Semantic Information Integration for Stream Reasoning

Fredrik Heintz and Zlatan Dragisic
Department of Computer and Information Science, Linköping University, Sweden
fredrik.heintz@liu.se and *zlatan.dragisic@liu.se*

***Abstract*—The main contribution of this paper is a practical semantic information integration approach for stream reasoning based on semantic matching. This is an important functionality for situation awareness applications where temporal reasoning over streams from distributed sources is needed. The integration is achieved by creating a common ontology, specifying the semantic content of streams relative to the ontology and then use semantic matching to find relevant streams. By using semantic mappings between ontologies it is also possible to do semantic matching over multiple ontologies. The complete stream reasoning approach is integrated in the Robot Operating System (ROS) and used in collaborative unmanned aircraft systems missions.**[1]

I. Introduction

The information available to modern autonomous systems such as robots is often in the form of streams. As the number of sensors and other stream sources increases there is a growing need for incremental reasoning over sets of streams in order to draw relevant conclusions and react to new situations as quickly as possible. Reasoning over incrementally available information in autonomous systems is needed to support a number of important functionalities such as situation awareness, execution monitoring and planning [1].

One major issue is integrating stream reasoning in robotic systems. To do symbolic reasoning it is necessary to map symbols to streams in a robotic system, which provides them with the intended meaning for the particular robot. This is currently done syntactically by mapping symbols to streams based on their names. This makes a system fragile as any changes in existing streams or additions of new streams require that the mappings be checked and potentially changed. This also makes it hard to reason over streams of information from multiple heterogeneous sources, since the name and content of streams must be known in advance.

To address these problems we have developed a semantic matching approach using semantic web technologies. An ontology is used to define a common vocabulary over which symbolic reasoning can be done. Streams are annotated with ontological concepts to make semantic matching between symbols and streams possible. One major advantage is that the semantic of a stream can now be described by the creator of the stream and then found by anyone based on this semantic annotation. Previously the user had to know the content and meaning of the streams to select the appropriate ones. This also opens up the possibility to find and fuse multiple streams with relevant information.

This is in line with recent work on semantic modeling of sensors [2], [3] and on semantic annotation of observations for the Semantic Sensor Web [4]–[6]. An interested approach is a publish/subscribe model for a sensor network based on semantic matching [4]. 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 work presented in this paper extends the stream reasoning functionality of the stream-based knowledge processing middleware framework DyKnow [7], [8]. DyKnow is integrated in the Robot Operating System (ROS) [9] which makes it available for a wide variety of robotic systems.

II. Stream Reasoning

One technique for incremental reasoning over streams is progression of metric temporal logic formulas. This provides real-time incremental evaluation of logical formulas as new information becomes available. First order logic is a powerful technique for expressing complex relationships between objects. Metric temporal logics extends first order logics with temporal operators that allows metric temporal relationships to be expressed. For example, our temporal logic, which is a fragment of the Temporal Action Logic (TAL) [10], supports expressions which state that a formula F should hold within 30 seconds and that a formula F' should hold in every state between 10 and 20 seconds from now. This fragment is similar to the well known Metric Temporal Logic [11]. Informally, $\Diamond_{[\tau_1,\tau_2]}\,\phi$ ("eventually") holds at τ iff ϕ holds at some $\tau' \in [\tau+\tau_1, \tau+\tau_2]$, while $\Box_{[\tau_1,\tau_2]}\,\phi$ ("always") holds at τ iff ϕ holds at all $\tau' \in [\tau+\tau_1, \tau+\tau_2]$. Finally, $\phi\,\mathsf{U}_{[\tau_1,\tau_2]}\,\psi$ ("until") holds at τ iff ψ holds at some $\tau' \in [\tau+\tau_1, \tau+\tau_2]$ such that ϕ holds in all states in (τ, τ').

We have for example used this expressive metric temporal logic to monitor the execution of complex plans [12] and to express conditions for when to hypothesize the existence and classification of observed objects in an anchoring framework [13]. In execution monitoring, for example, suppose that

[1]This work is partially supported by grants from the Swedish Foundation for Strategic Research (SSF) project CUAS, the Swedish Research Council (VR) Linnaeus Center CADICS, and the Center for Industrial Information Technology CENIIT.

a UAV supports a maximum continuous power usage of M, but can exceed this by a factor of f for up to τ units of time, if this is followed by normal power usage for a period of length at least τ'. The following formula can be used to detect violations of this specification: $\Box \forall uav.(\mathsf{power}(uav) > M \rightarrow \mathsf{power}(uav) < f \cdot M \, \mathsf{U}_{[0,\tau]} \, \Box_{[0,\tau']} \, \mathsf{power}(uav) \leq M)$.

