

Institutionen för Datavetenskap
Department of Computer and Information Science

Master's thesis

Semantic Matching for Stream Reasoning

by

Zlatan Dragisic

LIU-IDA/LITH-EX-A-11/041-SE

2011-10-03



Linköpings universitet

Institutionen för Datavetenskap
Department of Computer and Information Science

Master's thesis

Semantic Matching for Stream Reasoning

by

Zlatan Dragisic

LIU-IDA/LITH-EX-A-11/041-SE

2011-10-03

Supervisor: Fredrik Heintz
Department of Computer and Information Science
KPLAB - Knowledge Processing Lab

Examiner: Fredrik Heintz
Department of Computer and Information Science
KPLAB - Knowledge Processing Lab

På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ick-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: <http://www.ep.liu.se/>

Abstract

Autonomous system needs to do a great deal of reasoning during execution in order to provide timely reactions to changes in their environment. Data needed for this reasoning process is often provided through a number of sensors. One approach for this kind of reasoning is evaluation of temporal logical formulas through progression. To evaluate these formulas it is necessary to provide relevant data for each symbol in a formula. Mapping relevant data to symbols in a formula could be done manually, however as systems become more complex it is harder for a designer to explicitly state and maintain this mapping. Therefore, automatic support for mapping data from sensors to symbols would make system more flexible and easier to maintain.

DyKnow is a knowledge processing middleware which provides the support for processing data on different levels of abstractions. The output from the processing components in DyKnow is in the form of streams of information. In the case of DyKnow, reasoning over incrementally available data is done by progressing metric temporal logical formulas. A logical formula contains a number of symbols whose values over time must be collected and synchronized in order to determine the truth value of the formula. Mapping symbols in formula to relevant streams is done manually in DyKnow. The purpose of this matching is for each variable to find one or more streams whose content matches the intended meaning of the variable.

This thesis analyses and provides a solution to the process of semantic matching. The analysis is mostly focused on how the existing semantic technologies such as ontologies can be used in this process. The thesis also analyses how this process can be used for matching symbols in a formula to content of streams on distributed and heterogeneous platforms. Finally, the thesis presents an implementation in the Robot Operating System (ROS). The implementation is tested in two case studies which cover a scenario where there is only a single platform in a system and a scenario where there are multiple distributed heterogeneous platforms in a system.

The conclusions are that the semantic matching represents an important step towards fully automatized semantic-based stream reasoning. Our solution also shows that semantic technologies are suitable for establishing machine-readable domain models. The use of these technologies made the

semantic matching domain and platform independent as all domain and platform specific knowledge is specified in ontologies. Moreover, semantic technologies provide support for integration of data from heterogeneous sources which makes it possible for platforms to use streams from distributed sources.

Contents

List of Figures	vii
List of Tables	ix
1 Introduction	1
1.1 Background	1
1.2 Goal	2
1.3 Thesis outline	3
2 DyKnow	5
2.1 Overview	5
2.2 Basic Concepts	6
2.3 Architecture in ROS	8
2.3.1 ROS	8
2.3.2 ROS implementation	9
2.4 DyKnow Federations	11
2.4.1 Components	12
2.5 Summary	12
3 Semantic Technologies	15
3.1 Semantic Web	15
3.2 RDF/RDFS	16
3.2.1 Syntax	17
3.3 Ontologies	19
3.3.1 OWL	20
3.3.2 Semantic mappings	24
3.4 Summary	27
4 Analysis	29
4.1 Semantic stream representation	29
4.2 Matching symbols to topics	34
4.3 Integrating data from multiple platforms	39
4.4 Design	40
4.5 Related work	42

5	Implementation	47
5.1	Introduction	47
5.2	Proposed solution	47
5.3	Design	48
5.4	Matching a formula to topics	51
5.5	Multiple Platform Scenario	53
5.6	Integration	55
5.7	Summary	57
6	Case studies	59
6.1	Single platform scenario	59
6.2	Multiple platform scenario	64
6.3	Discussion	71
7	Performance evaluation	73
7.1	Test cases	73
7.2	Test setup	74
7.3	Results	75
7.4	Conclusion	83
8	Conclusion	85
8.1	Summary	85
8.2	Future work	86
	Bibliography	89
A	Acronyms	93
B	Ontologies	95
B.1	RDF/XML representation of the ontology for platform 1 . . .	95
B.2	RDF/XML representation of the ontology for platform 2 . . .	101
B.3	RDF/XML representation of the ontology for platform 3 . . .	103
C	Topic Specifications	107
C.1	XML representation of topic specifications for platform 1 . .	107
C.2	XML representation of topic specifications for platform 2 . .	108
C.3	XML representation of topic specifications for platform 3 . .	108

List of Figures

2.1	An example of a ROS computation graph.	9
2.2	DyKnow architecture in ROS.	10
2.3	An overview of the components of a DyKnow federation [17].	11
3.1	Visualization of the ontology for the single platform scenario.	22
3.2	Visualization of the ontology of platform 2.	23
3.3	Visualization of the ontology of platform 3.	24
4.1	Process of semantic matching.	34
4.2	DyKnow architecture in ROS.	41
5.1	Components in ROS implementation of DyKnow.	48
5.2	Multiple Platform scenario - example.	54
6.1	Visualization of the ontology for the single platform scenario.	60
6.2	Visualization of the ontology for platform 2.	65
6.3	Visualization of the ontology for platform 3.	65
7.1	Varying number of concepts.	76
7.2	Varying number of irrelevant individuals.	77
7.3	Varying number of relevant individuals.	78
7.4	Varying number of irrelevant topic specifications.	79
7.5	Varying number of relevant topic specifications.	80
7.6	Varying number of features in the formula.	81
7.7	Comparing quantified and non-quantified versions of a formula.	82

List of Tables

6.1	Unary relations and argument types.	60
6.2	Binary relations and argument types.	61
6.3	Ternary relation and argument types.	61
6.4	Unary relations and argument types for platform 2.	65
6.5	Unary relations and argument types for platform 3.	66
6.6	Binary relations and argument types for platform 3.	66
7.1	Varying number of concepts.	75
7.2	Varying number of irrelevant individuals.	76
7.3	Varying number of relevant individuals.	77
7.4	Varying number of irrelevant topic specifications.	78
7.5	Varying number of relevant topic specifications.	79
7.6	Varying number of features in a formula.	80
7.7	Comparing quantified and non-quantified versions of a formula.	82

Listings

3.1	Bridge rule in XML.	25
3.2	Bridge rules in XML.	26
4.1	A topic specifying the features Altitude and Speed for the sort UAV.	31
4.2	Formal grammar for SSL_T	31
4.3	Topic specifications in SSL_T	32
4.4	Topic specifications in SSL_T	33
4.5	DTD for the SSL_T XML syntax.	35
4.6	Topic specifications in SSL_T	36
5.1	Possible topics for feature Behind[car1, car2].	52
5.2	CreateGroundingService service and relevant messages.	55
5.3	State Processor service and relevant messages.	56
6.1	Topic specifications in SSL_T	61
6.2	Output from the Knowledge Manager (KM) for formula 1.	62
6.3	Output from the KM for formula 2.	63
6.4	Output from the KM for formula 3.	64
6.5	Mappings in XML.	66
6.6	Topic specifications for platform 2 in SSL_T	67
6.7	Topic specifications for platform 3 in SSL_T	67
6.8	Output from the KM for formula 1 in distributed system.	68
6.9	Output from the KM for formula 2 in distributed system.	69
6.10	Output from the KM for formula 3 in distributed system.	71
B.1	RDF/XML representation of the ontology for platform 1.	95
B.2	RDF/XML representation of the ontology for platform 2.	101
B.3	RDF/XML representation of the ontology for platform 3.	103
C.1	Topic specifications in XML.	107
C.2	Topic specifications for platform 2 in XML.	108
C.3	Topic specifications for platform 3 in XML.	108

Chapter 1

Introduction

This introductory chapter presents the background and goals of the thesis. The chapter also describes the use-case scenarios used in the rest of the thesis.

1.1 Background

Autonomous systems do a great deal of reasoning during execution. Some of the functionalities in the system, such as planning, execution monitoring or diagnosis require that the reasoning is done over incrementally available information. For example, when doing execution monitoring it is necessary to continuously gather information from the environment to see if the changes in the environment make the current plan invalid. The data for this kind of reasoning is provided by sensors. However, there exists a large gap between the numerical noisy data provided by the sensors and the exact symbolic information often needed for reasoning.

DyKnow is a stream-based knowledge processing middleware framework developed in the Artificial Intelligence and Integrated Computer Systems Division (AIICS) within the Department of Computer and Information Science (IDA) at Linköping University [17]. The main idea behind DyKnow is to bridge the gap between sensing and reasoning. Therefore, DyKnow provides support for accepting inputs at different levels of abstraction and from distributed sources. The output is provided in the form of sets of streams which represent for example objects, attributes, events and relations in the system [17].

In the fall semester of 2010 the CogRob group was formed. The aim of the group's projects is to make a Robot Operating System (ROS) implementation of DyKnow. ROS was chosen as a candidate for stream reasoning because it is language independent, thin and can be used for large runtime systems. Also, ROS is supported by a large community working on different types of projects and problems in robotics. Given the modularity of the

software developed for ROS this poses an interesting prospect for further extensions of robotic systems developed at AIICS.

The ROS implementation of DyKnow should allow evaluation of spatio-temporal logical formulas over streams of information coming from multiple sources in a distributed system. The evaluation of these formulas is needed in order to support certain functionalities in a system, i.e. run-time verification, complex event detection and spatio-temporal reasoning in general.

A temporal logical formula contains a number of variables whose values over time must be collected and synchronized in order to determine the truth value of the formula. In a system consisting of streams a natural approach is to map each variable to a single stream. This works very well when there is a stream for each variable 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 variables in a formula to streams in a system is needed. The purpose of this matching is for each variable to find one or more streams whose content matches the intended meaning of the variable. This is a form of semantic matching between logical variables and contents of streams. By adding this semantic matching DyKnow would support semantic stream reasoning. The process of matching variables to streams in a system requires that the meaning of the content of the data streams is encoded and that this encoding can be used for matching the intended meaning of variables with the actual content of streams.

1.2 Goal

The goal of this Master's thesis is twofold. The first goal is to analyze and provide a solution to the problem of semantic matching of the intended meaning of symbols (variables) in logical formulas and the content of streams. The analysis of the problem of semantic matching focuses on how existing technologies for representing semantics can be used in the matching process.

