknow_stream3 ;
    :instantiationOf :fusionRGBIR .
```

The second group of concepts of interest are :Annotation and :Specification. A specification as mentioned earlier describes how something is constructed. As such, the functional :hasSpecification object property can be used to assign one specification to for example a stream or a transformation. The :Annotation concept is used to provide annotations for objects in the ontology. The annotations are

used for describing transformations by their input and output features. DyKnow considers entities in the world to be classified as *sorts*, which represent alike groups of objects. For instance, a sort can be UAV, for which uav2 might be an object. *Features* are used to describe object properties or relations between objects. Therefore :Sort and :Feature are also part of this ontology, which allows us to specify hierarchical structures over sorts and features, e.g. Car $\sqsubseteq$ Vehicle (i.e. Car less general than Vehicle). Every ontological Thing can be regarded as a :Sort, and as such the two are considered to be equivalent concepts. The :Sort and :Feature concepts act like stubs that can be extended for a particular application domain. An example for transformations is shown in Listing 3, where the fusionRGBIR transformation is used.

Listing 3: Example transformation in Turtle syntax

```
:fusionRGBIR a :Transformation ;
   :hasAnnotation [
      :hasOutputAnnotation [
         :describesFeature :ImagePosition ;
         :describesSort :Human
      ] ;
      :hasInputAnnotation [
         :describesFeature :RawRGBCam ;
         :describesSort :self ;
         :nextSegment [
            :hasInputAnnotation [
               :describesFeature
                  :RawIRCam ;
               :describesSort :self
            ]
         ]
      ]
   ] ;
   :hasName ''fusionRGBIR"^^string .
```

In this example, :fusionRGBIR represents the transformation from RGB and IR camera images to 2D image coordinates. As expected, it is an individual of the :Transformation concept. The example also states that it has an annotation and a name string. Note that these :Annotation individuals are not the same as OWL annotations, which are treated as comments that are disregarded by DL reasoners. The annotation has an output annotation and an input annotation, both describing a feature and a number of sorts depending on the arity of the feature. For the input annotation, we specify two inputs where the ordering is kept explicit using :nextSegment in order to avoid ambiguity. The same construct can be used to construct non-unary features by specifying a list of sorts, making use of the :DescriptionSequence concept.

The ontology makes it possible to draw inferences over the internal state of the system. Even though the semantics manager only reports status updates, the ontology specifies properties over predicates that can be used to perform limited reasoning. For instance, using DL reasoners it is possible to determine for a given transformation which computational units are instances. Similarly, given a stream we can query which computational units it serves as inputs and outputs for. Likewise, one could query the ontology for transformations with some provided annotation.

The OWL-S (Martin et al. 2004) and SSN (Compton and others 2012) ontologies are closely related to the application focus of this paper. OWL-S is an upper ontology for services in which services can be described by service profiles. Being an upper ontology, it restricts itself to abstract representations, leaving more concrete extensions to users of the upper ontology. Similarly, the SSN ontology takes a sensor-centric approach. Our ontology differs by representing both the transformations (services) and streams through population of the ontology with individuals, and complements the aforementioned ontologies.

4.2 Maintaining the System Ontology Correspondence

The ontology presented in the previous section can be used as a knowledge base (KB) for stream reasoning frameworks in general. Note that the KB treats streams and transformations over streams as entities that can be assigned properties to. This approach is similar to semantic (or RDF) streams, which can be used to represent parts of an ontology that change over time. Rather than representing the data contained within streams, we chose to represent the streams themselves as entities. From a stream reasoning framework perspective, this allows us to model the active and potential streams and transformations.

Figure 1 showed a high-level outline of our approach. When considering the task of maintaining the knowledge base, our focus is on the semantics manager. The semantics manager is primarily tasked with detecting changes that take place in the system, such as new computational units being instantiated or existing computational units changing their subscriptions to streams. It is able to perform this task by listening to the status streams of computational units, which they use to notify subscribers when their individual configurations change. Of course, this leads to a bootstrapping issue of finding the computational units in the first place. Recall that the stream processing engine is used to execute configuration specifications. Given a configuration specification, it instantiates new computational units and provides information on the names of the streams to subscribe to or produce. The ability to instantiate new computational units is not limited to the stream processing engine, but it serves as the first computational unit in the system. As such, the semantics manager can presume its existence and listen to its status stream to capture the instantiation of any new computational units. By listening to status streams, the semantics manager is able to keep track of the state of the stream processing system and update the ontology to match the state.

