Ontology-Based Introspection in Support of Stream Reasoning

Daniel DE LENG [a] and Fredrik HEINTZ [a]
[a] *Linköping University, 581 83 Linköping, Sweden*

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 systems 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).

Keywords. ontology, introspection, semantic matching, stream reasoning

1. Introduction

Building complex systems such as autonomous robots usually requires 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 situation awareness, execution monitoring, and planning.

In this paper we extend earlier work on semantic matching [8] where we introduced support for generating indirectly-available streams. The extension focuses on ontology-based introspection for supporting adaptive on-demand semantic information integration.

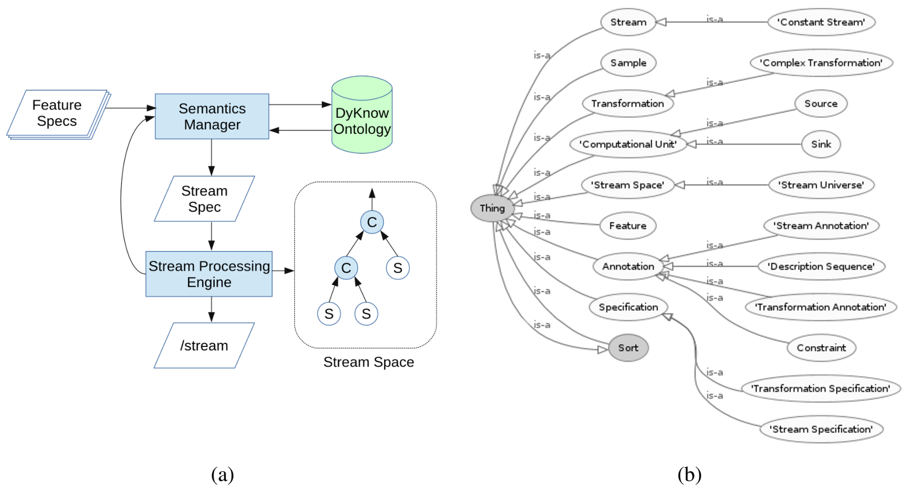


Figure 1. (a) High-level overview of our approach. The stream space shows streams as arrows produced by computational units (C) and sources (S). (b) The Protégé-generated concept graph of the application independent DyKnow Ontology for Stream Space Modeling.

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 generate 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. An overview of the approach is shown in Figure 1a. The semantic matching is done by the Semantics Manager (see Section 4) and the stream processing is done by the Stream Processing Engine (see Section 3).

The proposed semantic matching functionality is integrated with the DyKnow stream reasoning framework [7,9,10] which is integrated in the Robot Operating System (ROS) [15]. We have for example used DyKnow for metric temporal logic (MTL) reasoning [11] over streams. DyKnow is closely related to Data Stream Management Systems (DSMS) and Complex Event Processing (CEP) [5]. The approach is general and could 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. Experimental results on the scalability of our approach are presented in Section 5. The paper concludes with Section 6 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 [6,16] and work on semantic annotation of observations for the Semantic Sensor Web [3,18,2], where ontologies are used to help model streaming information. An interesting approach is a publish/subscribe model for a sensor network based on a different type of semantic matching by Bröring et al [3]. 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 [20] 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.

The work by Tang and Parker [19] 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 [12] related to the Ecology of Physically Embedded Intelligent Systems [17], also called the PEIS-ecology. A major difference 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. 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.

The OWL-S [13] and SSN [4] 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.

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 sequence of incrementally-available time-stamped *samples* each containing a set of (optionally 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. The execution of a stream 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 associated with an implementation and a collection of parameters. 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 outputs are called *sinks*. DyKnow's stream processing engine as shown in Figure 1a is responsible for manipulating the stream space based on declarative specifications.

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 updated through the use of declarative configuration specifications, which are provided using XML. An example of such a specification is shown in Listing 1.

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 spec: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. Here 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 spec:cu tag has a type. This type represents the transformation used to instantiate the computational unit, which is then given a unique name by DyKnow. 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. 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 sets up the necessary subscriptions for the result stream to be produced. 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 1a, where the /status stream reports to the semantics manager.

Listing 1: Configuration specification format

```xml
<?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>
```

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. This ontology describes the semantics of the information provided by a system. Our approach allows the system to reason about its own information and what is required to generate particular information. It takes away the need for a programmer or a computational unit 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. This 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 adapts automatically to provide this information. This is achieved through the use of ontologies and a technique called *semantic matching*. Both are maintained in our framework by the *semantics manager*.