The second goal is to extend the ROS implementation of DyKnow with support for this type of semantic matching. The extension of the ROS implementation provides the following functionalities:

- it allows explicit definitions of the meaning of streams in ROS
- it uses these definitions to determine which streams are relevant for variables in the formula

We suggest two scenarios which describe the need and use of semantic matching for stream reasoning. These scenarios are used to test our solutions.

Single Platform Scenario

The Single Platform Scenario describes a situation where a system only has one platform and therefore when it comes to the formula evaluation only data streams from this platform can be used. As an example of this scenario could be execution monitoring of an Unmanned Aerial Vehicle (UAV). One way to achieve execution monitoring is to define a set of logical formulas which capture the desired state of the system (i.e. the UAV should never be closer than 3 meters from any building). If any of these formulas is evaluated to false during execution then that means that the system is in an undesired state and steps should be taken to correct this. Given that the data is provided in the set of streams, an important step in the automation of the process of formula evaluation is to provide a way of matching the intended meaning of variables in the formula to the content of data streams.

Multiple Platform Scenario

This scenario deals with multiple platforms where each platform has its own instance of DyKnow. Each platform should be able to cooperate with other platforms in the system by using information which is distributed among these platforms. The platforms can be either homogeneous or heterogeneous with respect to the way streams are specified on the different platforms. This puts the requirement that platforms are capable of querying both heterogeneous and homogeneous sources.

An example of this scenario in the domain of UAVs is the situation where two unmanned aerial vehicles are monitoring traffic violations on a road. The road is divided into two regions, one for each UAV. The use of multiple platforms helps reduce uncertainty while the division of the road makes it possible to monitor multiple potential traffic violations at the same time. If we assume that the monitored traffic violation started in one region and ended in the other one then in order for some platform to determine if the violation actually happened it needs to cooperate with the other platform which is responsible for the region where the violation ended [18]. The reuse of data from distributed sources in this scenario would make it possible for platforms to follow the traffic violation from the beginning to the end regardless of which region they were assigned initially. Therefore the semantic matching should be capable of matching variables in the formula to both local and distributed streams.

1.3 Thesis outline

Chapter 2 gives an overview of the DyKnow knowledge processing middleware. This includes basic concepts, architecture including the ROS architecture and DyKnow Federations.

Chapter 3 describes the basic concepts of semantic technologies.

Chapter 4 analyses the problem of semantic matching.

Chapter 5 presents the design of the semantic matching component and the way it can be integrated into the ROS implementation of DyKnow.

Chapter 6 presents two different case studies which cover the functionalities and results of the implementation.

Chapter 7 evaluates the performance of the semantic matching component in the ROS implementation of DyKnow.

Chapter 8 gives a short discussion and summary of the thesis together with possible extensions.

Chapter 2

DyKnow

This chapter introduces DyKnow [17] which is a stream-based knowledge processing middleware framework. It presents basic concepts and the architecture of DyKnow together with an overview of a possible implementation in ROS. Finally, the chapter shortly describes DyKnow Federations.

2.1 Overview

Systems today have large amounts of data at their disposal. The data is often available either through a variety of sensors or from the Internet. However, it is most often incomplete and noisy. On the other hand, functionalities such as execution monitoring and complex event detection often require clear and symbolic knowledge about the real world. Therefore, in order to use data from sensors in the aforementioned functionalities it should be processed. The processing of this data is usually divided into a number of distinct functionalities which are modeled as knowledge processes [17]. For example, an object recognition process might accept an image as input and provide a set of recognized objects as output. To provide output some knowledge processes require input information from multiple knowledge processes. However, information needed for processing is usually incrementally available and the actual processing can not be started until all the necessary information has been acquired. To address this issue the information flow between processing components can be modeled in terms of streams [17]. Therefore, in this case inputs and outputs of knowledge processes would be in form of streams.

Given that a number of knowledge processes might use an output from some process there must exist a mechanism for replicating streams. This can be done via a stream generator to which knowledge processes need to subscribe to. Subscription to stream generators also includes a policy which specifies the desired properties of a stream such as delay and order.

DyKnow is a stream-based middleware framework and provides support

for modeling of knowledge processing and implementation of stream-based knowledge processing applications. The applications can be represented as a network of knowledge processes connected via streams [17]. The following section gives an overview of components in DyKnow and describes how they map to concepts in stream-based knowledge processing.

2.2 Basic Concepts

Domains used in DyKnow describe two different types of entities: objects and features. Objects are building blocks of the world and can be both abstract and non-abstract while features represent object properties.

DyKnow implements two knowledge processes: sources and computational units. Knowledge processes offer fluent streams generators which produce fluent streams which comply with a certain fluent stream policy. The rest of this section describes these concepts in more details.

Fluent stream

A fluent stream consists of a stream (set) of samples. Samples are triples of the form $\langle t_a, t_v, v \rangle$ where t_a is available time, t_v valid time and v is a value, which can either represent an observation or an approximation of some feature. Available time represents the time when the sample is available for processing by the receiving process. Valid time, on the other hand, represents the time when the fact is true. It is obvious that the valid time and the available time are most often not the same because a certain amount of time is needed to do the processing of data. Therefore we can define the delay of a fact to be the processing time, i.e. $t_a - t_v$. An example of a fluent stream is the following set:

$$f = \{\langle 1, 1, v_1 \rangle, \langle 2, 1, v_2 \rangle, \langle 4, 2, v_3 \rangle, \langle 5, 6, v_4 \rangle\}$$

In this case, the fluent stream f consists of four samples where each sample is defined with an available time, a valid time and a value.

Source

A source represents a primitive process. Unlike other types of knowledge processes, primitive processes do not require input streams. The input data for these processes comes from the outside world, for example sensors or databases. Primitive processes adapt this data and provide output in the form of streams.

A function which represents a source maps time points to samples [17]. An example of a source is a process which adapts the data from sensors into streams or processes which read an input from a user.

Computational Unit

A computational unit is a type of refinement process. A refinement process represents a knowledge process which generates one or more new streams of samples from one or more input streams. In the case of a computational unit the output is only a single fluent stream. Computations in a computational unit are done each time a new sample from some input stream is available. However, given that input streams might not produce samples with the same available time, a computational unit uses the most recent samples in the input streams which do not produce new samples at the time of computations. An example of a computational unit is a process which estimates the speed of an object based on the position of that object at certain time points.

Fluent Stream Policies

In some scenarios receivers might impose constraints on the properties of streams. For example, the receiver could have a requirement that the maximum delay of each sample is at most 2 ms. In order to impose desired properties on the streams, DyKnow defines fluent stream policies. Fluent stream policies make it possible to define 5 different types of constraints:

- change constraint - defines how consecutive samples relate to one another. For example, it is possible to define that a sample needs to differ in either value or time stamp from the previous sample. Change constraint can also be used to restrict samples based on their valid times. An example is the situation where valid times of two consecutive samples need to differ by value t which represents the sample period.
- delay constraint - used for specifying maximum delay of a fluent stream (the difference between available and valid time).
- duration constraint - used to specify a constraint on the valid times of the samples in a stream, for example if a duration is defined to be between time-point 200 and time-point 300 then samples with valid times which are not in this interval will be filtered out.
- order constraint - used for specifying ordering of samples based on their available times. For example, it is possible to specify an ordering where each subsequent sample has a valid time larger or equal to the valid time of the sample before it.
- approximation constraint - defines how the system deals with the situation when some of samples in a stream are missing or they do not satisfy the policy. One way to deal with this problem is to produce a sample based on the approximation of available samples. DyKnow allows two types of approximation constraints: no approximation in

which case approximations are not allowed and most recent sample where approximations are allowed and are made using the most recent samples of available samples.

Fluent stream generator

Each knowledge process has a fluent stream generator which provides output in the form of streams. The streams generated by a fluent stream generator comply with the constraints defined by their fluent stream policies. This makes it possible to produce a number of different streams from the same input which are adopted to the needs of the receivers.

2.3 Architecture in ROS

2.3.1 ROS

ROS is a software framework for robotic software and as such it includes appropriate libraries and development tools. However, it also provides low-level device control and hardware abstractions which are the functionalities of an operating system. The main goal of ROS is to "support code reuse in robotics research and development" [9]. The framework itself is multilingual with the full support for C++, Lisp and Python. Each programming language has its associated client library which provides the tools and functions needed for developing ROS software.

The framework was made to be as thin as possible meaning that the ROS software is easy to integrate with other frameworks [9]. ROS developers are also encouraged to write libraries which reveal only the functional interfaces while hiding unnecessary complexities. These libraries should not depend on ROS thus making them reusable in other systems [24].

The software written for ROS is organized into packages which contain nodes, libraries or configurations. Packages can be organized into stacks providing certain "aggregate functionality" [9].

Nodes represent computational processes in the system and are written using the client libraries. The communications between nodes is done by passing messages on topics using the XML-RPC where topics represent a named bus. Therefore, in order for a node to communicate with other nodes it needs to publish or subscribe to a certain topic. Messages in this case are simple structures containing primitive types or nested structures. Arrays of primitive types or arrays of structures (messages) are also allowed. Topics support multiple subscribers and publishers, however each topic can be used for publishing messages of only one type.

ROS also provides support for request/reply communication using services. A service is defined as a combination of two messages, a request message and a reply message. The node which provides a certain service has an associated name used for discovery.

The architecture of a system written for ROS is in the form of a *Computation Graph*. The concepts in this graph are organized into a peer to peer network. Taking this into account the *Computation Graph* requires a discovery service. This is done through the ROS Master which provides registration and lookup services to the nodes. Whenever a node wants to publish something or provide a service it needs to advertise it with the Master. Similarly, the nodes use the Master to find information about other nodes and to properly set up the communication with them [9].