The semantics of these formulas are defined over infinite state sequences. To make metric temporal logic suitable for stream reasoning, formulas are incrementally evaluated using progression over a stream of timed states. The result of progressing a formula through the first state in a stream is a new formula that holds in the remainder of the state stream if and only if the original formula holds in the complete state stream. If progression returns true (false), the entire formula must be true (false), regardless of future states. Even though the size of a progressed formula may grow exponentially in the worst case, it is always possible to use bounded intervals to limit the growth. It is also possible to introduce simplifications which limits the growth for many common formulas [14].

DyKnow views the world as consisting of *objects* and *features*, where features may for example represent properties of objects and relations between objects. A *sort* is a collection of objects, which may for example represent that they are all of the same type.

Due to inherent limitations in sensing and processing, an agent cannot always expect access to the actual value of a feature over time, instead it will have to use approximations. Such approximations are represented as streams of *samples* called *fluent streams*. Each sample represents an observation or estimation of the value of a feature at a specific point in time called the *valid time*. A sample is also tagged with its *available time*, the time when it is ready to be processed by the receiving process after having been transmitted through a potentially distributed system. The available time allows to formally model delays in the availability of a value and permits an application to use this information introspectively to determine whether to reconfigure the current processing network to achieve better performance. DyKnow also provides support for generating streams of states by synchronizing distributed individual streams. Using the stream specifications it is possible to determine when the best possible state at each time-point can be extracted [14].

III. Semantic Information Integration

A temporal logic formula consists of symbols representing variables, sorts, objects, features, and predicates besides the symbols which are part of the logic. Consider $\forall x \in \mathsf{UAV}.x \neq \mathsf{uav1} \rightarrow \Box \, \mathsf{XYDist}[x, \mathsf{uav1}] > 10$, which has the intended meaning that all UAVs, except uav1, should always be more than 10 meters away from uav1. This formula contains the variable x, the sort UAV, the object uav1, the feature XYDist, the predicates $\neq$ and $>$, and the constant value 10, besides the logical symbols. To evaluate such a formula an interpretation of its symbols must be given. Normally, their meanings are predefined. However, in the case of stream reasoning the meaning of features can not be predefined since information about them becomes incrementally available. Instead their meaning has to be determined at run-time. To evaluate the truth value of a formula it is therefore necessary to map feature symbols to streams, synchronize these streams and extract a state sequence where each state assigns a value to each feature [14].

In a system consisting of streams a natural approach is to syntactically map each feature to a single stream. This works well when there is a stream for each feature and the person writing the formula is aware of the meaning of each stream in the system. However, as systems become more complex and if the set of streams or their meaning changes over time it is much harder for a designer to explicitly state and maintain this mapping. Therefore automatic support for mapping features in a formula to streams in a system is needed. The purpose of this matching is for each feature to find one or more streams whose content matches the intended meaning of the feature. This is a form of semantic matching between features and contents of streams. The process of matching features to streams in a system requires that the meaning of the content of the streams is represented and that this representation can be used for matching the intended meaning of features with the actual content of streams.

The same approach can be used for symbols referring to objects and sorts. It is important to note that the semantics of the logic requires the set of objects to be fixed. This means that the meaning of an object or a sort must be determined for a formula before it is evaluated and then may not change. It is of course possible to have different instances of the same formula with different interpretations of the sorts and objects.

Our goal is to automate the process of matching the intended meaning of features, objects and sorts to content of streams in a system. Therefore the representation of the semantics of streams needs to be machine readable. This allows the system to reason about which stream content corresponds to which symbol in a logical formula. The knowledge about the meaning of the content of streams needs to be specified by a user, even though it could be possible to automatically determine this in the future. By assigning meaning to stream content the streams do not have to use predetermined names, hard-coded in the system. This also makes the system domain independent meaning that it could be used to solve different problems in a variety of domains without reprogramming.

Our solution involves creating an ontology acting as a common vocabulary of features, objects and sorts, a language for representing the content of streams relative to an ontology, and a semantic matching algorithm for finding all streams which contain information relevant for a feature, object or sort in the ontology.

A. Ontologies

An important step in semantic information integration is to establish a vocabulary of the information. One way to do this is to model the entities and the relationships between the entities in the domain. To make this usable in the semantic matching process the domain model should be interpretable

by machines. Ontologies provide suitable support for creating machine readable domain models [15]. Ontologies also provide reasoning support and support for semantic mapping which is necessary for the integration of streams on multiple platforms, see Section III-B.

Reasoning about an ontology is used to make knowledge that is implicit in the ontology explicit. For example if A is a subclass of B and B is a subclass of C, a reasoning process could infer that A is also a subclass of C. This type of reasoning is based on class inference. Reasoners can also be used to determine the coherence of an ontology or more precisely determine if some concept (class) is unsatisfiable. A class is unsatisfiable if the interpretation of that class is the empty set in all models of the ontology.