4.1. Ontology for Configuration Modeling

Ontologies are used to describe concepts and relations between concepts. The Web Ontology Language (OWL) [14] was designed to describe such ontologies, and is closely related to Description Logic (DL). In efforts to further the Semantic Web [1], 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, we developed our own ontology to serve as a data model for our stream reasoning framework.

To describe the stream space, we developed the *DyKnow Ontology for Stream Space Modeling*[1]. Figure 1b shows the corresponding concept graph. We use the prefix 'dyknow:' to refer to concepts in this ontology. The goal was 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 dyknow:Stream concept to represent an existing stream. Similarly, 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 dyknow:hasInput and dyknow:isInput) which computational units a stream acts as input and output for. Therefore, 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.

The second group of concepts of interest are dyknow:Annotation and dyknow: Specification. A specification describes how something is constructed. As such, the functional dyknow:hasSpecification object property can be used to assign one specification to for example a stream or a transformation. The dyknow:Annotation concept is used to annotate transformations in the ontology. This is useful when considering the third and final group of concepts.

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 dyknow:Sort and dyknow: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). An individual in the ontology is regarded as an object of the concepts (sorts) it is an individual of. By supporting sorts, features and objects in the ontology, other individuals such as streams, transformations and computational units can be annotated via the use of dyknow:Annotation. Note that dyknow:Annotation individuals are not the same as OWL annotations, which are treated as comments that are disregarded by DL reasoners.

4.2. Maintaining the Correspondence Between System and Ontology

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 to which one can assign properties of interest. This approach is different from but related to for example semantic (or RDF) streams, which can be used to represent a changing part of an ontology or statements on ontology instances: 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 keep a model of the active and potential streams and transformations.

Figure 1a showed a high-level outline of our approach. The semantics manager is primarily tasked with recording changes that take place in the system, such as new computational units being instantiated or existing computational units changing their subscriptions to streams. It performs this task by listening to the status streams of compu-

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

tational units, which they use to notify subscribers when their individual configurations change. Given a configuration specification, the stream processing engine 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.

In addition to capturing 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 number of such combinations consisting of a specific implementation and a number of parameter assignments. Every such combination is represented as a singular 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 in takes and produces. We make use of this to perform semantic matching.

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 type of content for that stream changes.

4.3. Semantic Matching Approach

Semantic matching in the context presented here is the task of providing a stream specification given a desired feature. Such a specification may use 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. The focus is on desired features because we are interested in reasoning over temporal logic formulas in MTL, 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, and is performed by the semantics manager.

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 annotations during run-time, both by a human operator or a computational unit. Consider the following example transformations, where the name of the transformation is followed by the input and output stream annotations:

- $\texttt{gps} : \emptyset \Rightarrow \mathsf{GPS}[\mathsf{self}]$
- $\texttt{imu} : \emptyset \Rightarrow \mathsf{IMU}[\mathsf{self}]$
- $\texttt{rgbCam} : \emptyset \Rightarrow \mathsf{RGB}[\mathsf{self}]$
- $\texttt{irCam} : \emptyset \Rightarrow \mathsf{IR}[\mathsf{self}]$
- $\texttt{attitude} : \mathsf{IMU}[\mathsf{Thing}] \Rightarrow \mathsf{Attitude}[\mathsf{Thing}]$

- `GPSto3D` : $\mathsf{GPS}[\mathsf{Thing}] \Rightarrow \mathsf{GeoLocation}[\mathsf{Thing}]$
- `humanDetector` : $\mathsf{RGB}[\mathsf{RMAX}], \mathsf{IR}[\mathsf{RMAX}] \Rightarrow \mathsf{PixelLocation}[\mathsf{Human}]$
- `humanCoordinates` : $\mathsf{PixelLocation}[\mathsf{Human}], \mathsf{GeoLocation}[\mathsf{RMAX}],$ $\mathsf{Attitude}[\mathsf{RMAX}] \Rightarrow \mathsf{GeoLocation}[\mathsf{Human}]$