In addition to tracking the system configuration and modeling this in the ontology, the semantics manager is able to model additional information. In our conceptualisation, computational units are instances of transformations, which in turn represent the combination of implementations and parameters. For example, a people tracker implementation may need a number of parameter assignments in order to work properly on a specific UAV type. There may be a num-

ber of such combinations consisting of a specific implementation and a number of parameter assignments. Every such combination is represented as a labelled transformation. A transformation can have multiple computational unit instances, which are combinations of transformations with specific input and output streams. Transformations thus do not exist themselves as entities in the system state, but the ontology is able to describe them and relate them to computational unit instances. Similarly, it is possible to annotate entities with additional information. For example, in the stream reasoning context, it is useful to annotate transformations with the features it takes and produces. This is used to perform semantic matching, described in detail below.

By providing an interface to the model of the system state (or configuration), computational units themselves can request changes to be made to the ontology. This can be useful when properties change and have to be updated accordingly, such as may be the case when describing the semantics of a stream using annotations in cases where the stream may change due to environmental changes.

4.3 Semantic Matching Algorithm

Semantic matching in the context of a stream reasoning framework presented here is the task of providing a stream specification given a desired feature. Such a specification may make use of existing streams and computational units, or it may use its knowledge of transformations to reconfigure the system in such a way that it produces a stream with the desired feature. We call such streams *latent streams*. The focus is on desired features because we are interested in reasoning over metric temporal logic formulas, where the feature symbols need to be grounded in streams in order for them to have meaning. The semantic matching procedure is another powerful form of introspection using the ontology. Semantic matching is thus performed by the semantics manager and constitutes its secondary responsibility of providing high-level services related to the ontology.

By providing semantic annotations for transformations, we can specify which features a transformation produces or requires. The semantics manager's services make it possible to provide these semantic annotation during run-time, both by a human operator or a computational unit. Features describe properties of objects or relations between objects. For example, Altitude(UAV) describes the unary Altitude feature over the UAV sort. A transformation produces a single output stream and any number of input streams. We consider the following example transformations, where the name of the transformation is followed by the input and output stream annotations, shown below.

- `gps` : $\emptyset \Rightarrow$ GPS[self]
- `imu` : $\emptyset \Rightarrow$ IMU[self]
- `rgbCam` : $\emptyset \Rightarrow$ RGB[self]
- `irCam` : $\emptyset \Rightarrow$ IR[self]
- `attitude` : IMU[Thing] $\Rightarrow$ Attitude[Thing]
- `GPSto3D` : GPS[Thing] $\Rightarrow$ GeoLocation[Thing]
- `humanDetector` : RGB[RMAX], IR[RMAX] $\Rightarrow$ PixelLocation[Human]
- `humanCoordinates` : PixelLocation[Human], GeoLocation[RMAX], Attitude[RMAX] $\Rightarrow$ GeoLocation[Human]

In this small example, the source transformations are marked as having no input features. RGB and IR are intended to represent colour and infrared camera streams. A Yamaha RMAX is a type of rotary UAV, and self is assumed to be of sort RMAX. We also represent a human detector, which in the 2D version produces pixel location information from the camera data. This can then be combined with the state of an RMAX to produce an estimation of the 3D position of a detected human. Note that the detectors are specific to the RMAX sort because they depend on certain parameters that are specific to the UAV platform used. This allows for the same implementation to be used with different parameters for a different platform, and in such a case it is treated as a different transformation.

If we are interested in a stream of GeoLocation features for the Human sort, we can generate a specification that produces such a stream if we make use of the above transformation annotations. While the example can provide one specification, in some cases we may have multiple possible alternative specifications for generating the desired feature information. This could happen when there already exists a computational unit producing the desired feature information, or even just part of the information needed in order to generate the desired feature information. Additionally, there might simply be multiple ways of generating the same feature information. For example, assume we add a transformation that uses both the GPS and IMU to determine location:

$$\texttt{IMUGPSto3D} : \mathsf{GPS}[\mathsf{Thing}], \mathsf{IMU}[\mathsf{Thing}] \Rightarrow \mathsf{GeoLocation}[\mathsf{Thing}]$$

Now there are two ways of getting GeoLocation information. In order to avoid a lot of duplicate subtrees, we make use of a tree datastructure, in which every node represents a transformation or computational unit instance and edges correspond to features. A node's children are collections of nodes that produce the same feature. The transformation tree is produced for some desired feature, which then yields a set of valid subtrees each of which produces the desired feature. A subtree is valid iff none of its leaf nodes require any input features, i.e. computational unit instances or source transformations. By adding the constraint that features may only occur once along every path in the tree, we prevent cycles.