To represent ontologies the Web Ontology Language (OWL), a W3C ontology language recommendation, is used [16]. OWL uses a RDF/XML syntax but other syntaxes also exist which provide higher readability for humans. There are three different versions of OWL [16]: OWL Full, OWL DL, and OWL Lite. These languages differ in the expressiveness and correspondingly the decidability and computational complexity. We have chosen to use OWL DL which supports the decidable fragment of OWL Full. It is based on description logics and guarantees completeness and decidability [17].

In OWL it is possible to define classes, properties and instances. OWL includes additional constructs which give knowledge engineers more expressive power. These constructs allow them to describe classes as the union, intersection or complement of other classes. To specify restrictions on the values of properties, OWL supports universal and existential quantifiers together with three different cardinality restrictions: minimal, maximal and exact.

To represent features, objects and sorts we propose to use an ontology with two different class hierarchies, one for objects and one for features. The feature hierarchy is actually a reification of OWL relations. The reason for this is that OWL only supports binary relations while we need to handle arbitrary relations. Figure 1 shows an example ontology.

As the object hierarchy shows, our example domain deals with two types of objects, static and moving objects. Static objects in this case represent points of interest while moving objects are different types of vehicles. The actual objects or instances of the classes are not shown in the figure but must also be included in the ontology. The example ontology includes 5 objects: uav1 and uav2 of type UAV and car1, car2 and car3 of type Car.

Features describe relations in the domain and are represented as relations in the ontology. These relations in our ontology are described as intersection classes. The intersection includes the class which defines the arity of the feature ($UnaryRelation$, $BinaryRelation$, or $TernaryRelation$) and an enumeration of possible classes for each argument. The enumeration is done using object properties $arg1$, $arg2$ and $arg3$ which specify the order of the arguments. For example, the feature XYDist representing the distance between two objects in the XY-plane is defined as follows: XYDist $\sqsubseteq$

Fig. 1: An example ontology defining objects and features.

$BinaryRelation \sqcap \forall arg1.\text{Object} \sqcap \forall arg2.\text{Object}$ meaning that XYDist is a binary relation where both arguments must be of type Object.

Reasoning is required to infer implicit relations in the ontology. As a simple example, consider the feature XYDist which takes arguments of type Object. Therefore objects uav1 and uav2 of type UAV could not be used as arguments even though all objects defined in the ontology are of type Object. Using the reasoner these relations would be included in the ontology and uav1 and uav2 could be used as arguments to XYDist.

B. Mappings Between Ontologies

To evaluate formulas over streams from many different platforms, each having their own ontology, it is necessary to align multiple ontologies. It could for example be the case that no single platform has all the relevant streams. To align two ontologies, they have to have some overlap and partially describe the same aspects of the environment. An issue is that even if two ontologies model the same part of the environment they might use different concept names. An example of this is shown in Figure 1 and Figure 2. The first ontology deals with both aerial and ground vehicles while the second ontology only contains ground vehicles. However, since they use different concepts it is not possible to directly fuse the information from the two ontologies, even if they contain information about the same vehicles in the environment.

One approach to fuse knowledge from multiple ontologies is to specify the relations between concepts in the different ontologies, so called *semantic mappings*, and use a reasoning mechanism which can reason over multiple ontologies [18]. This is a multiple ontology approach [19] which has the ad-

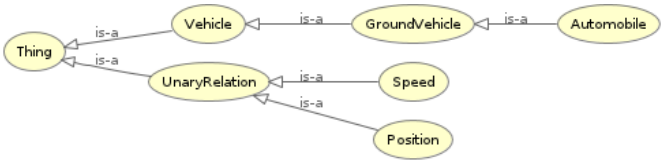

Fig. 2: A second example ontology.

vantage that information sources are independent and changes in a local ontology does not influence other local ontologies.

Much work has been done related to semantic mappings between ontologies [20]–[23]. The work done by Serafini and Tamilin [18] differs from these as it also presents a reasoning mechanism for reasoning over multiple ontologies connected by semantic mappings. Their semantic mappings are based on C-OWL [21]. In order to support reasoning over multiple distributed ontologies they use the idea of Distributed Description Logics (DDL) [24] which provides support for formalizing collections of ontologies connected by semantic mappings. The DDL reasoning is based on a tableau reasoning technique for local description logics which was extended to support multiple ontologies.

We use C-OWL [21] for representing semantic mappings as it provides support for explicit semantic mappings between classes and individuals in ontologies together with an XML representation which can easily be queried. In this representation a mapping between two ontologies is represented as a set of *bridge rules*. Each bridge rule defines a semantic correspondence, such as equivalence, more general or more specific, between a source and a target ontology entity.

For example, consider the two ontologies in Figure 3. Even though they refer to the same objects in the environment it is impossible to infer this directly. However, by specifying bridge rules between concepts and individuals in the ontologies it is possible to infer this by reasoning over the combined ontology.

C. Semantic Annotation of Streams

After establishing a common vocabulary the next step is to represent streams and their content in terms of this vocabulary. Each representation of a stream should include all the information necessary to subscribe to the stream. Since our approach is integrated with ROS this means that streams correspond to *topics*. Each topic has a name and a message type. To access the content of a stream it is also necessary to know which message field contains which information.