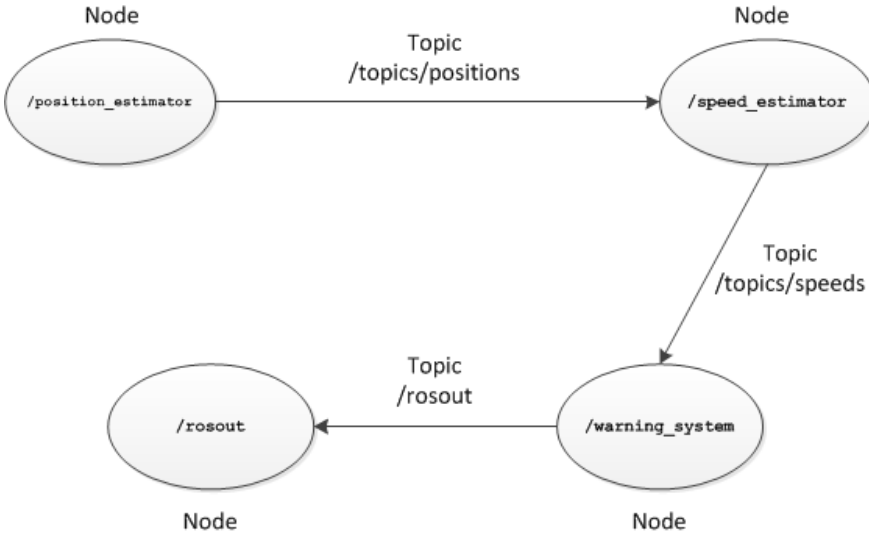


Figure 2.1: An example of a ROS computation graph.

Figure 2.1 gives an example of a ROS computation graph. It represents a process which takes a position and estimates a speed of an object. If an object's speed is above some threshold a warning system publishes a warning message on a standard output. In order to make speed estimations the `/speed_estimator` node has to communicate with the `/position_estimator` node to acquire the current position of some object. This communication is done over the `/topics/positions` topic. The `/position_estimator` publishes new position estimations on this topic while the `/speed_estimator` node needs to subscribe to it in order to acquire the latest estimations.

Communication between the `/speed_estimator` node and the `/warning_system` node is implemented in the similar manner through the `/topics/speeds` topic. Finally, the builtin `/rosout` topic is used by the `/warning_system` node to publish warning messages to the standard output.

2.3.2 ROS implementation

The architecture of the ROS implementation of DyKnow is shown in Figure 2.2.

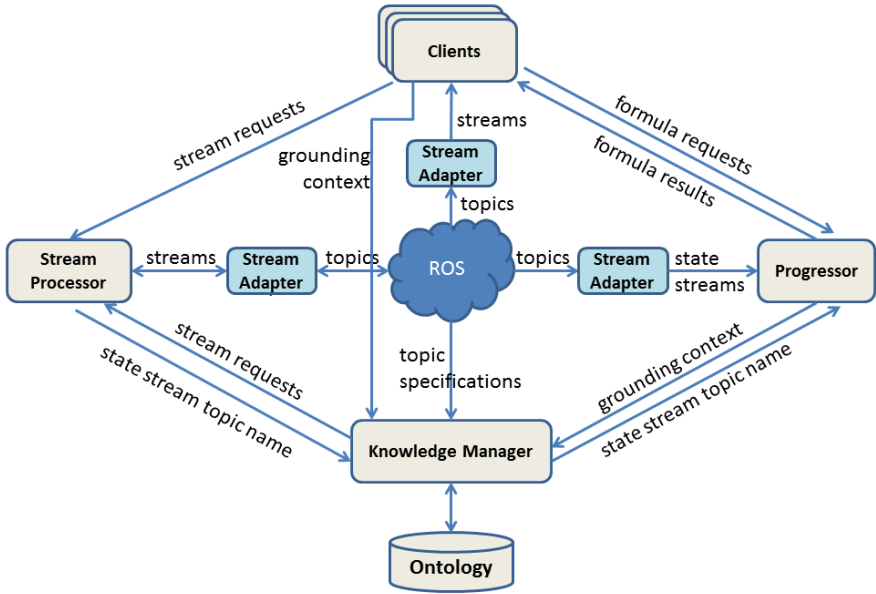


Figure 2.2: DyKnow architecture in ROS.

As the figure shows there are three main components in the system: Stream Processor, Knowledge Manager and Formula Progressor. The underlying ROS system keeps track of all available topics in the system. Topics in ROS map to fluent streams in DyKnow. ROS also provides all the necessary discovery services.

When it comes to actual applications which use DyKnow they usually require data from a number of fluent streams at precise points in time. However, fluent streams do not necessarily have the same valid times and therefore not all of them are available at the time points as the application needs them. To deal with this problem, the Stream Processor was introduced into the architecture. The main task of the Stream Processor is to merge and synchronize the required streams (topics) in the ROS system into a single state stream. The state stream is defined as a stream of state samples where state samples are samples which have state as a value, i.e. a tuple of values.

The Knowledge Manager in this architecture is a mediator between the Stream Processor and the Formula Progressor. The idea behind the Knowledge Manager is to provide a service which returns a state stream name (topic name) of a stream which contains the necessary data for formula progression. To achieve this the Knowledge Manager first extracts features from the formula. These features are then checked against the defined topics to find those which contain the relevant data. This information is then sent to the Stream Processor which generates the state stream.

If the state stream was successfully set up the Formula Progressor can use it to acquire the data needed for evaluating formula using the progression.

2.4 DyKnow Federations

Many robotic applications require cooperation of multiple agents in order to complete a certain mission or a task. Cooperation requires sharing and merging of information from distributed sources. One approach most commonly used in multi-agent applications today uses a central node responsible for merging and processing of information distributed among a number of agents. However, this approach introduces a high communication overhead and puts large requirements on the central node. DyKnow Federations use a decentralized model in which each node does much of the computations and processing locally.

In order to deal with the communication overhead, DyKnow Federations proposes a model where each platform has its own DyKnow instance.

In multi-agent environments agents have to delegate some tasks or plans to other agents in order to achieve cooperative tasks [17]. To deal with this issue DyKnow Federation framework uses the delegation framework from [11]. This framework requires that DyKnow instances on platforms are treated as services which interact with each other using the speech-act based interaction. Usually platforms have a number of agents with a set of services which together with an Interface Agent form an Agent level. Each platform also has a Platform specific level with the DyKnow instance. The communication between layers is done through the interface of the Gateway Agent while the communication between platforms (more specifically agents) is done through the Interface Agent. Figure 2.3 gives an overview of components in DyKnow Federation.

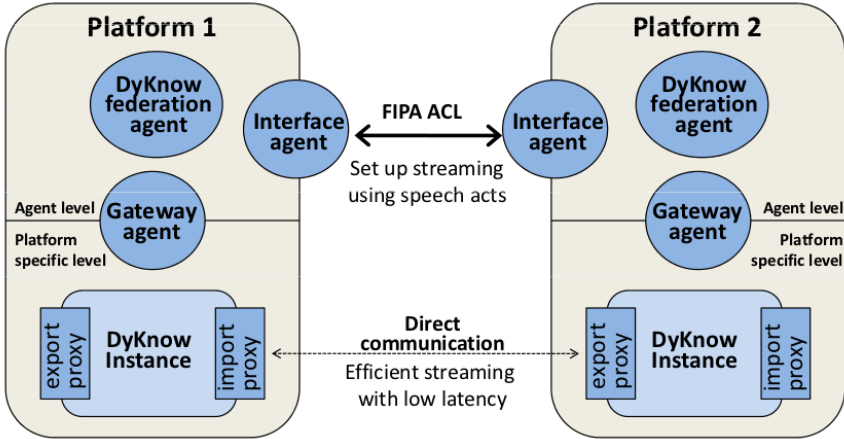


Figure 2.3: An overview of the components of a DyKnow federation [17].

The delegation framework deals with three different types of services:

- private - service available only to agents on the same platform
- public - available to all agents
- protected - service available to agents on the same platform or to agents on other platforms but in this case communication is done through the Interface Agent

In order to keep track of available services and allow for their discovery the delegated framework uses a Directory Facilitator (DF) database. Each platform has its own local instance of DF with the information about protected and private services local to that platform. Information about public services are kept in the global DF.

2.4.1 Components

In order for a platform to participate in a federation DyKnow Federation, the framework requires that it implements 3 components: DyKnow Federation Service, Export Proxy and Import Proxy.

The DyKnow Federation Service is the central component in the DyKnow Federation framework. It is used for both finding and sharing information among DyKnow instances. Each platform which implements this component is registered with the local DF. Therefore, the DF can be used for the discovery of other platforms participating in the federation. The communication between platforms is done indirectly through the Interface Agent. In other words, if a platform A wants to communicate with platform B, the request is sent to the Interface Agent on platform B which forwards the request to the DyKnow Federation Service on that platform. This indirection is required because the DyKnow Federation Service is implemented as a protected service. However, one issue with this kind of communication is that platforms should be aware of the labels of available services (stream generators) on the other platform. A proposed solution is to form a set of global semantic labels to which the local labels would be mapped to. Thus if platform A needs information about current altitude of platform B it can translate its local label into an agreed semantic label which is used in the request.

The Export Proxy deals with the subscriptions to stream generators on a platform. It implements the export method which is used to set up a subscription for a receiver. On the receiving end the Import Proxy is used for receiving streams and making them locally available.

2.5 Summary

Data provided by sensors is usually noisy and incomplete. On the other hand, autonomous systems require clear and symbolic knowledge in order

to implement certain functionalities. Therefore, in order to make use of sensor data it should be processed. A natural approach is to model the sensor data as streams. In this case the processing could be implemented as a set of knowledge processes where each knowledge process has some distinct functionality and has input and output in the form of streams.

DyKnow is a stream-based middleware framework and provides support for modeling of knowledge processing and implementation of stream-based knowledge processing applications. It defines two types of entities, objects and features where features represent building blocks of the world and features are object properties. DyKnow also provides support for sharing and merging of information from multiple distributed sources through DyKnow Federations.

This chapter has dealt with the ROS implementation of DyKnow. ROS is a software framework for robotic software. It is based on a publish/subscribe architecture meaning that computational units (nodes) are communicating by publishing messages or subscribing to a named bus (topic).

The ROS implementation of DyKnow consists of three main components: Stream Processor, Knowledge Manager and Formula Progressor. The main task of the Formula Progressor is evaluation of logical formulas through progression. In order to do this the Formula Progressor needs to subscribe to a stream which contains relevant data for each symbol in a formula. Finding relevant topic specifications is the task of the Knowledge Manager which bases this decision on the meaning of content of streams. The relevant topic specifications are passed to the Stream Processor which sets up a state stream to which the Formula Progressor has to subscribe to evaluate the formula.

Chapter 3

Semantic Technologies

This chapter gives an overview of relevant semantic technologies for this thesis. The focus is on the semantic technologies used on the Semantic web, more specifically the Resource Description Framework (RDF) and the Web Ontology Language (OWL).

