Semantically Grounded Stream Reasoning Integrated with ROS

Fredrik Heintz
Department of Computer and Information Science
Linköping University, Sweden
fredrik.heintz@liu.se

***Abstract*— High level reasoning is becoming essential to autonomous systems such as robots. Both the information available to and the reasoning required for such autonomous systems is fundamentally incremental in nature. A *stream* is a flow of incrementally available information and reasoning over streams is called *stream reasoning*. Incremental reasoning over streaming information is necessary to support a number of important robotics functionalities such as situation awareness, execution monitoring, and decision making.**

This paper presents a practical framework for semantically grounded temporal stream reasoning called DyKnow. Incremental reasoning over streams is achieved through efficient progression of temporal logical formulas. The reasoning is semantically grounded through a common ontology and a specification of the semantic content of streams relative to the ontology. This allows the finding of relevant streams through semantic matching. By using semantic mappings between ontologies it is also possible to do semantic matching over multiple ontologies. The complete stream reasoning framework is integrated in the Robot Operating System (ROS) thereby extending it with a stream reasoning capability.

I. Introduction

High level reasoning is becoming essential to autonomous systems such as robots as they become equipped with more sensors, actuators, and computational power and expected to carry out ever more complex missions in more and more challenging and unstructured environments.

Both the information available to and the reasoning required for such autonomous systems is fundamentally incremental in nature. A flow of incrementally available information is called a *stream* of information. As the number of sensors and other stream sources increases there is a growing need for incremental reasoning over streams in order 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 a number of important functionalities in autonomous systems such as situation awareness, execution monitoring, and decision making [1].

One major issue is grounding 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 related to the general problem of symbol grounding and anchoring [2], [3]. Existing systems do this 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.

The main contribution of this work is the practical realization of semantically grounded logic-based stream reasoning in the Robot Operating System (ROS). Incremental temporal reasoning using a metric temporal logic is achieved by evaluating temporal logical formulas over streams using progression. To address the problem of grounding stream reasoning in existing robotic systems 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. This is a significant extension of the stream-based knowledge processing middleware DyKnow [4], [5] which was realized using CORBA.

To the best of our knowledge there does not really exist any other system which provides similar stream reasoning functionality. The KnowRob [6] system is probably the closest match with its sophisticated and powerful knowledge processing framework. However, it does not support reasoning over streaming information and the support for temporal reasoning is limited.

II. Semantically Grounded Stream Reasoning

The semantically grounded stream reasoning framework consist of three main parts: a stream processing part, a stream reasoning part, and a semantic grounding part. Each part consists of a set of components. There are three types of components: engines, managers and coordinators. An *engine* takes a specification and carries out the processing as specified. A *manager* keeps track of related items and provides an interface to these. A *coordinator* provides a high-level functionality by coordinating or orchestrating other functionalities. An overview of the parts and the components is shown in Fig 1. The design is very modular as almost every component can be used independently.

The *stream processing* part is responsible for generating streams by for example importing, merging and transforming

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, the ELLIIT Excellence Center at Linköping-Lund in Information Technology, the Vinnova NFFP5 Swedish National Aviation Engineering Research Program, and the Center for Industrial Information Technology CENIIT.

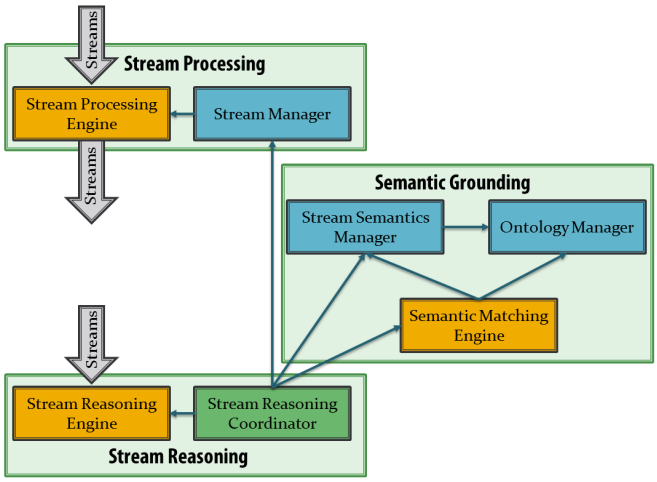

Fig. 1. An overview of the DyKnow components.

streams. The *Stream Manager* keeps track of all the streams in the system. Streams can either be generated by a stream processing engine or by some external program. A *Stream Processing Engine* takes a stream specification and generates one or more streams according to the specification.