To represent the content of streams we propose the Semantic Specification Language for Topics (SSL_T). Each topic defined in SSL_T includes all the necessary information needed for subscribing to the topic. The formal grammar for SSL_T is presented in Listing 1.

Listing 1: Formal grammar for SSL_T.

```
spec          : expression+ ;
expression    : 'topic' topic_name 'contains'
                feature_list ;
topic_name    : NAME ':' NAME ;
```

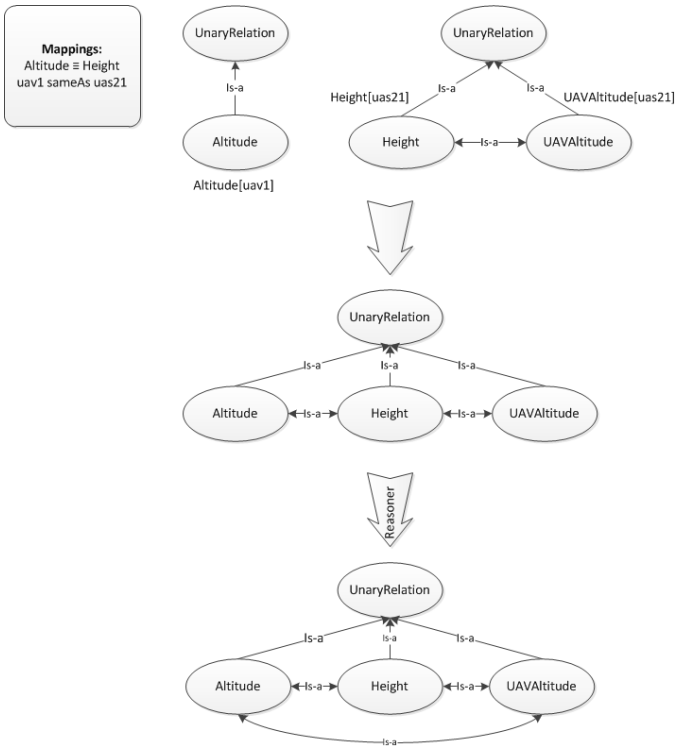

Fig. 3: Examples of mappings between ontologies and the supported inference.

```
feature_list  : feature (',' feature)* ;
feature       : feature_name '=' MSGFIELD for_part? ;
feature_name  : NAME '(' feature_args ')' ;
feature_args  : feature_arg (',' feature_arg)* ;
feature_arg   : entity_name alias?;
for_part      : 'for' entity (',' entity)* ;
entity        : sort | object ;
entity_name   : NAME;
alias         : 'as' NAME;
object        : entity_full ;
sort          : sort_type entity_full ;
entity_full   : NAME '=' MSGFIELD ;
sort_type     : 'some' | 'every' ;
NAME          : ('a'..'z'|'A'..'Z'|'0'..'9')+ ;
MSGFIELD      : NAME '.' NAME;
```

Listing 2: Example SSL_T sort specifications.

```
topic topic1:UAVMsg contains sort some UAV = msg.id
topic topic2:UAVMsg contains sort every UAV = msg.id
```

Listing 2 gives the SSL_T specifications for two topics enumerating objects in a sort. `topic1` contains only a subset of all objects of sort UAV while `topic2` contains all objects of the same sort. As the listing shows, the topic specifications include names of the topics (`topic1` and `topic2`), message types (`UAVMsg`) and the id fields (`msg.id`). The id field is important since it contains the identifier for the particular object instance.

Topics containing features are specified in a similar manner. However, in this case a list of arguments for each feature is also needed. Arguments can either be objects or sorts in which case the topic contains values for the feature for all possible combinations of the specified argument. Listing 3

shows five different topic specification examples. The first topic, `topic1`, contains feature Altitude for the specific object uav1. The actual altitude value is stored in the field `msg.alt`. Similarly, topic `topic2` defines the feature Speed for uav1. However, it is important to note the difference between these two topics. In the case of `topic1` the object argument, uav1, is explicit while in topic `topic2` the object argument is implicit and needs to be computed from a field in the message. This field is specified after the for keyword.

`topic3` defines the same feature but for multiple UAVs. In this case the feature is only defined for a subset of UAVs. `topic4`, `topic5` and `topic6` specify topics which contain feature XYDist which has arity 2. They differ only in the types of arguments they have. It is important to note that if we are defining a topic for a feature which has a sort as an argument we need to define if the topic contains some or all instances of this sort. This is done using the some and every keywords. If these keywords are not specified then the specified topic contains data for a single object with a name and id field specified after the for keyword.