3.1 Semantic Web

The World Wide Web (WWW) offers incredible amounts of easily accessible data. The data is organized into documents which are interconnected with hyperlinks and thus can easily be browsed. The simplicity of the WWW can be considered as the main reason for its fast development and success [20]. Web pages have some structure which is mostly in the form of meta-data used by web-browsers to display them correctly. However, the body of a web page is usually without explicit structure. This lack of structure of the documents and the data makes it difficult for automated agents to interpret the meaning of the information.

In some cases query answering on the WWW requires the combination of data from different sources. Horrocks [20] gives an example of a query which should return all heads of states of all EU countries. To answer this query two lists are required, a list of EU countries and a list of heads of states. If we assume that these lists have different sources then with the current design of the WWW queries of this kind could not be answered.

The Semantic Web¹ is a World Wide Web Consortium (W3C) extension proposal which aims to make the WWW more accessible to machines and thus allow automatic processing of data. To achieve this the Semantic Web introduces annotations to documents on the web which capture the semantics of the content [1]. The W3C provides a set of recommendations for the technologies to be used on the Semantic Web. The semantic annotations

¹<http://www.semanticweb.org>

are done using a combination of the Extensible Markup Language (XML) and RDF. XML in this case provides a syntax and an exchange mechanisms while RDF provides components needed for describing resources and relations between them. However, RDF lacks the expressiveness needed for the modeling of problem domains. Domains are modeled using ontologies which allow for the formal description of a conceptualization [1]. The following sections give more details about the technologies used on the Semantic Web, more specifically RDF and the Web Ontology Language (OWL).

As stated before, the WWW lacks the support for queries which require combinations of data from different distributed sources. The use of ontologies in the Semantic Web makes it possible to define mappings between concepts in distributed sources and in this way enable the aforementioned queries. However, automated mapping is still limited and is an active research area [14]. Current automated mapping strategies are mostly based either on structure of ontologies or linguistic properties of concepts in the ontologies. However, these strategies can automatically map only a part of semantically related concepts. [7].

From the perspective of artificial intelligence the development of the Semantic Web represent an interesting aspect. Both the Semantic Web and artificial intelligence aim at making machines capable of intelligent behavior. Artificial intelligence aims at the human-level intelligence which the Semantic Web can not provide but as Halpin [14] argues the artificial intelligence could benefit from the development of a usable ontology of the real world.

However, the Semantic Web has a number of challenges to overcome. An obvious problem is that currently the Semantic Web is not widespread and therefore the process of upgrading the WWW to conform to the Semantic Web poses a considerable challenge given the size of the current WWW [3]. To achieve integration of data from different sources the Semantic Web should be able to cope with sources that are heterogeneous in some sense, i.e. language, design of ontologies, etc [3].

3.2 RDF/RDFS

RDF provides the means for describing resources on the WWW in the form of declarative statements. Resources in this case are Web documents and RDF is used to describe information such as title, author, creation date, etc. [22]. Statements are usually written in XML, but other notations are also possible. RDF is based on the notion of a Uniform Resource Identifier (URI). This makes it possible to directly reference non-local resources on the Internet [20]. URIs are usually organized into namespaces. To shorten the syntax, the XML-based RDF syntax makes use of qualified names for the URIs of the RDF resources in which case a namespace is assigned a prefix which together with the local name forms a qualified name of the resource [22].

Statements in RDF are triples consisting of a subject, a predicate and

an object. Each statement describes a subject with a value (object) for a certain property (predicate) and can be represented as two nodes (subject and object) connected by an edge (predicate). A set of statements then forms a graph [20]. The following example gives the value *altitude1* for the property altitude of the UAV instance *uav1*. Each component of this triple is represented as a URI.

```
<<http://www.example.org/uavs/uav1>,
<http://www.example.org/features/altitude>,
<http://www.example.org/altitude/altitude1>>
```

By using qualified names for URIs the previous example could be represented in the following manner:

```
<xmlns:uavs="http://www.example.org/uavs/#">
<xmlns:features="http://www.example.org/features/#">
<xmlns:altitudes="http://www.example.org/altitude/#">

<uavs:uav1 , properties:altitude , altitudes:altitude1>
```

In this case namespaces of the subject, the predicate and the object were specified and assigned a prefix (**uavs**, **feature**, **altitudes**) which together with a local name (**uav1**, **altitude** and **altitude1**) form a qualified name of the resource.

However, RDF is domain independent and does not provide adequate support for modeling domains [1]. Therefore, an extension called the Resource Description Framework Schema (RDFS) was proposed which includes the expressive power needed for defining ontologies. The RDFS allows engineers to describe classes and properties and to define hierarchies of classes and hierarchies of properties. These notions are very similar to Object Oriented Programming.

3.2.1 Syntax

Each XML-based RDF document begins with an XML declaration together with a declaration of the namespaces used in the document. The RDF and RDFS specific tags are also organized into namespaces which refer to the defining RDF documents.

RDF resources are defined using the **rdf:Description** element. This element has an attribute **rdf:about** which holds a reference to the resource of the subject [22]. The property of the subject is represented as the content of the **rdf:Description** element [1]. RDF allows multiple declarations of properties in one **rdf:Description** element. Properties are referenced in the same way as subjects, with qualified names. The value of a property can either be a plain literal or another resource. Literals are usually treated as strings, however if an application which uses RDF resources needs explicit types, it is possible to assign datatypes by pairing URI reference of the datatype with the literal [22]. If the value of a property is another resource

then it is possible to either declare a new resource description nested under the property or to make a reference to a defined resource. The referencing is done with the use of the `rdf:Resource` attribute.

The following example defines the resource *platform1* and its three properties (type, color and altitude). The color is declared as a nested resource while the property altitude refers to an already defined resource.

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:properties="http://www.example.org/properties/#"
  xmlns:color="http://www.example.org/colors/#">

  <rdf:Description rdf:about="platform1">
    <properties:type>flying</properties:type>
    <properties:color>
      <rdf:Description rdf:about="color1">
        <color:code rdf:datatype="http://www.w3.org/2001/XMLSchema#Integer">123</color:code>
      </rdf:Description>
    </properties:color>
    <properties:altitude rdf:resource="alt1"/>
  </rdf:Description>
</rdf:RDF>
```

As stated before, RDF is not suitable for describing domains as a domain specification usually includes information which captures the relations between the classes in the domain and RDF allows only specifications of simple statements about instances of classes. RDFS is a W3C recommendation which gives the support for defining classes and properties together with their hierarchies. To achieve this RDFS introduces a number of specific resources and properties [22]. Classes are defined as regular RDF resources but the property is set to `rdf:type` and the property value is set to a RDFS resource `rdfs:Class`. Properties are defined in an analogous manner with the RDFS resource `rdfs:Property`. The property `rdf:type` is also used to declare that a RDF resource is an instance of certain class.

Class and property hierarchies are defined using the properties `rdfs:subPropertyOf` and `rdfs:subClassOf`. It is possible for a class or a property to have any number of super and sub concepts.

RDFS also introduces a possibility of defining restrictions on the properties. In RDFS it is possible to define the domain and range of a certain property. A domain defines classes which can have a certain property while the range defines which types (classes) can be used for the value of the property.

The following code gives an example of a simple class hierarchy. The class `Object` is the top class and `MovingObject` is its child. Class `FlyingObject` is a subclass of `MovingObject`. Finally, class `UAV` inherits from both `FlyingObject` and `MovingObject`.


```

<?xml version="1.0" ?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xml:base="http://example.org/schemas/vehicles">
  <rdf:Description rdf:about="Object">
    <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-
      schema#Class" />
  </rdf:Description>

  <rdf:Description rdf:ID="MovingObject">
    <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-
      schema#Class" />
    <rdfs:subClassOf rdf:resource="#Object" />
  </rdf:Description>

  <rdf:Description rdf:ID="FlyingObject">
    <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-
      schema#Class" />
    <rdfs:subClassOf rdf:resource="#MovingObject" />
  </rdf:Description>