The *stream reasoning* part is responsible for evaluating logical formulas over streams. A *Stream Reasoning Engine* takes a logical formula and a stream of states and evaluates the formula over this stream. A *Stream Reasoning Coordinator* takes a logical formula, finds all the relevant streams needed to evaluate the formula, creates a stream specification for generating a single stream of states from all the relevant streams, and instructs the stream reasoning engine to evaluate the formula over the stream as it is generated by a stream processing engine. There are two different ways of grounding a formula, which in our case means finding the relevant streams for a formula. Either *syntactic grounding*, where the formula explicitly states the names of the streams, or *semantic grounding*, where the formula is written using concepts from an ontology which are then matched against the available streams.

The *semantic grounding* part is responsible for finding streams based on their semantics relative to a common ontology. The *Stream Semantics Manager* keeps track of semantically annotated streams, where an annotation describes the semantic content of a stream. The *Ontology Manager* keeps track of the ontology which provides a common vocabulary. The *Semantic Matching Engine* finds all streams whose semantic annotation matches a particular ontological concept. The semantic grounding part is used by the stream reasoning part to find the relevant streams in order to evaluate a logical formula. When the relevant streams have been found they are merged together and synchronized by the stream processing part to create the stream of states that is required to evaluate the formula.

The following sections describe each of the parts in detail.

III. Stream Processing

Stream processing has a long history [7] and data stream management systems such as XStream [8], Aurora [9], and Borealis [9] provide continuous query languages supporting filters, maps, and joins on streams. There are also commercial stream processing tools such as RTMaps [10]. The stream processing part of DyKnow provides similar functionality specially designed for stream reasoning. One major difference is that time is essential in stream reasoning while data stream management systems usually treat time as any other value.

A *stream* is a sequence of time-stamped tuples which becomes incrementally available. A stream is an abstraction which has many realizations in different systems such as topics in ROS and event channels in CORBA. A tuple in a stream has one or more *fields* and is often called a *sample* or a *stream element*. In ROS each tuple is a *message*.

Each tuple is associated with two time stamps, the valid time and the available time. The *valid time* is the specific time-point when the information in the tuple is valid. The *available time*, is the time when the tuple is ready to be processed by the receiving process after having been transmitted through a potentially distributed system. The available time is unique for each stream, i.e. there may not be two tuples in a stream with the same available time. 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. For example, assume a picture was taken at time t, an object was extracted from this image and made available in a stream at time-point $t + 5$ then the valid time for the object would be t and the available time $t + 5$.

A *stream specification* is a declarative description of a stream. It can include constraints on the stream such as start and end times, maximum delay, sample period, and sample period deviation. For example, if the maximum delay allowed is 100ms then the difference between the available time and the valid time may not be greater than 100ms. If the sample period is 200ms and the sample period deviation allowed is 50ms then the difference in valid times between two consecutive samples has to be in the range 150ms to 250ms. If the sample period deviation is 0 then the sample period has to be exact.

The DyKnow stream processing functionality currently supports selecting values from a stream, merging multiple streams, and synchronizing multiple streams. These are the basic operations on streams required for stream reasoning. Further, it is also possible to have user defined operations on streams called *computational units*.

The stream specification language supported by DyKnow for ROS has the following grammar:

```
DECL ::= SOURCE_DECL | SINK_DECL
| COMPUNIT_DECL | STREAM_DECL
DECLS ::= DECL | DECL SEMICOLON DECLS
SOURCE_DECL ::= source TYPE_DECL SOURCE_NAME
SINK_DECL ::= sink STREAM
```