Listing 3: Example SSL_T feature specifications.

```
topic topic1:UAVMsg contains Altitude(uav1)=msg.alt
topic topic2:UAVMsg contains Altitude(uav1)=msg.alt
    for uav1=msg.id
topic topic3:UAVMsg contains Altitude(UAV)=msg.alt
    for some UAV=msg.id
topic topic4:UAVMsg contains XYDist(uav1,uav3)=msg.
    dist for uav1=msg.id1, uav3=msg.id2
topic topic5:UAVMsg contains XYDist(uav2,uav1)=msg.
    dist for uav2=msg.id1, uav1=msg.id2
topic topic6:UAVMsg contains XYDist(UAV,uav2)=msg.
    dist for every UAV=msg.id1, uav2=msg.id2
```

D. Semantic Matching Single Ontology

To achieve automatic semantic matching of the intended meaning of symbols to content of streams it should be possible to automatically determine which streams are relevant for which symbol based on their content. To find all the streams relevant for a logical formula it is necessary to first find and extract all features from the formula and then match these features to relevant streams. Figure 4 gives an overview of the complete matching process.

The first step is to extract features from a formula. As an example, consider the formula $\forall x \in \mathsf{UAV}.x \neq \mathsf{uav1} \rightarrow \Box(\mathsf{Altitude}[\mathsf{uav1}] > 20 \vee \mathsf{XYDist}[x, \mathsf{uav1}] > 10)$. XYDist and Altitude are features, uav1 is a constant, and x is a quantified variable which may take any value from the sort UAV. From this formula, the features XYDist[UAV, uav1] and Altitude[uav1] would be extracted. After completing the feature extraction the next step is to find the relevant streams for each of these features.

This problem can be stated as: Given an ontology, a stream specification, and a parameterized feature find a set of streams which allows the estimation of the value of the feature over time. If a feature is parameterized, then the arguments may either be objects corresponding to constants in a formula or sorts corresponding to quantified variables in a formula.

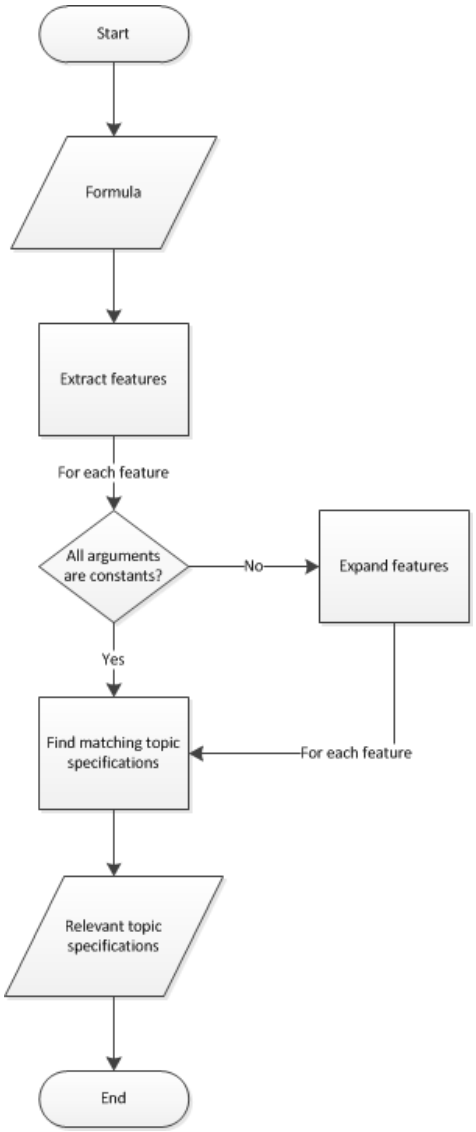

Fig. 4: The semantic matching process.

Altitude[uav1] is an example of a feature with a constant argument, XYDist[UAV,UAV] a feature with two sort arguments, and XYDist[uav1,UAV] a feature with one sort and one constant argument.

As described in the previous section, the meaning of streams is represented in the stream specification. However, the proposed language is not suitable for querying. Therefore, we use an equivalent XML representation. XML is a suitable candidate because it is machine readable and provides good support for querying.

Each topic is defined in the element Topic. This element contains two attributes, `msgtype` and `name` which correspond to the message type and topic name. Features are defined using the `feature` tag which contains the name of the feature and the topic field which has the value for this feature. The list of feature arguments are defined as children to the `feature` element. Each argument includes

the type (object or sort), name and id field. The sorts also include a boolean `all_objects` attribute which defines if the argument covers all objects of certain type or only a subset.

Finding the relevant topics differs depending on which types of arguments a feature has. First we consider the case where a feature only has constant arguments. In this case, topics that match both the feature name and the specific feature argument should be found. This process includes multiple steps. In the first step, only topics that match the feature name are extracted from the topic specifications. For example, querying the specification given in Listing 3 for relevant topics for feature XYDist[uav1, uav2] would find `topic4`, `topic5`, and `topic6`.