Once a transformation tree has been generated, it contains all possible ways of generating the desired feature. A stream specification can be generated by traversing the tree and picking a single transformation for every set of applicable transformations. In the process, subtrees can be removed based on some strategy.

Fast solution By doing a depth-first traversal, a stream specification can be found while excluding potentially a large part of the search space. This might be useful when a quick solution is desired, after which a more expensive strategy can be used to find a better solution to switch to.

Minimise cost Similar to the configuration problem work by (Lundh, Karlsson, and Saffiotti 2008), we can assign a

cost to instantiating transformations as computational units. The cost can be a property of a transformation in the ontology. This way the algorithm is encouraged to make use of pre-existing streams where possible. Another cost could be assigned to the number of subscribers for a particular stream, which can be determined from the ontology.

Maximise quality Instead of minimising the cost, we can maximise the quality. This strategy picks transformations with the highest information quality. Since cost and quality are not necessarily inversely proportional, the two strategies are distinct. In the example, quality maximisation might mean that the strategy would pick the `IMUGPSto3D` transformation rather than the `GPSto3D` transformation, combining IMU and GPS data even if this comes at a performance cost.

4.4 Adaptive Subscriptions

Semantic matching makes it possible to find or generate streams of desired information, but it can also be used to deal with instances where streams stop. This could occur due to a source no longer providing information, or a transformation becoming unresponsive, both of which are real problems in integrated systems. This allows for *adaptivity*. Another issue that might occur in for example robotic systems is a change of semantics for a particular stream due to the actions performed by the system. Being able to detect these changes and provide an appropriate response makes a system more rigid to (unexpected) changes. *Adaptive subscriptions* make use of semantic matching to this end, and thereby employ a more refined form of ontology-based introspection.

Subscriptions are usually based on the name of a generated stream. The problem is that these break easily and fail to capture the intended behaviour of a subscription. A client should not have to care about whether a stream is still active, or whether its semantics changes over time. In our stream reasoning context, we are interested in streams representing features. Using semantic matching, we are able to automatically construct a specification to generate a stream for a specific feature. However, if the stream stops for whatever reason, or if the semantics of the stream changes, it is important that the subscription is revised to take these changes into account. The semantics manager is able to do so to some degree. The main challenge is the detection and reporting of these changes. A computational unit is able to determine when the quality of a stream drops, for example when delays increase, and can request the semantics manager to for example change the quality property of some stream-producing computational unit to reflect the poor quality. Similarly, a change in semantics caused by a new computational unit can be reported by that computational unit.

Since ontology updates are handled by the semantics manager, it can respond to those updates accordingly. In some cases this might mean that semantic matching needs to be performed to find an alternative stream for a desired feature. Because the semantics manager is able to perform ontology-based introspection, it can make sure to only change the subscriptions for those computational units that require them. The resulting adaptivity greatly enhances the ability of the system to cope with unexpected changes, and shows further importance of introspective capabilities.

5 Conclusions and Future Work

We have presented an approach to ontology-based introspection supporting stream reasoning. Introspection is used to configure the stream processing system and adapt it to changing circumstances. The presented semantic matching approach based on introspection makes it possible to specify information of interest, which is then provided automatically. This functionality makes it possible to provide high-level descriptions, for example in the evaluation of spatio-temporal logical formulas over streams, without having to worry about individual streams or transformations. The high-level descriptions use an ontology, which provides a data model and a common language. Our DyKnow ontology for stream space modeling has been designed to be implementation-independent, and can therefore be used in other stream-based frameworks. Since the ontology represents both the system's information about the world and its internal stream processing many other powerful forms of introspection are also made possible.