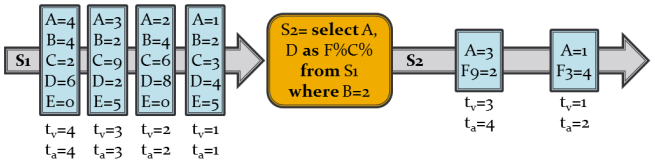

Fig. 2. Stream processing example: select

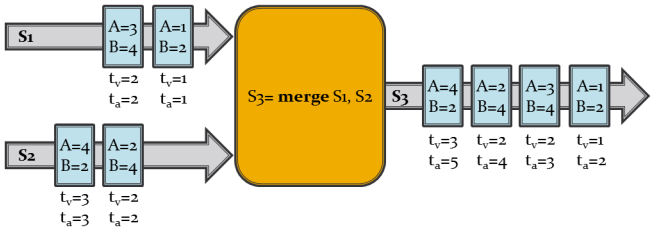

Fig. 3. Stream processing example: merge

```
COMPUNIT_DECL ::=
  compunit BASIC_TYPE COMPUNIT_NAME
  LP TYPE_DECL (COMMA TYPE_DECL)* RP
STREAM_DECL ::= stream NAME EQ STREAM
TYPE_DECL ::= BASIC_TYPE | COMPLEX_TYPE
BASIC_TYPE ::= NAME COLON TYPE
COMPLEX_TYPE ::=
LP BASIC_TYPE (COMMA BASIC_TYPE)* RP
TYPE ::= int | float | string | boolean
STREAM ::= STREAM_TERM with STREAM_CONSTRAINTS
STREAMS ::= STREAM | STREAM COMMA STREAMS
STREAM_TERM ::= STREAM_NAME
| SOURCE_NAME
| COMP_UNIT_NAME LP STREAMS RP
| select SELECT_EXPRS from STREAM
  where WHERE_EXPRS
| merge LP STREAMS RP
| sync LP STREAMS RP
SELECT_EXPR ::= FIELD_ID (as PSTRING)?
SELECT_EXPRS ::= SELECT_EXPR
| SELECT_EXPR COMMA SELECT_EXPRS
WHERE_EXPR ::= FIELD_ID EQ VALUE
WHERE_EXPRS ::= WHERE_EXPR
| WHERE_EXPR and WHERE_EXPRS
FIELD_ID ::= STRING | STRING DOT FIELD_ID
PSTRING ::= STRING
| STRING? PERCENT FIELD_ID PERCENT PSTRING?
STREAM_CONSTRAINTS ::= STREAM_CONSTRAINT
| STREAM_CONSTRAINT COMMA STREAM_CONSTRAINTS
STREAM_CONSTRAINT ::=
  start_time EQ NUMBER | end_time EQ NUMBER
| max_delay EQ NUMBER | sample_period EQ NUMBER
| sample_period_deviation EQ NUMBER
```

The informal semantics of "**select** *select_expression* **from** *stream* **where** *where_expression* **with** *stream_constraints*" is a stream that contains fields according to the *select_expression* for every tuple in *stream* that satisfies the *where_expression* and such that the resulting stream satisfies the *stream_constraints*. Fig 2 shows an example of selecting the fields A and D from a stream S1 when the value of the field B is 2 and renaming D to Fx where x is the value of the field C.

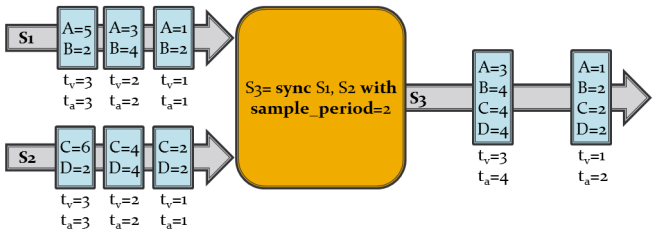

Fig. 4. Stream processing example: sync

The informal semantics for "**merge** *streams* **with** *stream_constraints*" is a stream that contains every tuple from every stream in *streams* as long as the resulting stream satisfies the *stream_constraints*. An example of merge is shown in Fig 3, where two streams S1 and S2 are merged into a stream S3. One important observation is that when two input tuples are available to the merge process at exactly the same time it is non-deterministic which order they are processed. In the resulting stream, they will have different available times.