In the next step of the algorithm, this list of specifications is used to extract only those topics whose feature arguments match. To find topics with matching arguments the ontology is queried to find the sorts of each argument. In each subsequent step of the algorithm an argument object is chosen and used to extract topic specifications which have this object or a sort of this object as an argument. This step is repeated for every argument of the feature using the list from the previous iteration of the algorithm. In the previous example, the next step of the algorithm would be to extract topic specifications which either have uav1 or sort UAV as the first argument. Therefore, the output from this step of the algorithm would be topics `topic4` and `topic6`.

In the final step, topic specifications based on their second argument are extracted. which in our example should either be the object uav2 or the sort UAV. In this case only `topic6` is relevant as `topic4` specifies feature XYDist which has an object uav3 as the second argument. Therefore the result of matching feature XYDist[uav1, uav2] to the topic specifications in Listing 3 is `topic5`.

However, `topic5` contains feature XYDist for multiple objects since the first argument is a sort. Therefore, in order to get the value of feature XYDist[uav1, uav2] it is necessary to filter out messages which have uav1 and uav2 as the first and second argument respectively. In our approach, this is done using the id fields defined in the specification. Currently the value of the id field is acquired directly from an object name and represents the numeric part of the name. Therefore, in the case of `topic5` only messages which have values `msg.id1` = 1 and `msg.id2` = 2 would contain values for feature XYDist[uav1, uav2].

Next we consider the case where a feature has one or more sort arguments. In this case, the first step of the matching algorithm is to find all possible values for each argument. The set of objects in a sort is represented in the ontology. Therefore to acquire all possible assignments of a variable it is necessary to query the ontology for all instances (objects) of a certain sort. After acquiring these objects, the next step is to form new features, one for each of these objects. This requires taking the Cartesian product of the set of sorts to find all possible combinations. For example, if the sort UAV has two instances, uav1 and uav2, according to the ontology, then the result after expansion for XYDist[uav1, UAV] would be XYDist[uav1, uav1] and XYDist[uav1, uav2]. The expanded features are matched to topic specifications using the algorithm for matching features with constant arguments presented earlier.

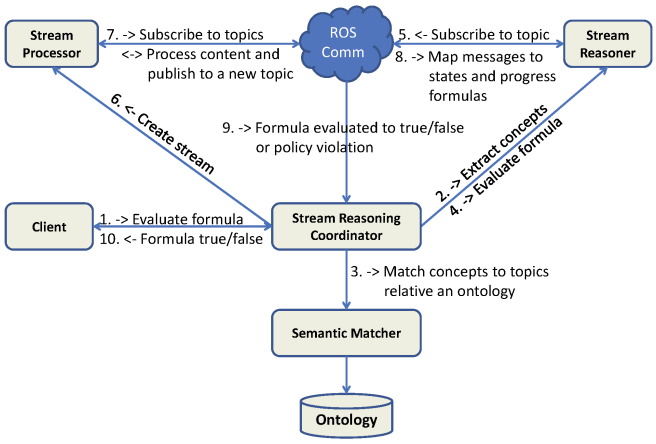

Fig. 5: The stream reasoning architecture.

When the matching is done, the relevant streams must be collected and synchronized in order to evaluate the formula. This is done by generating a state stream specification from the relevant topic specifications which is sent to a stream processing engine. The stream processor generates a stream of states based on the specification which is used by the stream reasoner to evaluate the formula. Figure 5 gives an overview of the complete stream reasoning architecture and the steps involved in temporal logical stream reasoning.

E. Semantic Matching Multiple Ontologies

To support matching topics over multiple ontologies, the process of matching topics to formulas described earlier needs to be extended. Each feature from the formula (after expansion) is taken and first checked against the local topic specifications. In the next step of the algorithm the feature is checked against the distributed topic specifications. This is done by first constructing an ontology which combines the local and the remote ontology, as in Figure 3. The new ontology also includes relevant mappings which can be used by a reasoner (Pellet) to add implicit relations. This new model is used to query for features equivalent to the selected feature. The same thing is done for the arguments of the selected feature. If an equivalent feature exists and all arguments of the selected feature can be mapped to objects in the remote ontology, the feature is added to the list of features that needs to be checked for correctness in the remote ontology. This step is crucial as the ontology and the mappings are defined by a user and are therefore susceptible to accidental errors, e.g., a feature Speed in the remote ontology is defined to accept arguments of type Car, but car1 from the local ontology maps to uav1 which is of type UAV which means that this feature is not valid on this remote platform and therefore it cannot provide the relevant information. If the features are correct in the remote ontology, try to find relevant topics in

the distributed topic specifications following the same method described earlier for the local topic specifications. Relevant topics are added to the list of topics that need to be merged for the selected feature. This process is repeated for each remote ontology.