  <rdf:Description rdf:ID="UAV">
    <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-
      schema#Class" />
    <rdfs:subClassOf rdf:resource="#MovingObject" />
    <rdfs:subClassOf rdf:resource="#FlyingObject" />
  </rdf:Description>
</rdf:RDF>

```

RDF provides support for defining groups of items, for example it is possible to define that a certain property has a group of resources or literals as a value. Three different container types are possible:

- **rdf:Bag** - defines a group that may have duplicates and where the ordering is not important
- **rdf:Seq** - defines a group that may have duplicates and where the ordering is important
- **rdf:Alt** - defines a group of alternatives.

3.3 Ontologies

An ontology represents a formal model of a domain [28]. It includes class hierarchies, properties and relationships between the concepts. As stated before, RDFS provides a way of describing ontologies but it lacks the support for describing more complicated concepts usually needed for describing more

complex domains. Antoniou and Harmelen [1] list a number of limitations of RDFS:

- it does not support cardinalities
- inability of representing disjoint classes
- no support for boolean expressions, conjunction, disjunction, negation
- only supports a limited number of restrictions on properties

3.3.1 OWL

The Web Ontology Language (OWL) is a W3C ontology language recommendation. OWL uses a RDF/XML syntax but other syntaxes also exist which provide higher readability for humans. This section uses the XML/RDF syntax.

Reasoning about an ontology is used to extract or rather make explicit knowledge which is implicit in the ontology. 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 to determine if some concept (class) is unsatisfiable. A class is unsatisfiable if the interpretation of that class is an empty set in all models of the ontology.

Three different versions of OWL currently exists. These sub-languages differ in the expressiveness they have and thus give the engineer a possibility to choose the one which satisfies his application's ontology requirements [29].

- OWL Full - Has maximal expressiveness with complete compatibility with RDF. However, it does not guarantee completeness nor decidability for reasoning.
- OWL DL - Supports only the decidable subset of OWL Full expressiveness. It is based on description logics and guarantees completeness and decidability [19]. The disadvantage is that it does not have full compatibility with RDF.
- OWL Lite - Provides the basic features which include class and property hierarchies and support for simple cardinalities (0 or 1). Some restrictions on properties are possible and reasoning is both complete and decidable. With these restrictions OWL Lite is easier to use than OWL DL and is suitable for inexperienced users. It also has a lower complexity than OWL DL [29].

When it comes to reasoning in OWL DL reasoners guarantee completeness and decidability. The main reason for this is the fact that OWL DL is based on Description Logics which itself is based on a decidable fragment of First Order Logic [29]. OWL Full on the other hand does not restrict type

separation meaning that it is possible to define a class which is an individual at the same time and therefore can not guarantee decidability [29].

Each OWL document has a header which defines the namespaces used in the ontology, together with imports and assertions about the ontology (version, comments etc.). Imports, defined with `owl:import`, allow users to reuse parts of or whole ontologies. The rest of the OWL document represents the body and includes entity declarations. Similar to RDFS, in OWL it is possible to define classes, properties and instances.

Classes are defined using the `owl:Class` element. OWL defines two special classes, `owl:Thing` and `owl:Nothing`. Each defined class is a subclass of `owl:Thing`. `owl:Nothing` defines an empty class and thus each defined class is a super class of `owl:Nothing`. Therefore if some class is equivalent to `owl:Nothing` that means that the class is unsatisfiable. Class hierarchies are defined in the same way as in RDFS using the `rdfs:subClassOf` element. However, OWL also adds support for defining disjoint classes. This is done using the `owl:disjointWith` element.

OWL defines two types of properties: datatype property and object property. A datatype property has a data value as the value. An object property, on the other hand, has a class instance as the value and is used to define relations between two instances. Hierarchy, range and domain of properties are defined using the same methods and syntax as in RDFS. OWL also implements a number of special types of properties. These include:

- transitive property - $P(x, y) \wedge P(y, z) \Rightarrow P(x, z)$
- symmetric property - $P(x, y) \Leftrightarrow P(y, x)$
- functional property - $P(x, y) \wedge P(x, z) \Rightarrow y = z$
- inverse functional property - $P(y, x) \wedge P(z, x) \Rightarrow y = z$

As stated before OWL includes additional constructs which give the engineer more expressive power. These constructs allow the engineer to describe classes as boolean expressions of other classes and properties. OWL supports union, intersection and complement which correspond to `owl:unionOf`, `owl:intersectionOf` and `owl:complementOf` in OWL syntax. In order to specify additional restrictions on the values of properties OWL supports universal and existential quantifiers together with the cardinality mechanisms. Universal and existential quantifiers are declared using the `owl:allValuesFrom` and `owl:someValuesFrom` constructs. When it comes to cardinality, three different cardinality restrictions can be specified in OWL: minimal, maximal and exact. These restrictions are specified through the `owl:minCardinality`, `owl:maxCardinality` and `owl:cardinality` constructs respectively.

As an example ontology we are going to use our example scenarios presented in section 1.2. In the Single Platform Scenario we are dealing with a single platform which is doing execution monitoring. In order to do execution monitoring the platform needs to have input from its environment. For

example, assume that one of the monitoring formulas says that the platform should never be closer than 3 meters to any building then in that case the sensors on the platform need to provide data about the current position of the platform and the position of the buildings in the area. However, to evaluate formulas of this kind the platform also requires the list of all buildings in the environment in order to check that neither of them is closer than 3 meters. One way to deal with this issue is to model the environment. The domain model in Single Platform Scenario would include all the entities in the environment together with their relations. As discussed earlier, ontologies support representing formal models of a domain.

In DyKnow we differentiate between two types of entities, objects which represent the building blocks of the world (cars, houses, etc.) and features which represent properties of the world and its objects (altitude, position, etc.). Therefore our ontology for the scenarios would have to reflect this. To achieve this we propose two different hierarchies, one for objects in the domain and one for features of the domain. Figures 3.1 shows one of the ontologies that could be used in the Single Platform Scenario.



Figure 3.1: Visualization of the ontology for the single platform scenario.

As the object hierarchy shows, the domain deals with two types of objects, static and moving objects. Static objects in this case represent some points of interest while moving objects enumerate different types of vehicles in the domain. The actual objects or instances of the classes are not shown in the visualization but also need to be included in the ontology. The ontology includes 5 objects: **uav1** and **uav2** of type *UAV* and **car1**, **car2** and **car3** of type *Car*. The full ontology specification can be found in Appendix B.1.

Features describe relations in the domain and are represented as relations in the ontology. These relations in our ontology are described as an intersection class. The intersection includes the class which defines the arity of the feature (*UnaryRelation*, *BinaryRelation*, *TernaryRelation*) and 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, feature *Altitude* which represents an altitude of some *UAV* is defined as follows:

$$Altitude \subseteq UnaryRelation \cap \forall arg1.UAV$$

meaning that *Altitude* is a unary relation and the first argument must be of type *UAV*.

Another example is feature *Behind*:

$$Behind \subseteq BinaryRelation \cap \forall arg1.Object \cap \forall arg2.Object$$

This feature describes a relation which takes two arguments and is used to test if some entity (argument 1) is behind another entity (argument 2). In this case both arguments have to be of type *Object*.

In the case of the Multiple Platform Scenario we are dealing with three platforms. The ontology for the first platform is the same as the one proposed earlier for the Single Platform Scenario. Unlike the first platform, the second and the third platform capture only the part of the environment presented in the ontology for the first platform. There are 2 objects defined in the ontology, **car11** and **car12** of type *Automobile* which correspond respectively to objects **car1** and **car2** in the first ontology. The ontology for the second platform is given in Figure 3.2 and deals only with cars in the environment. The ontology also defines two features, *Speed* and *Position*. The full ontology definition is given in Appendix B.2.

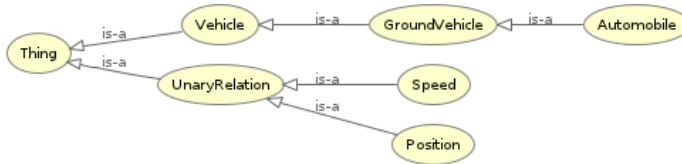


Figure 3.2: Visualization of the ontology of platform 2.

Similarly, the third platform deals only with flying vehicles or aircrafts and defines 5 objects, **uas20**, **uas21** and **uas22** of type *UnmannedAircraftSystem* and **heli1** and **heli2** of type *MannedAircraftSystem*. The ontology defines 4 relations, unary relations *Alt*, *Height* and *Spd* and a binary relation *Near*. Relations *Alt* and *Height* are defined to be equivalent.

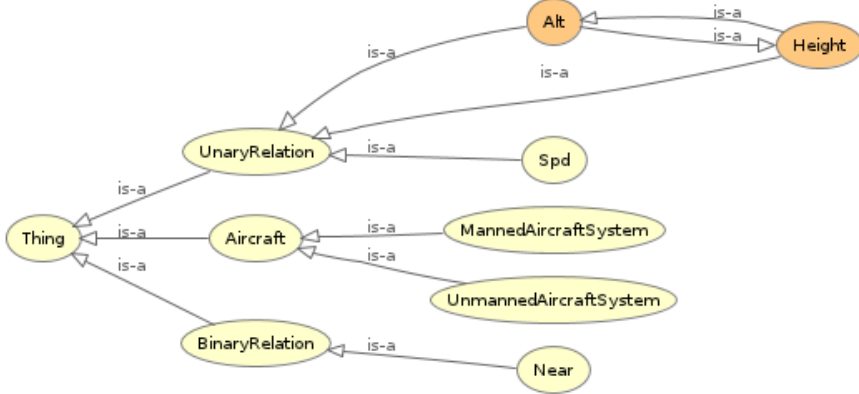


Figure 3.3: Visualization of the ontology of platform 3.

Reasoning in these scenarios is required to infer implicit relations in the ontologies. As a simple example we can take the feature *Behind* from the first ontology which accepts arguments of type *Object*. Therefore objects **uav1** and **uav2** of type *UAV* could not be used as arguments to this feature even though all objects defined in the ontology are essentially of type *Object*. With the reasoner support these relations would be included in the ontology and **uav1** and **uav2** could be used as the arguments to feature *Behind*. The full specification of the ontology is provided in Appendix B.3.

3.3.2 Semantic mappings

In order to reuse knowledge from other ontologies we need to specify the relations between concepts in different ontologies together with a reasoning mechanism which can reason over multiple ontologies [26]. The relations are called semantic mappings and implement relations such as subclass, superclass and equivalence.

Much work has been done related to representation of semantic mappings between ontologies such as [6], [13], [21]. Work done by Serafini and Taminin [27] differs from the aforementioned works as it also presents a reasoning mechanism for reasoning over multiple ontologies connected by semantic mappings. The semantic mappings specifications in their work are based on Context OWL (COWL) presented in [6]. In order to support reasoning over multiple distributed ontologies Serafini and Taminin reuse the idea of Distributed Description Logics (DDL) presented in [4] which provides

the support for formalizing collection of ontologies connected by semantic mappings. The reasoning with DDL is based on a tableau reasoning technique for local description logics which was extended to support multiple ontologies.

In this report we are going to use the method for representing semantic mappings presented in [6] as it provides the support for explicit semantic mappings between classes and individuals in ontologies together with a XML representation which can easily be queried. In this representation mappings between ontologies is represented as a set of bridge rules (mappings). Each bridge rule requires the specification of a source and a target ontology entity together with the type of the bridge rule. An entity can be a property, a concept or an individual in which case it is only possible to specify that individuals are the same. Supported bridge rule types are:

- $c1 \equiv c2$ – $c1$ is equivalent to $c2$
- $c1 \sqsubseteq c2$ – $c1$ is more specific than $c2$
- $c1 \sqsupseteq c2$ – $c1$ is more general than $c2$
- $c1 \perp c2$ – $c1$ is disjoint with $c2$
- $c1 \star c2$ – $c1$ is compatible with $c2$ meaning that $c1$ might relate to $c2$
- $i1 \equiv i2$ – individual $i1$ is the same as individual $i2$

Bouquet et al. [6] also suggested an XML schema for representing bridge rules. An example of a bridge rule represented in XML is given in 3.1.

Listing 3.1: Bridge rule in XML.

```
<owl:bridgeRule owl:br-type="equiv">
  <owl:sourceConcept rdf:resource="http://www.example.com/ontology#
    Position"/>
  <owl:targetConcept rdf:resource="http://www.semanticweb.org/
    ontologies/2011/5/Distributed.owl#Position"/>
</owl:bridgeRule>
```

As the listing shows, the source concept and the target concept need to be represented with full URIs. The attribute **br-type** holds the type of the bridge rule. In the XML schema the following names are used for bridge rule types:

- equiv – \equiv
- into – \sqsubseteq
- onto – \sqsupseteq
- incompat – \perp
- compat – \star
- same – individuals are the same

If we go back to the ontologies in the Multiple Platform scenario presented in the previous section we see that even though they refer to the same objects in the real world there is no way for a machine to infer that these objects are the same. Therefore to support reuse of information from multiple platforms it is required to specify the bridge rules between concepts and individuals in the ontologies. The bridge rules for the ontologies presented in the previous section are given in listing 3.2.

Listing 3.2: Bridge rules in XML.