The informal semantics for "**sync** *streams* **with** *stream_constraints*" is a stream that contains tuples which are constructed by joining a tuple from each stream in *streams* to a new tuple so that all the tuples are valid at the same time and the resulting stream satisfies the *stream_constraints*. The synchronization time-points, i.e. the time-points for which a synchronized value will be computed is either every time-point for which an input tuple exists or determined by the start time and sample period for the resulting stream. If no start time is given, then the start time will be the minimum valid time of the input streams. To approximate the value at a time-point t for a stream without a tuple with a valid time of t the tuple with the latest valid time before t will be used. This corresponds to assuming that all value changes are contained in the stream. Other approximation policies are also possible. Fig 4 shows the synchronization of streams S1 and S2 into a stream S3 with the sample period of 2.

IV. Logic-Based Temporal Stream Reasoning

One technique for incremental temporal 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) [11], 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 [12]. 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 [13] and to express conditions for when to hypothesize the existence and classification of observed objects in an anchoring framework [14]. In execution monitoring, for example, suppose that 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 monitor and detect violations of this specification: $\Box \forall \mathit{uav} : (\mathsf{power}(\mathit{uav}) > M \rightarrow \mathsf{power}(\mathit{uav}) < f \cdot M \,\mathsf{U}_{[0,\tau]}\, \Box_{[0,\tau']}\, \mathsf{power}(\mathit{uav}) \leq M)$.

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 [13]. 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 common formulas [4].

V. Semantic Grounding

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 represented by the valid time of the sample.

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 [4].

In a system consisting of streams, a natural approach is to syntactically map each feature to a single stream, we call this *syntactic grounding*. 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 based on their semantics is needed, we call this *semantic grounding*. 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 still 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 understandable. This allows the system to reason about the correspondence between stream content and symbols used in logical formulas. 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 approach to semantic grounding uses semantic web technologies to define and reason about ontologies. Ontologies provide suitable support for creating machine understandable domain models [15]. Ontologies also provide reasoning support and support for semantic mapping which is necessary for the grounding of symbols to streams from multiple robotic systems.

To represent ontologies we use the Web Ontology Language (OWL) [16]. Features, objects, and sorts are represented in 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 a feature

might be an arbitrary relation.

To represent the semantic content of streams in terms of features, objects, and sorts we have defined a semantic specification language called SSL [17]. This is used to annotate the semantic content of streams.

Finally, a semantic matching algorithm has been developed which finds all streams which contain information relevant to a concept from the ontology, such as a feature. This makes it possible to automatically find all the streams that are relevant for evaluating a temporal logical formula. 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. By introducing semantic mapping between ontologies from different robotic systems (e.g. C-OWL [18]) and reasoning over multiple related ontologies (e.g. DDL [19]) it is even possible to find relevant streams distributed among multiple robots [17].

VI. Integration with ROS

ROS is an open-source framework for robot software development which allows interfaces and services to be clearly specified [20]. Software written for ROS is organized into *packages* which contain nodes, libraries, and configurations. *Nodes* represent computational processes in the system and are written using language specific client libraries. These nodes communicate in two ways. First, by passing structured messages on *topics* using XML-RPC where topics can be seen as named buses to which nodes can subscribe. Second, by using request/reply communication through *services*. To manage services and topics there is a common ROS Master, a standard ROS component providing registration and lookup functionality.

To integrate semantically grounded stream reasoning with ROS each component in Fig 1 is realized as a ROS node providing a number of related services. The details of the most important services are provided below.

One interesting aspect is the realization of streams. Streams are naturally realized as topics. However, there are two issues. First, topics are strongly typed in ROS while a single type for representing stream tuples is required to support the creation and processing of states containing arbitrary fields. Second, ROS topics do not provide strong guarantees about delays or reorderings of messages while according to the semantics of streams the constraints associated with a stream should be satisfied at the receiving end.

The first issue is handled by introducing a new message type called Sample which is used to represent an arbitrary stream element. It contains a common header, the valid time and available time of the element, and the fields. Each field consists of a string representation of its type, the name of the field, and the value of the field represented as a string. To automatically handle the conversion from ROS types to Sample and back, a script has been written which generates the necessary conversion functions directly from ROS message types.

The second issue is handled by introducing a client side helper class called StreamProxy which subscribes to a topic and creates a stream that satisfies the associated constraints. This could for example include reordering samples and throw away samples that have been delayed for too long. An overview of the realization of streams is shown in Fig 5.