Let us consider the example in Figure 3. A mapping shows that the feature Altitude in the local ontology is equivalent to the feature Height in the remote ontology. The feature Height is equivalent to the feature UAVAltitude where both concepts are in the remote ontology. While matching Altitude using the previously defined matching process UAVAltitude would not be considered as there does not exist an explicit relationship between these two concepts. In order to find both explicitly and implicitly defined equivalent features we introduce a reasoner in our implementation, Pellet. The reasoner adds new entailments to the ontology, which means that while searching for the equivalent features and objects the features that can be acquired through transitivity would also be included. Using the reasoner the feature UAVAltitude is also found in the example.

IV. Performance Evaluation

To evaluate the performance of our solution a number experiments were made. We varied the size of the ontology, the size of the stream specification, and the type of features being matched. In each case the input was a temporal logical formula, the task was to match all the features in the formula to streams, and we measured the execution time for the different steps in the process. The different steps are:

1) Preprocessing – loading ontologies and topic specifications
2) Extracting features – extracting features from a formula
3) Checking features – checking extracted features against an ontology
4) Matching features – matching features to relevant topic specifications
5) ROS integration – generating appropriate ROS messages

The experiments were run on a Dell Studio 1558, with a 1.6 GHz sixcore Intel i7-720QM processor and 4 gigabytes of memory running Ubuntu 11.04 with ROS 1.4.9 (Diamondback), Jena Semantic Web Framework 2.6.3[2] and Pellet 2.2.2[3] OWL2 reasoner.

A. Varying the size of the ontology

In this experiment the number of concepts in the ontology varies from 25 to 200 while the number of individuals is constant at 200. The topic specification contains 20 specifications and each feature has exactly one relevant specification. The following formula with 9 features is used $X[b1] \wedge X[b2] \wedge X[b3] \wedge Y[b1, b1] \wedge Y[b1, b2] \wedge Y[b1, b3] \wedge Z[b1, b2, b1] \wedge Z[b1, b2, b2] \wedge Z[b1, b2, b3]$.

Figure 6 shows a linear increase in execution time when increasing the number of concepts in the ontology. The sudden increase in the feature checking phase when going from 75 to 100 concepts is probably due to the internals of the Jena Semantic Web Framework. As expected, increasing the number of concepts has the highest impact on the feature checking phase while the matching phase is constant.

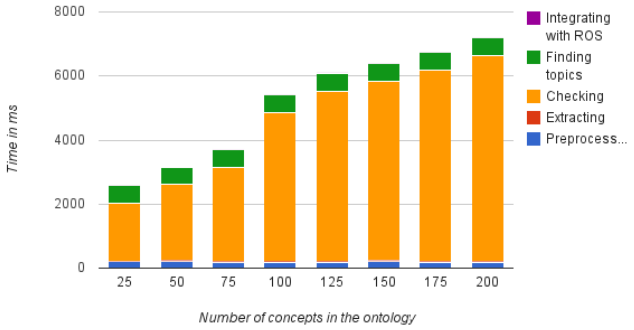

Fig. 6: Performance when varying the number of concepts.

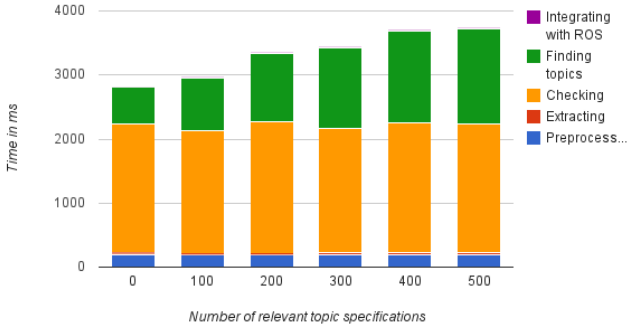

Fig. 7: Performance when varying the number of topic specifications.

An experiment were the number of individuals in the ontology is varied gives similar results.

B. Varying the size of the stream specification

In this experiment the number of extra stream specifications besides those that are necessary for the formula varies from 0 to 500 while the size of the ontology is constant (50 concepts and 50 individuals). The same formula with 9 features as before is used.

Figure 7 shows that the checking phase is constant while the matching phase increase linearly with the number of topic specifications.

C. Varying the type of features

In this experiment the number of quantified feature arguments for a ternary feature varies from 1 to 3 (all). The size of the ontology (50 concepts and 50 individuals) and the topic specification (20 specifications) is constant. The following formulas and their expanded versions are used:

[2]http://jena.sourceforge.net/
[3]http://clarkparsia.com/pellet/

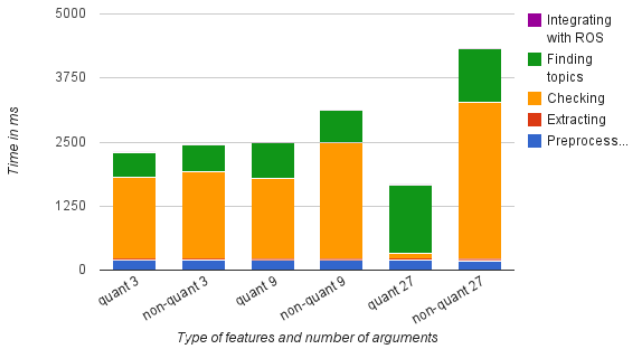

Fig. 8: Comparing quantified and non-quantified versions of a formula.