```
<owl:mapping>
  <owl:bridgeRule owl:br-type="equiv">
    <owl:sourceConcept rdf:resource="http://www.example.com/ontology#
      GroundVehicle"/>
    <owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
      /2011/5/Distributed.owl#GroundVehicle"/>
  </owl:bridgeRule>

  <owl:bridgeRule owl:br-type="equiv">
    <owl:sourceConcept rdf:resource="http://www.example.com/ontology#Car"/>
    <owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
      /2011/5/Distributed.owl#Automobile"/>
  </owl:bridgeRule>

  <owl:bridgeRule owl:br-type="equiv">
    <owl:sourceConcept rdf:resource="http://www.example.com/ontology#Position"/>
    <owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
      /2011/5/Distributed.owl#Position"/>
  </owl:bridgeRule>

  <owl:bridgeRule owl:br-type="equiv">
    <owl:sourceConcept rdf:resource="http://www.example.com/ontology#Speed"/>
    <owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
      /2011/5/Distributed.owl#Speed"/>
  </owl:bridgeRule>

  <owl:bridgeRule owl:br-type="equiv">
    <owl:sourceConcept rdf:resource="http://www.example.com/ontology#UAV"/>
    <owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
      /2011/6/distributed_uavs.owl#UnmannedAircraftSystem"/>
  </owl:bridgeRule>

  <owl:bridgeRule owl:br-type="equiv">
    <owl:sourceConcept rdf:resource="http://www.example.com/ontology#Close"/>
    <owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
      /2011/6/distributed_uavs.owl#Near"/>
  </owl:bridgeRule>

  <owl:bridgeRule owl:br-type="equiv">
    <owl:sourceConcept rdf:resource="http://www.example.com/ontology#Altitude"/>
    <owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
      /2011/6/distributed_uavs.owl#Height"/>
  </owl:bridgeRule>

  <owl:bridgeRule owl:br-type="equiv">
    <owl:sourceConcept rdf:resource="http://www.example.com/ontology#Speed"/>
    <owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
      /2011/6/distributed_uavs.owl#Spd"/>
  </owl:bridgeRule>

  <owl:bridgeRule owl:br-type="same">
    <owl:sourceConcept rdf:resource="http://www.co-ode.org/ontologies/ont.owl#
      uav1"/>
    <owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
      /2011/6/distributed_uavs.owl#uas20"/>
  </owl:bridgeRule>

  <owl:bridgeRule owl:br-type="same">
    <owl:sourceConcept rdf:resource="http://www.co-ode.org/ontologies/ont.owl#
      uav2"/>
    <owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
      /2011/6/distributed_uavs.owl#uas21"/>
  </owl:bridgeRule>

  <owl:bridgeRule owl:br-type="same">
    <owl:sourceConcept rdf:resource="http://www.co-ode.org/ontologies/ont.owl#
      car1"/>
    <owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
      /2011/5/Distributed.owl#car11"/>
  </owl:bridgeRule>

  <owl:bridgeRule owl:br-type="same">
    <owl:sourceConcept rdf:resource="http://www.co-ode.org/ontologies/ont.owl#
```



```

        car2" />
    <cowl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
        /2011/5/Distributed.owl#car12" />
    </cowl:bridgeRule>
</cowl:mapping>

```

In this case only two types of bridge rules are used, equivalence between classes and `owl:sameAs` relation between individuals.

3.4 Summary

This chapter has presented the idea of the Semantic Web. The Semantic Web represents a World Wide Web Consortium (W3C) extension proposal which aims at making current WWW more machine accessible by encoding semantics of data on the Web. This is achieved by semantically annotating data on the Web through a number of semantic technologies. One such technology is the Resource Description Framework (RDF) which provides the support for describing resources on the WWW in the form of declarative statements. However, RDF lacks the support for representing structure. The Resource Description Framework Schema (RDFS) is an extension of the RDF which provides the mechanism for domain modeling and allows definition of classes and their hierarchies. RDFS has a number of limitations, more specifically it is not possible to define boolean expression nor cardinality between classes in a model. These expressions are supported in the Web Ontology Language (OWL) which provides the support for representing ontologies where ontologies represent a formal model of a domain [28]. Ontologies contain three types of entities: individuals (instances), concepts (classes) and properties. With OWL it is possible to define boolean expressions in an ontology such as conjunction, disjunction and equivalence as well as universal and existential quantifiers. The OWL also allows specifications of cardinality constraints, for example minimum and maximum cardinality.

In order to integrate data from multiple ontologies it is necessary to establish relations between classes and instances in different ontologies. These relations are called semantic mappings and implement relations such as subclass, superclass and equivalence. The chapter has presented a COWL representation of semantic mappings in which case the mappings between classes and instances are represented as bridge rules.

Chapter 4

Analysis

Reasoning over incrementally available information in autonomous systems is needed to support a number of functionalities such as execution monitoring and planning. The data needed for reasoning is provided by physical sensors and requires processing [16]. As shown earlier, DyKnow can be used to bridge the gap between the data available from sensors and the information needed for reasoning. The output from DyKnow is in the form of streams.

One technique for incremental reasoning with streams supported by DyKnow is "progression of metric temporal logic to incrementally evaluate logical formulas" [15]. In order to evaluate formulas it is necessary to map variables to streams of data based on the content of data streams. This can be done manually. However, this makes the system fragile as any changes in the streams or additions of new streams would mean that the mappings have to be checked and potentially changed. The automation of this process would make the system more versatile and easier to use.

The problem of matching variables in logical formulas and content of streams can be divided into two subproblems:

- representing knowledge about the semantics of content of streams
- using this knowledge to match the intended meaning of variables to appropriate streams

The following sections describe these problems in more detail and provide possible solutions.

4.1 Semantic stream representation

Our goal is to automate the process of matching the intended meaning of variables to content of streams in a system. Therefore the representation of semantics of streams needs to be machine readable. This makes it possible

for the system to reuse this knowledge in reasoning about which stream content corresponds to which variable in a logical formula. The knowledge about the meaning of content of streams needs to be specified by a user. By assigning meaning to stream content the streams do not have to use predetermined names, hard-coded in the system. This would also make the system domain independent meaning that it could be used to solve different problems in a variety of domains.

An important step in semantic stream representations is to establish the vocabulary for the representation. In other words it is necessary to establish which entities exist in the environment. One way to do this is it to model the domain and the relationships between the entities in the domain. To make this usable in the process of semantic matching the domain model should be interpretable by machines.

Ontologies provide suitable support for modeling machine readable domain models. As discussed, ontologies also provide reasoning support and support for semantic mapping which is necessary for the reuse of data on distributed platforms. The ontologies used to model the domain in DyKnow differentiate between two types of entities, objects and features.

After establishing a vocabulary of the domain the next step is to find a way of representing streams and encoding knowledge about the content of the streams. Each representation of a stream should include all the information necessary to subscribe to the stream. In our solution we want to make a ROS implementation of DyKnow in which case streams correspond to topics in ROS. Each topic has an associated topic name and a message type which are used for establishing subscriptions. To access the output of a stream it is also necessary to know which field contains which data. All this information together with the encoding of the content of the stream should be included in the stream representation.

The stream representation should support specification of the following topic categories:

- topics containing sorts
- topics containing features
- topics containing objects

The first category represents topics which are used to enumerate objects of a certain sort. These topics can be used when there is a need for the list of all objects of some sort. For example, if we have a formula which is quantified over some variable then in order to evaluate it all possible values of this variable are needed.

The second category represents topics which contain fluents for features. An example of a topic containing data for the features Altitude and Speed for every object of sort UAV is given in Listing 4.1.

```

Topic name: topic1
Message type: UAVMsg
Fields:
int id
float alt
float spd

```

Listing 4.1: A topic specifying the features Altitude and Speed for the sort UAV.

Finally, the third category represents topics which contain a single object. These topics are used to add a level of indirection. For example, a topic can contain some feature for a currently tracked object while the id of a currently tracked object can be a value of another topic.

In order to encode the content of the aforementioned topics we propose the Semantic Specification Language for Topics (SSL_T). The SSL_T provides the means for representing content of topics in a system. Each topic representation defined in the SSL_T includes all the necessary information needed for subscribing to the topic such as topic name, message type, fields containing feature data and id fields. The formal grammar for SSL_T is presented in Listing 4.2.

Listing 4.2: Formal grammar for SSL_T .

```

prog : expression+ | EOF ;

expression :      'topic' topic_name 'contains'
                  ( feature_list | sort | object ) ;

topic_name :      NAME ':' NAME ;

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;

```

As mentioned earlier, the first category of topics is used to enumerate objects of some sort. In SSL_T it is possible to define both topics containing all objects of a sort and topics containing only a subset of objects. This can be specified in SSL_T using the **some** and **every** keywords.

Listing 4.3: Topic specifications in SSL_T .

```

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

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

```

Listing 4.3 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 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 data for the feature for all possible combinations of the specified argument. Listing 4.4 shows an example of five different topic specifications. The first topic, **topic1**, contains feature Altitude for object uav1. The data for the altitude can be found 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** object argument uav1 is implicit and therefore no additional data needs to be specified. On the other hand, in topic **topic2** the object argument is not implicit and needs to be computed from a field in a topic. This field is specified after the **for** keyword.

Topic **topic3** defines the same feature but for multiple UAVs. In this case the feature is only defined for a subset of UAVs. Topics **topic4**, **topic5** and **topic6** specify topics which contain feature Behind which is of 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 **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.

SSL_T allows the definition of multiple features in one topic specification.

Listing 4.4: Topic specifications in SSL_T.