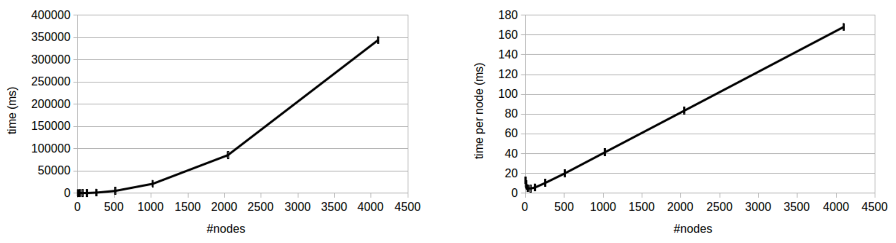
In this example, the source transformations are marked as having no input features (represented by the empty set). RGB and IR represent colour and infrared camera streams. An RMAX is a type of rotary UAV, and self is assumed to be of sort RMAX. We also represent a human detector, which in the 2D version 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.

If we are interested in a stream of $\mathsf{GeoLocation}$ 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: `IMUGPSto3D` : $\mathsf{GPS}[\mathsf{Thing}], \mathsf{IMU}[\mathsf{Thing}] \Rightarrow \mathsf{GeoLocation}[\mathsf{Thing}]$. Now there are two ways of getting $\mathsf{GeoLocation}$ information. To avoid duplicate subtrees, we use 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 doing so, subtrees can be removed based on some strategy. For example, depth-first search can be used to quickly find a solution, or one can optimise for minimum cost or maximum quality using existing search techniques.

5. Experimental Results

To evaluate semantic matching, we consider its scalability by generating collections of transformations that can be organised into a binary tree. The features follow a prefix naming notation: The transformation producing the desired feature F_1 is the root of this tree, and its children are F_{11} and F_{12} (i.e. $F_{11}, F_{12} \Rightarrow F_1$), followed by their respective children F_{111}, F_{112} for F_{11} and F_{121}, F_{122} for F_{12}, etc. Experiments are generated based on a depth value d, yielding full binary trees consisting of $n = 2^d - 1$ nodes. This means that



(a) (b)

Figure 2. (a) Semantic matching processing times. (b) Semantic matching processing times per node.

for every successive value of d, the number of transformations that need to be considered given a desired feature grows exponentially.

Figure 2a shows the average processing times over 20 runs for semantic matching to generate an XML specification given desired feature F_1, relative to the number of transformation nodes. The experiments were run using an Intel Core i7-5500U processor with 8GB of RAM. As the depth increases, there is an exponential increase in total processing time. This is due to the exponential growth of transformations to be considered. Figure 2b shows the processing time per transformation node compared to the total number of nodes. For every node added, it needs to be considered for a potential desired feature match at every other node, resulting in a per-node slowdown that increases linearly as expected.

The number of nodes required for generating a particular desired feature depends heavily on the granularity of the transformations involved: If transformations generate high-level information, few transformations are required to generate a high-level feature. In contrast, if the transformations generate low-level information with many intermediate levels of abstraction, generating a high-level feature may require more nodes. As such, a careful balance must be struck between the granularity of transformations and the computational resources available to the system. This in turn affects the expressivity of semantic annotations and thus the ontology. It is further important to point out that we make use of a binary tree in these experiments, which assumes all transformations take two input streams. There is no restriction on the input streams for transformations; depending on the implementation they may require more or fewer such streams, and this varies by transformation. This shows that the performance of semantic matching further relies on the properties of the transformations it handles.

6. 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 formulas over streams, without having to worry about individual streams or transformations. The high-level descriptions make use of 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 systems information about the world and its internal stream processing many other powerful forms of introspection are also made possible.

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

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

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

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

[4] M. Compton 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.

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

[6] J.C. Goodwin and D.J. Russomanno. Ontology integration within a service-oriented architecture for expert system applications using sensor networks. *Expert Systems*, 26(5):409–432, 2009.

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

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

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

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

[11] R. Koymans. Specifying real-time properties with metric temporal logic. *RTS*, 2(4):255–299, 1990.

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

[13] D. Martin et al. OWL-S: Semantic markup for web services. *W3C member submission*, 2004.

[14] D. McGuinness, F. Van Harmelen, et al. OWL web ontology language overview. *W3C rec.*, 2004.

[15] 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.

[16] D.J. 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.

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

[18] A. Sheth, C. Henson, and S. Sahoo. Semantic sensor web. *IEEE Internet Computing*, 2008.

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

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