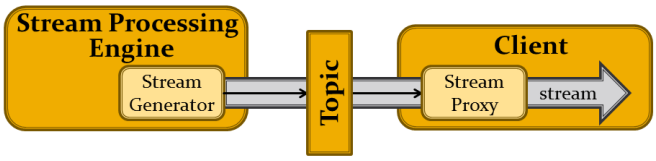

Fig. 5. Realizing a stream using a topic.

A major benefit of the design is that any topic can be treated as a stream by the framework since the conversion from a ROS specific type to the general stream type is handled by a StreamProxy. This makes it possible to use any method to generate streams which can then be used by the framework in the same manner as streams generated by the framework itself.

A. Services

1) Stream Processing Engine: A stream processing engine takes specifications in the DyKnow stream specification language and creates streams according to those. There can be many stream processing engines in a system. Each stream processing engine also implements some of the stream manager services since they manage their own local streams.

The following service specifications are written on the form Name(parameters) : return values. The message types are named to be self explanatory, instead of explicitly defined. If a type ends with [] then it is a vector type.

```
StreamCreateFromSpec(StreamSpec spec)
 : ExitStatus exit_status,
   string stream_topic,
   string stream_constraint_violation_topic
StreamDestroy(string stream_name)
 : ExitStatus exit_status
ListStreams()
 : ExitStatus exit_status,
   string[] stream_names
```

2) Stream Manager: The Stream Manager keeps track of all the streams in a system. There is only one Stream Manager in the system. A stream that has been created by a Stream Processing Engine can be registered with the Stream Manager. It is also possible to create new streams from stream specifications. The Stream Manager then selects an appropriate stream processing engine that will process the specification. The Stream Manager also keeps track of the mapping between topic names and stream names. It is possible to configure DyKnow to use a convention for mapping between topic names and stream names to remove the need for storing this explicitly.

```
StreamRegister(string engine_name,
               string stream_name)
 : ExitStatus exit_status
StreamDeregister(string stream_name)
 : ExitStatus exit_status
StreamCreateFromSpec(StreamSpec spec)
 : ExitStatus exit_status,
   string stream_topic,
```

```
   string stream_constraint_violation_topic
StreamDestroy(string stream_name)
 : ExitStatus exit_status
StreamGetTopic(string stream_name)
 : ExitStatus exit_status,
   string topic_name
ListStreams()
 : ExitStatus exit_status,
   string[] stream_names
```

3) Ontology Manager: The Ontology Manager keeps track of the ontology used in the system. It has services for adding concepts and adding instances of concepts. It also has services for getting the concepts of an instance. for getting instances of a concept, and for evaluating OWL queries. The service calls are translated into queries to an OWL ontology.

```
ConceptCreate(string concept_name,
              string[] parent_concepts)
 : ExitStatus exit_status
ConceptGetInstances(string concept_name)
 : ExitStatus exit_status,
   string[] instances
ConceptDestroy(string concept_name)
 : ExitStatus exit_status
InstanceCreate(string instance_name,
               string[] concept_names)
 : ExitStatus exit_status
InstanceGetConcepts(string instance_name)
 : ExitStatus exit_status,
   string[] concepts
InstanceDestroy(string instance_name)
 : ExitStatus exit_status
```

4) Stream Semantics Manager: The Stream Semantics Manager keeps track of the semantic annotation of streams. Any stream can be annotated by providing a SSL specification of its semantic content.

```
AddSemanticSpecification(string ssl_statement)
 : ExitStatus exit_status
```

5) Semantic Matching Engine: The Semantic Matching Engine is responsible for matching concepts from the ontology, managed by the Ontology Manager, to annotated streams, managed by the Stream Semantics Manager. It is a separate component since it can do semantic matching between any ontology and any collection of stream specifications annotated using that ontology. The service for finding all matching streams for a specific concept, which might be quantified, from the ontology returns a set of stream specifications since it might be necessary to create a new stream rather than to directly use an existing stream.

```
FindAllMatchingStreams(string concept)
 : ExitStatus exit_status,
   string[] stream_specifications
```