```
topic topic1:UAVMsg contains Altitude(uav1)=msg.alt

topic topic2:UAVMsg contains Speed(uav1)=msg.spd for uav1=
    msg.id

topic topic3:UAVMsg contains Altitude(UAV)=msg.alt for some
    UAV = msg.id

topic topic4:UAVMsg contains Behind(uav1, uav3)=msg.behind
    for uav1 = msg.id1, uav3 = msg.id2

topic topic5:UAVMsg contains Behind(uav2, uav1)=msg.behind
    for uav2 = msg.id1, uav1 = msg.id2

topic topic6:UAVMsg contains Behind(UAV, uav2)=msg.behind
    for every UAV = msg.id1, uav2 = msg.id2
```

The syntax might introduce some ambiguities, for example let us consider the next example:

```
topic topic1:UavMsg contains Behind(uav1, uav1) for uav1 =
    msg.id1, uav1 = msg.id2
```

In this case the topic defines the feature Behind with two arguments which have the same identifier, uav1. However, the arguments are in different parts of the message, argument 1 is computed from the field id1 while the second argument is computed from the field id2. Given that both arguments have the same identifier the only way to determine which is the first and which is the second argument in the **for** part is to base this decision on the ordering. However, in cases when a topic defines multiple features with the same argument the topic specifications would be unreadable. To deal with this ambiguity the SSL_T introduces aliases. The aliases are specified in the feature argument list and later used after the **for** keyword. Therefore, the unambiguous topic specification for the example would be defined as follows:

```
topic topic1:UavMsg contains
    Behind(uav1 as arg1, uav1 as arg2) for arg1 = msg.
        id1, arg2 = msg.id2
```

4.2 Matching symbols to topics

So far we have discussed how to specify the meaning of a very general class of typed streams which have tuples as values. To achieve automatic semantic matching of intended meaning of symbols to content of streams an agent should be able to automatically determine which topics are relevant for which symbol based on their content. In the case of metric temporal formulas, features correspond either to terms (non-boolean value) or predicates (boolean values) in the formula while the arguments correspond to objects in the environment. Therefore the matching problems consists of two subproblems, finding and extracting features from the logical formula and finding relevant topic specifications for these features. The process of semantic matching is shown in Figure 4.1.

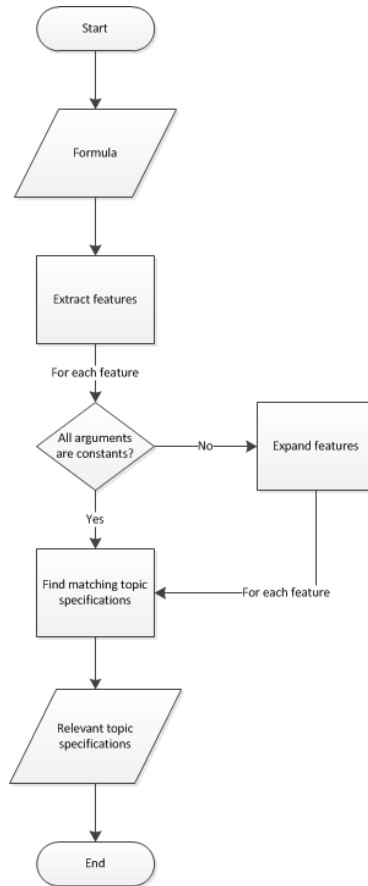


Figure 4.1: Process of semantic matching.

Feature arguments in a logical formula can either be constants or variables. Arguments which are constants are considered as objects in the en-

vironment. Variables on the other hand are quantified variables and can be replaced with any value from the domain of a variable. The domain of a variable represents all possible values a variable can have and in our case this maps to a sort in the domain model. Therefore, arguments which are variables can be replaced with a sort from the domain model. As an example let us consider the following formula

$$\text{forall } x \text{ in UAV Behind}[x, \text{uav1}] \text{ and Altitude}[\text{uav1}] > 10$$

Both *Behind*[*x*, *uav1*] and *Altitude*[*uav1*] are terms and therefore represent features in the domain model. *uav1* is a constant and therefore has to refer to an object in the environment. However, *x* is a quantified variable and therefore all instances of sort UAV in the domain model need to be considered. Taking this into account features *Behind*[UAV, *uav1*] and *Altitude*[*uav1*] would be extracted from the formula.

After completing the feature extraction from a logical formula the next step in semantic matching is to find relevant topics for these features. As we mentioned before the semantics of each topic is encoded in the topic specification. However, the proposed topic specification language is not suitable for querying. Therefore, we propose an XML structure which would include all the information from the SSL_T topic specification. Listing 4.5 presents the Document Type Definition (DTD) for the SSL_T XML syntax. XML is a suitable candidate because it is machine readable and provides good support for querying.

Listing 4.5: DTD for the SSL_T XML syntax.

```
<!DOCTYPE Topic_specs [
  <ELEMENT Topic_specs (Topic)*>
  <ELEMENT Topic (SSL, (feature+ | sort | object)) >
  <!ATTLIST Topic msgtype NMTOKEN #REQUIRED >
  <!ATTLIST Topic name ID #REQUIRED >

  <ELEMENT SSL ( #PCDATA ) >

  <ELEMENT feature ( object | sort )+ >
  <!ATTLIST feature name NMTOKEN #REQUIRED >
  <!ATTLIST feature value NMTOKEN #REQUIRED >

  <ELEMENT object EMPTY >
  <!ATTLIST object name NMTOKEN #REQUIRED >
  <!ATTLIST object value NMTOKEN #IMPLIED >

  <ELEMENT sort EMPTY >
```

```

<!ATTLIST sort all_objects NMOKEN #REQUIRED >
<!ATTLIST sort name NMOKEN #REQUIRED >
<!ATTLIST sort value NMOKEN #REQUIRED >
]>

```

An example of an XML structures for the topic specifications from the previous example are given in listing 4.6.

Listing 4.6: Topic specifications in SSL_T

```

<Topic_specs>
  <Topic msgtype="UAVMsg" name="topic1">
    <feature name="Altitude" value="msg.alt">
      <object name="uav1" />
    </feature>
  </Topic>

  <Topic msgtype="UAVMsg" name="topic2">
    <feature name="Speed" value="msg.spd">
      <object name="uav1" value="msg.id" />
    </feature>
  </Topic>

  <Topic msgtype="UAVMsg" name="topic3">
    <feature name="Altitude" value="msg.alt">
      <sort name="UAV" value="msg.id" all_objects="false" />
    </feature>
  </Topic>

  <Topic msgtype="UAVMsg" name="topic4">
    <feature name="Behind" value="msg.behind">
      <object name="uav1" value="msg.id1" />
      <object name="uav3" value="msg.id2" />
    </feature>
  </Topic>

  <Topic msgtype="UAVMsg" name="topic5">
    <feature name="Behind" value="msg.behind">
      <object name="uav2" value="msg.id1" />
      <object name="uav1" value="msg.id2" />
    </feature>
  </Topic>

  <Topic msgtype="UAVMsg" name="topic6">
    <feature name="Behind" value="msg.behind">
      <sort name="UAV" value="msg.id1" all_objects="true" />
      <object name="uav2" value="msg.id2" />
    </feature>
  </Topic>
</Topic_specs>

```

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 stores the data 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.

The aforementioned XML specifications consisting of topic specifications are queried in order to find relevant topics for the extracted features. Finding 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 containing the matching feature with matching arguments should be found. This process includes multiple steps. In the first step only topics containing the queried feature are extracted from the topic specifications. For example, assume that we are querying the list of topic specifications given in Listing 4.6 for relevant topics for the feature *Behind*[*uav1*,*uav2*] then the output from the first step of the matching would be the following topic specifications:

```
<Topic msgtype="UAVMsg" name="topic4">
  <feature name="Behind" value="msg.behind">
    <object name="uav1" value="msg.id1" />
    <object name="uav3" value="msg.id2" />
  </feature>
</Topic>

<Topic msgtype="UAVMsg" name="topic5">
  <feature name="Behind" value="msg.behind">
    <object name="uav2" value="msg.id1" />
    <object name="uav1" value="msg.id2" />
  </feature>
</Topic>

<Topic msgtype="UAVMsg" name="topic6">
  <feature name="Behind" value="msg.behind">
    <sort name="UAV" value="msg.id1" all_objects="true" />
    <object name="uav2" value="msg.id2" />
  </feature>
</Topic>
```

In the next step of the algorithm this new list of topic specifications is used to extract only those topics which contain matching arguments. To find topics with matching arguments an ontology is queried to find 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. Taking this into account the next step of the algorithm in the previous example 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, we are extracting topic specifications based on their second argument which in our example should either be the object `uav2` or the sort `UAV`. The following list of topic specifications is used as the input to this iteration of the algorithm.

```
<Topic msgtype="UAVMsg" name="topic4">
  <feature name="Behind" value="msg.behind">
    <object name="uav1" value="msg.id1" />
    <object name="uav3" value="msg.id2" />
  </feature>
</Topic>

<Topic msgtype="UAVMsg" name="topic6">
  <feature name="Behind" value="msg.behind">
    <sort name="UAV" value="msg.id1" all_objects="true" />
    <object name="uav2" value="msg.id2" />
  </feature>
</Topic>
```

In this case only topic `topic6` is relevant as topic `topic4` specifies feature `Behind` which has an object `uav3` as the second argument. Therefore the result of matching feature `Behind[uav1,uav2]` to topic specifications in Listing 4.6 is the following topic specification:

```
<Topic msgtype="UAVMsg" name="topic5">
  <feature name="Behind" value="msg.behind">
    <sort name="UAV" value="msg.id1" all_objects="true" />
    <object name="uav2" value="msg.id2" />
  </feature>
</Topic>
```

Topic `topic5` contains feature `Behind` for multiple objects as the first argument is a sort. Therefore, in order to get data for feature `Behind[uav1,uav2]` it is necessary to filter out messages which have `uav1` and `uav2` for the first and second argument respectively. In our design, 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 `id1 = 1` and `id2 = 2` would contain the data for feature `Behind[uav1,uav2]`.

Next we consider the case where a feature has one or more variable arguments. In this case, the first step of the matching algorithm is to find all possible values for variable arguments. As mentioned earlier, the domain of a variable corresponds to a sort in the ontology. Therefore to acquire all possible values 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 on an object level meaning that the arguments to these features need to be objects. This requires making all possible combinations with other arguments using the acquired objects. For example if *Behind*[*uav1*, *UAV*] is extracted from the logical formula and *UAV* has two instances in the domain, *uav1* and *uav2* then *Behind*[*uav1*, *uav1*] and *Behind*[*uav1*, *uav2*] would be the result after expansion. The expanded features are matched to topic specifications using the same algorithm for features with constant arguments presented earlier.

4.3 Integrating data from multiple platforms

Previous sections described a possible solution for semantic matching between the intended meaning of variables and content of streams. In some situations it might happen that a platform does not have the necessary streams to evaluate a logical formula. Platforms in a distributed system might be able to reuse relevant streams from distributed platforms. However, in order to do so ontologies on the distributed platforms need to describe the same parts of the environment as the ontology on which the formula is being evaluated.

Another issue is that even if the ontologies deal with the same part of real world they might not use the same concept names. An example of this problem is visible in the ontologies for the Multiple Platform Scenario described in section 3.3. The ontology for the first platform deals with both aerial and ground vehicles while the ontology for the second platform describes only the subset which covers cars. However, even though these two ontologies refer to the same cars in the real world the topic specification can not be used on different platforms because their names differ.

This problem can be solved with semantic mapping between two ontologies described in Section 3.3.2. The simplest approach to a semantic matching process with multiple platforms is to take each expanded feature and try to map this feature and its arguments to a feature on a distributed platform. If this is successful then the last step is to query the distributed platform's topic specification for this mapped feature using the matching algorithm described in the previous section.

For example, assume that we are trying to match the feature *Near*[*uav1*, *uav2*] to topic specifications on distributed platforms. The extract from the list of specified bridge rules is given in the following listing.