There remain many interesting extensions and improvements. Currently the semantic annotations have focused on features and sorts. However, this is a narrow semantic specification for e.g. sensors. Existing work towards the semantic sensor web such as the SensorML (Botts and Robin 2007) and the SSN ontology (Compton and others 2012) may provide hints for such an extension. It is also interesting to consider the case where transformations are provided with their own ontologies, and how such an ontology could be used in combination with the DyKnow ontology for stream space modeling. Presently humans usually provide semantic annotations for transformations, although it is technically possible for a program to provide annotations. A different direction is a multi-agent approach where local collections of streams can be shared between heterogeneous agents.

The presented work presents a great improvement towards our earlier semantic matching efforts by leveraging ontology-based introspection. This makes it possible to provide semantic subscriptions, which are more reliable than syntactic subscriptions. Furthermore, by providing an ontology to describe our stream processing system's internal state, we have a common vocabulary that could be shared between multiple such systems. We believe that our taken direction complements the work done by the semantic/RDF stream reasoning community, by focusing on stream processing at a higher level while using similar semantic web technology tools, and by providing an ontology of our own.

Acknowledgments

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.

References

Berners-Lee, T.; Hendler, J.; and Lassila, O. 2001. The semantic web. *Scientific American*.

Botts, M., and Robin, A. 2007. OpenGIS sensor model language (SensorML) implementation specification. *OpenGIS Implementation Specification OGC*.

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

Bröring, A.; Maué, P.; Janowicz, K.; Nüst, D.; and Malewski, C. 2011. Semantically-enabled sensor plug & play for the sensor web. *Sensors* 11(8):7568–7605.

Compton, M., et al. 2012. The SSN ontology of the W3C semantic sensor network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web* 17.

Cugola, G., and Margara, A. 2012. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*.

Goodwin, J., and Russomanno, D. 2009. Ontology integration within a service-oriented architecture for expert system applications using sensor networks. *Expert Systems* 26(5).

Heintz, F., and de Leng, D. 2013. Semantic information integration with transformations for stream reasoning. In *Proc. Fusion*.

Heintz, F., and Doherty, P. 2004. DyKnow: An approach to middleware for knowledge processing. *J. of Intelligent and Fuzzy Syst.* 15(1).

Heintz, F., and Dragisic, Z. 2012. Semantic information integration for stream reasoning. In *Proc. Fusion*.

Heintz, F.; Kvarnström, J.; and Doherty, P. 2010. Bridging the sense-reasoning gap: DyKnow – stream-based middleware for knowledge processing. *J. of Advanced Engineering Informatics* 24(1):14–26.

Heintz, F. 2009. *DyKnow: a stream-based knowledge processing middleware framework*. Ph.D. thesis, Linköping U.

Heintz, F. 2013. Semantically grounded stream reasoning integrated with ROS. In *Proc. IROS*.

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

Lenzerini, M. 2002. Data integration: A theoretical perspective. In *Proc ACM SIGMOD-SIGACT-SIGART*.

Lundh, R.; Karlsson, L.; and Saffiotti, A. 2008. Autonomous functional configuration of a network robot system. *Robotics and Autonomous Systems* 56(10):819–830.

Martin, D.; Burstein, M.; Hobbs, J.; Lassila, O.; McDermott, D.; McIlraith, S.; Narayanan, S.; Paolucci, M.; Parsia, B.; Payne, T.; et al. 2004. Owl-s: Semantic markup for web services. *W3C member submission*.

McGuinness, D. L.; Van Harmelen, F.; et al. 2004. OWL web ontology language overview. *W3C recommendation*.

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

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

Saffiotti, A.; Broxvall, M.; Gritti, M.; LeBlanc, K.; Lundh, R.; Rashid, J.; Seo, B.; and Cho, Y.-J. 2008. The PEIS-ecology project: vision and results. In *Proc. IROS*.

Sheth, A.; Henson, C.; and Sahoo, S. 2008. Semantic sensor web. *IEEE Internet Computing* 78–83.

Tang, F., and Parker, L. E. 2005. Asymtre: Automated synthesis of multi-robot task solutions through software reconfiguration. In *Robotics and Automation*, 1501–1508. IEEE.

Whitehouse, K.; Zhao, F.; and Liu, J. 2006. Semantic streams: a framework for composable semantic interpretation of sensor data. In *Proc. EWSN*.