6) Stream Reasoning Engine: The Stream Reasoning Engine evaluates metric temporal logical formulas using progression. To evaluate multiple formulas over the same domain and the same stream of states a *formula group* is introduced which consists of a set of formulas, a domain, and the name of an appropriate state stream topic. Each tuple in the state stream must contain one field for each ground feature that appears in any of the formulas. If not, then the progression will fail. The domains, i.e. the set of objects, for the sorts are fixed for each formula group. To support the extraction of feature and sort symbols from a formula the Stream Reasoning Engine provides a service for this.

```
FormulaGroupCreate(string formula_group_name)
 : ExitStatus exit_status,
   int formula_group_id
FormulaGroupAddFormula(int formula_group_id,
                       string formula_name,
                       string formula)
 : ExitStatus exit_status,
   int formula_id
FormulaGroupEvaluate(int formula_group_id,
                     Domain[] domains,
                     string state_stream,
                     string result_topic)
 : ExitStatus exit_status
FormulaGroupExtractSymbols(int formula_grp_id)
 : ExitStatus exit_status,
   Symbol[] symbols
FormulaGroupDestroy(int formula_group_id)
 : ExitStatus exit_status
```

7) Stream Reasoning Coordinator: The Stream Reasoning Coordinator orchestrates the stream reasoning, the semantic matching, and the stream processing necessary to evaluate temporal logical formulas. The coordination required depends on whether the formula is expressed with explicit stream names or if it is expressed using feature concepts from an ontology. In the first case there is no need to do semantic matching which is required in the second case. A benefit of the Stream Reasoning Coordinator is that a user only needs to interact with a single service to evaluate a formula. A detailed description of this process is described in the next section.

```
EvaluateFormulaBlocking(string formula)
 : ExitStatus exit_status,
   bool result
EvaluateFormulaNonBlocking(string formula,
                           string result_topic)
 : ExitStatus exit_status
```

VII. Case Study: Execution Monitoring

Execution monitoring is an important application for stream reasoning. It can for example be used to make sure that the execution of a plan is progressing as expected and that a robot does not violate any restrictions placed on it, such as going too fast, flying too high or too low, and so on.

This section describes the process of evaluating a temporal logical formula in detail. The description shows the cooperation among the components and how they all contribute.

1) A user calls an EvaluateFormula service implemented by the Stream Reasoning Coordinator with the formula f. The formula could for example be generated as part of the planning process of the robot [13].
2) The Semantic Reasoning Coordinator calls the FormulaGroupCreate service to create a group of formulas that are all evaluated over the same state stream, and then the FormulaGroupAddFormula service with the formula f to add it to the group, both are implemented by the Stream Reasoning Engine.
3) The Stream Reasoning Coordinator extracts the sorts and features from the formula by calling the FormulaGroupExtractSymbols service, also implemented by the Stream Reasoning Engine.
4) For each sort, the Stream Reasoning Coordinator computes the domain of the sort by getting all the instances

of the sort by calling the ConceptGetInstances service implemented by the Ontology Manager.

5) For each feature, the Stream Reasoning Coordinator requests all matching streams from the Semantic Matching Engine by calling the FindAllMatchingStreams service.
6) If every feature has at least one matching stream then the Stream Reasoning Coordinator takes the individual stream specifications and creates a stream specification which synchronizes the streams for the individual features into a single stream of states.
7) The Semantic Reasoning Coordinator calls the FormulaGroupEvaluate service with the domains of the sorts and the name of the topic on which the state stream will be delivered. The Stream Reasoning Engine subscribes to the state stream topic. It is now ready to start evaluating the formula.
8) The Stream Reasoning Coordinator subscribes to the result topic of the formula group and when it receives a true or false message it destroys the state stream and returns the result to the client by completing its call to the coordinator.
9) The Stream Reasoning Coordinator creates the state stream by calling the StreamCreateFromSpec service implemented by the Stream Manager.
10) The Stream Manager calls the StreamCreateFromSpec service implemented by a Stream Processing Engine.
11) The Stream Processing Engine sets up the appropriate topic subscriptions, processes the messages from these subscriptions according to the specification, creates synchronized states, and publishes them on the state stream topic.
12) For every state in the state stream, the Stream Reasoning Engine progresses the formula over that state. If the formula is progressed to either true or false a message is sent on the result topic of the formula group.
13) During the process the Stream Processing Engine might find that one of the constraints associated with the stream specification is violated. In this case, a message is sent on the constraint violation topic. The coordinator subscribes to this topic and tries to handle the situation if possible. Otherwise the call from the client is terminated and an error message is returned.