$$forall\ x\ in\ B\ (Z[x, d1, e1])$$
$$forall\ x\ in\ B, y\ in\ D\ (Z[x, y, e1])$$
$$forall\ x\ in\ B, y\ in\ D, z\ in\ E(Z[x, y, z])$$

Figure 8 show that formulas with quantified arguments require less time for semantic matching. The main difference is in the feature checking phase. This is expected as checking features with object arguments is done in two steps, the first where an ontology is queried for arguments' sorts and the second where these sorts are checked against the ontology. If one or more arguments of a feature is a sort then the first step is skipped. The results indicate that queries to acquire arguments' sorts are more expensive than checking these sorts against a relation in the ontology. The results also show that the matching phase requires more time when quantified variables are used. This was expected as in the case of features with sort arguments additional steps are required for feature expansion.

V. Conclusions and Future Work

We have presented a practical semantic matching approach to finding streams of information based on their meaning. This makes temporal reasoning over streams of information from multiple heterogeneous robots possible which is necessary in many situation awareness applications. The approach uses semantic web technologies to define ontologies (OWL), to define mappings between ontologies (C-OWL) and to reason over multiple related ontologies (DDL). To represent the semantic content of streams in terms of features, objects, and sorts the Semantic Specification Language for Topics SSL_T is introduced. Given a feature the proposed approach finds all streams which contain information relevant for the feature. This makes it possible to automatically find all the streams that are relevant for evaluating a temporal logical formula even if they are distributed among multiple platforms. These streams can then be collected, fused, and synchronized into a single stream of states over which the truth value of the formula is incrementally evaluated. The conclusion is that the proposed approach makes it possible to integrate stream reasoning in real robotic systems based on semantic matching. This functionality is central to creating and maintaining situational awareness in for example collaborative unmanned aircraft systems [25]. We also showed that the approach is practical as it grows linearly with the size of the ontology and the size of the stream specification.

The two most interesting directions for future work are the dynamic adaptation to changes in streams and extending the semantic matching to consider properties such as quality and delays of streams.

References

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

[2] 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.

[3] 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, pp. 17–18.

[4] 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.

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

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

[7] F. Heintz and P. Doherty, "DyKnow federations: Distributing and merging information among UAVs," in *Proc. Fusion*, 2008.

[8] 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.

[9] 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.

[10] P. Doherty and J. Kvarnström, "Temporal action logics," in *Handbook of Knowledge Representation*. Elsevier, 2008.

[11] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-Time Systems*, vol. 2, no. 4, pp. 255–299, 1990.

[12] P. Doherty, J. Kvarnström, and F. Heintz, "A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems," *J. of Auton. Agents and Multi-Agent Systems*, vol. 19, 2009.

[13] F. Heintz, J. Kvarnström, and P. Doherty, "A stream-based hierarchical anchoring framework," in *Proc. IROS*, 2009.

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

[15] I. Horrocks, "Ontologies and the Semantic Web," *Communications of the ACM*, vol. 51, no. 12, p. 58, Dec. 2008.

[16] M. K. Smith, C. Welty, and D. L. McGuinness, "OWL Web Ontology Language Guide," 2004.

[17] I. Horrocks, "OWL: A description logic based ontology language," *Logic Programming*, pp. 1–4, 2005.

[18] L. Serafini and A. Tamilin, "DRAGO: Distributed reasoning architecture for the semantic web," *The Semantic Web: Research and Applications*, pp. 361–376, 2005.

[19] H. Wache, T. Voegele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, and S. Hübner, "Ontology-based integration of information-a survey of existing approaches," in *IJCAI-01 workshop: ontologies and information sharing*, vol. 2001, 2001, pp. 108–117.

[20] N. Choi, I. Song, and H. Han, "A survey on ontology mapping," *ACM Sigmod Record*, vol. 35, no. 3, pp. 34–41, 2006.

[21] P. Bouquet, F. Giunchiglia, F. Harmelen, L. Serafini, and H. Stuckenschmidt, "C-OWL: Contextualizing ontologies," in *Proc. ISWC*, 2003.

[22] B. Grau, B. Parsia, and E. Sirin, "Working with multiple ontologies on the semantic web," in *Proc. ISWC*, 2004.

[23] A. Maedche, B. Motik, N. Silva, and R. Volz, "MAFRA – a mapping framework for distributed ontologies," *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web*, 2002.

[24] A. Borgida and L. Serafini, "Distributed description logics: Assimilating information from peer sources," *J. on Data Semantics*, 2003.

[25] F. Heintz and P. Doherty, "Federated dyknow, a distributed information fusion system for collaborative UAVs," in *Proc. ICARCV*, 2010.