The described execution monitoring approach has been integrated with a planner that generates plans together with execution monitoring formulas and used in UAV applications [13]. The formulas associated with a plan are automatically added and removed appropriately during the execution of the plan, thereby ensuring the proper monitoring of the execution of the plan.

Other case studies using DyKnow are for example traffic monitoring [21], distributed fusion for collaborative UAVs [22], and anchoring [14].

VIII. Performance Evaluation

Estimating the performance of a complex stream processing and stream reasoning framework like DyKnow is an important and multi-faceted problem. This section briefly discusses the most important aspects of the performance for each part in the framework.

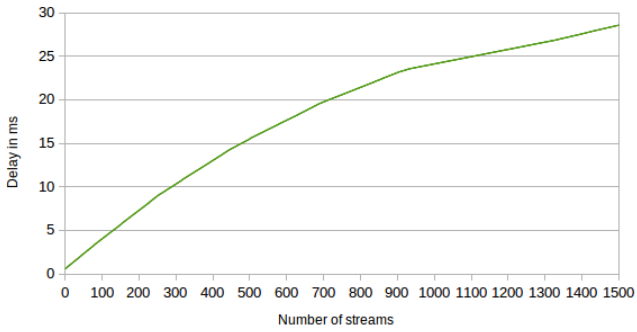

Fig. 6. Stream delays when varying the number of subscriptions.

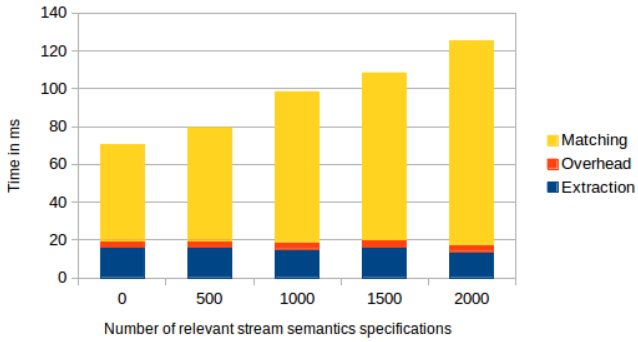

Fig. 7. Performance when varying the number of stream specifications.

A. Efficiency of stream processing

Estimating the performance of the stream processing is basically about measuring the throughput and delays of streams and stream processes. Since streams are implemented as ROS topics, the performance of streams are equal to the performance of ROS topics, which has a small overhead of a few micro seconds per subscriber and message as can be seen from Fig. 6. For merging and synchronization the overhead is linear in the number of input streams and has been measured to be about 50-100 microseconds per stream on an 8 core Intel i7 CPU. Network delays are usually the limiting factor, not the processing time.

B. Efficiency of semantic grounding

The performance of finding all matching streams for a given feature, i.e. feature concept in the ontology, depends both on the size of the ontology and the number of stream specifications. Our experiments show that the cost is roughly linear in the number of feature concepts in the ontology and in the number of stream specifications [17]. Fig 7 shows an experiment where the number of relevant stream semantics specifications for a concept was varied and the time to match this concept against the specification was measured. The three categories are the time used for parsing the formula,

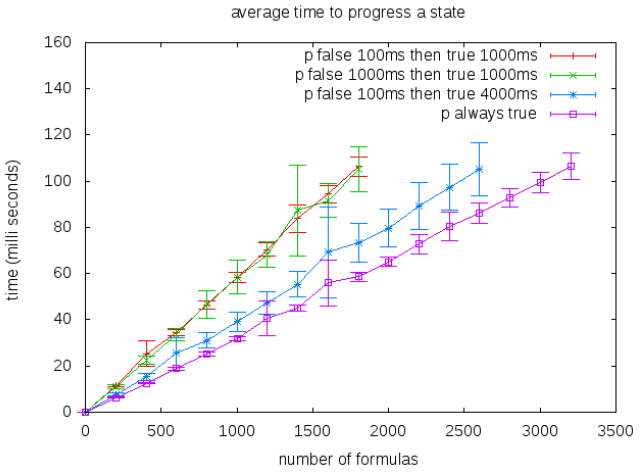

Fig. 8. Average progression time: $\Box \neg p \rightarrow \Diamond_{[0,1000]} \Box_{[0,999]} p$.

the ROS communication overhead for communicating the parsing result to the semantic manager node, and the time used by the semantic matching itself. It shows that the time used for matching increases linearly from about 50ms for 1 relevant specification to 110ms for 2000 relevant specifications on an 8 core Intel i7 CPU.

C. Efficiency of temporal stream reasoning

To be practically useful it is important that the stream reasoning is fast. Even though the size of a progressed formula may grow exponentially in the worst case, the growth can always be limited by introducing metric bounds on the temporal operators. Our experiments show that even with the worst possible inputs complex formulas can be evaluated in less than 1 millisecond per state and formula on a 1.4GHz Pentium M. One example formula is $\Box \neg p \rightarrow \Diamond_{[0,1000]} \Box_{[0,999]} p$, corresponding to the fact that if p is false, then within 1000 ms, there must begin a period lasting at least 1000 ms where p is true. To estimate the cost of evaluating this formula, it was progressed through several different state streams corresponding to the best case, the worst case, and two intermediate cases. A new state in the stream was generated every 100 ms, which means that all formulas must be progressed within this time limit or the progression will fall behind. Fig. 8 shows that 100 ms is sufficient for the progression of between 1500 and 3000 formulas of this form on the computer on-board our UAV (a 1.4GHz Pentium M), depending on the state stream.

IX. Conclusion

DyKnow is a pragmatic semantically grounded stream reasoning framework supporting efficient incremental temporal logical reasoning over streams of information. The semantic grounding allows the automatic collection and synchronization of the streams needed to evaluate a particular formula based on the semantics of the symbols in the formula and the content of the available streams. This greatly increases the value of DyKnow as it simplifies dynamic applications where streams and concepts are added and removed at runtime. By integrating the framework with ROS we make this very powerful processing and reasoning capability available to a wide range of robotic systems. DyKnow is designed for and intended to be used in sophisticated autonomous systems where there is a need to dynamically reason about the system or its environment using streams of information.

We are currently working on several extensions. For example, supporting the automatic federation and fusion of streams generated by multiple robotic systems; automatically changing the set of streams used to evaluate a formula when the set of available streams is dynamically changing during the evaluation; and extending the temporal reasoning with support for spatial reasoning to allow spatio-temporal stream reasoning which is important for many applications.

Semantically grounded stream reasoning is an important step towards a new era of intelligent robots and systems that meets the demanding needs of the future.

References

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

[2] S. Harnad, "The symbol-grounding problem." *Physica D*, no. 42, 1990.

[3] S. Coradeschi and A. Saffiotti, "An introduction to the anchoring problem," *Robotics and Autonomous Systems*, vol. 43, no. 2–3, 2003.

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

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

[6] M. Tenorth and M. Beetz, "KnowRob – A Knowledge Processing Infrastructure for Cognition-enabled Robots. Part 1: The KnowRob System," *Int. J. of Robotics Research*, vol. 32, no. 5, 2013.

[7] R. Stephens, "A survey of stream processing," *Acta Informatica*, vol. 34, no. 7, 1997.

[8] L. Girod, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden, "Xstream: a signal-oriented data stream management system," in *Proc. ICDE*, 2008.

[9] U. Çetintemel et. al., "The Aurora and Borealis Stream Processing Engines," in *Data Stream Management: Processing High-Speed Data Streams*, 2007.

[10] Intempora, "Rtmaps, a rapid and modular environment for your real-time applications." [Online]. Available: http://www.intempora.com/

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

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

[13] 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.

[14] F. Heintz, J. Kvarnström, and P. Doherty, "Stream-based hierarchical anchoring," *Künstliche Intelligenz*, vol. 27, no. 2, pp. 119–128, 2013.

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

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

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

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

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

[20] 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.

[21] F. Heintz, P. Rudol, and P. Doherty, "From images to traffic behavior – a UAV tracking and monitoring application," in *Proc. Fusion*, 2007.

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