```
<owl:mapping>
  <owl:bridgeRule owl:br-type="equiv">
    <owl:sourceConcept rdf:resource="http://www.example.com/ontology#Near"/>
    <owl:targetConcept rdf:resource="http://www.another-example.com/ontology#
      Close"/>
  </owl:bridgeRule>

  <owl:bridgeRule owl:br-type="same">
    <owl:sourceConcept rdf:resource="http://www.example.com/ontology/instances#
      uav1"/>
    <owl:targetConcept rdf:resource="http://www.another-example.com/instances#
      uas22"/>
  </owl:bridgeRule>
```

```

</cowl:bridgeRule>
<cowl:bridgeRule cowl:br-type="same">
<cowl:sourceConcept rdf:resource="http://www.example.com/ontology/instances#
uav2"/>
<cowl:targetConcept rdf:resource="http://www.another-example.com/instances#
uas23"/>
</cowl:bridgeRule>
</cowl:mapping>

```

As the listing shows feature *Near* maps to feature *Close* on a distributed platform. The next step is to map all the arguments and in our example objects *uav1* and *uav2* map to objects *uas22* and *uas23* respectively. Therefore, feature *Close*[*uas22*, *uas23*] is used while querying for topic specifications on the distributed platform.

4.4 Design

The Single Platform Scenario introduced in section 1.2 deals with execution monitoring. One way to implement execution monitoring is to provide a set of control formulas which need to be evaluated at certain time points. The platform gets information about the environment from a number of different sensors. For example, the position might be provided by a GPS system, altitude by an on-board altimeter and distance to nearby objects by a proximity sensor. However, if we take a formula that says that the platform should never be closer than 3 meters to any building then in order to evaluate this formula it must be possible to acquire values for each symbol in the formula. As mentioned before, DyKnow adapts the raw sensor data and provides the output in the form of streams. Before these streams can be used for the formula evaluation, it is necessary to map features represented by symbols in the formula to streams containing the relevant data for these features.

In the Multiple Platform Scenario the platforms are monitoring a road for potential traffic violations. Similar to the Single Platform Scenario, each platform has a number of sensors which provide information about the environment, i.e. radar speed guns used to monitor the speed of cars on the road. Platforms deal with a set of formulas which need to be evaluated, i.e. no car on the monitored road should ever exceed a speed of 60 km/h for more than 5 minutes. As we mentioned in section 1.2 the monitored region is divided between the platforms in order to be able to monitor multiple potential traffic violations. Therefore, if a car is driving over the limit in one region for 2 minutes before it enters the other region then the platform needs data about the speed of the car from the other platform in order to determine if the driver made a traffic violation.

In both scenarios each formula might require a number of streams to be evaluated. However, fluent streams do not necessarily have the same valid times and therefore not all of them are available at the time point the system needs them.

Therefore, to support reasoning over incrementally available information

the system should solve a number of issues. First of all, the system should be able to evaluate logical formulas. To evaluate formulas each symbol in a formula should be automatically mapped to one or more streams which contain values for this symbol. However, before these streams can be used for the formula evaluation they need to be synchronized.

Our solution consists of three main components Stream Processor, Knowledge Manager (KM) and Formula Progressor which deal with the aforementioned issues. The ROS implementation of DyKnow is shown in figure 4.2.

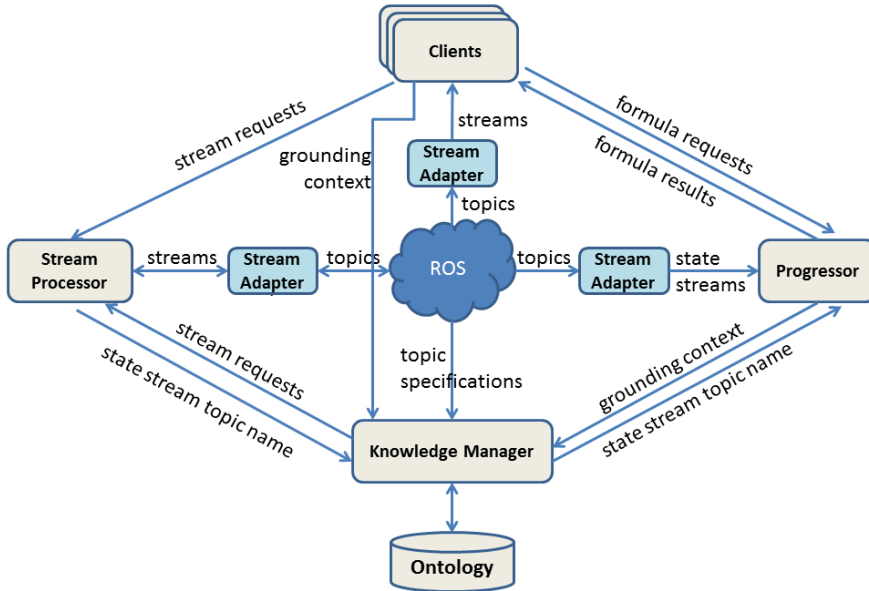


Figure 4.2: DyKnow architecture in ROS.

The Formula Progressor evaluates formulas through progression. If a formula is evaluated to false then certain measures should be taken, for example if the platform in the Single Platform Scenario is closer than 3 meters to a building measures should be taken in order to avoid collision. To do the evaluation the Formula Progressor requires the state stream name where the state stream represents a stream of state samples. State samples are defined as samples whose value is a state, i.e. a tuple of values. The process of state synchronization is done by the Stream Processor which provides the state stream topic name as an output.

The task of finding relevant topics (streams) for variables in the logical formula is passed to the Knowledge Manager. The Knowledge Manager supports the process of semantic matching of the intended meaning of variables in the formula and content of streams. Therefore, the Knowledge Manager accepts a grounding context as an input and provides a stream request as an

output. A grounding context contains a formula and stream constraints and specifies how to ground symbols in a formula to the content of topics in ROS. On the other hand, a stream request contains relevant topics for features in a formula together with all the necessary information needed for subscribing to these topics. To achieve this the Knowledge Manager requires access to a domain ontology and topic specifications in a system. Stream constraints from a grounding context are passed to the Stream Process which takes this information into account while setting up a state stream topic. In some cases variables in a logical formula might have multiple relevant topics. However, the Formula Progressor expects only a single sample for each variable in the formula. In our solution, a stream request contains all relevant topic specifications for each feature in a formula and the actual merging of these topics into a single output is done by the Stream Processor.

Stream adapters in the architecture bridge the gap between DyKnow and ROS and are used for adapting streams from DyKnow to topics in ROS and vice versa.

In distributed systems each platform is independent and implements the aforementioned components. The platforms can reuse data from other distributed platforms and to do this each platform has to be able to access both ontologies and topic specifications on different platforms. To support interoperability between the platforms the user has to specify semantic mappings between the ontologies.

The following chapter provides the implementation details for the Knowledge Manager.

4.5 Related work

Semantic Web technologies provide support for annotating data with semantic meta-data. Works by Bröring et al. [8] and Sheth et al. [28] use these technologies for representing the meaning of sensor observations on a sensor network which conforms to the Sensor Web Enablement (SWE). The SWE is an initiative proposed by the Open Geospatial Consortium (OGC) which aims at providing a set of standards for enabling discovery, exchange and processing of sensor observations [5]. In order to support these functionalities sensors need to be modeled to capture their characteristics. These characteristics include sensor name, sensor model as well as temporal (e.g. sensor sample rate), spatial (e.g. sensor location) and thematic characteristics (e.g. sensor provides data about water temperature). Bröring et al. [8] propose a publish/subscribe model for sensor network based on a semantic matchmaking mechanism. The semantic matching is done by creating an ontology for each publishing sensor based on its characteristic and an ontology which represents a requested service. The next step of semantic matching is to align these ontologies and determine if they match. If they do then the sensor provides relevant data for the service.

Similar work has been done by Goodwin and Russomanno [12] who propose a discovery system which semantically annotates sensors and their services. Sensor semantic annotations are based on the OntoSensor ontology [25] which describes sensor characteristics such as sensitivity, mass, frequency, etc.

Work done by Sheth et al. [28] presents an extension of sensor modeling languages in SWE which includes semantic annotations called Semantic Sensor Web. By connecting these semantic annotations to different ontologies it is possible to do more complex queries as well as reason about sensor domains. For example, a timestamp of a sensor observation could be semantically annotated with a concept from an ontology which describes temporal concepts. This extension would allow more complex queries about the timestamp, for example a query to determine if the timestamp is within some interval given that a concept of interval is described in the temporal ontology. It is important to note that in the case of Semantic Sensor Web the focus is on semantic annotation of sensor observations while the works by Bröring et al. and Goodwin and Russomanno focus on semantic sensor modeling.

Another example of the work which uses semantic technologies for knowledge representation is the KNOWROB framework [31]. KNOWROB is a knowledge processing framework for autonomous personal robots [31]. Ontologies in this framework are used for representing encyclopedic knowledge about an environment such as possible object classes or action classes in an environment. Individuals on the other hand represent specific instances of objects, actions or events which are computed from observations coming from sensors. The fact that the knowledge is encoded in ontologies makes it possible to share this knowledge among multiple systems. Somewhat similar approach is the idea of RoboEarth proposed by Waibel et al. [33]. RoboEarth is a World Wide Web for Robots and as such is used for sharing knowledge across multiple heterogeneous platforms. It consists of a database which stores both data and its semantic information encoded using the Web Ontology Language (OWL). The ontology used for encoding data stores information about objects, environments and action recipes. This allows both syntactic-based and semantic-based queries on data and therefore allows platforms to reuse knowledge which was acquired earlier.

In the Multiple Platform scenario we have dealt with the situation where multiple heterogeneous information sources have to share information in order to solve some task. One way to solve this interoperability problem is by using ontologies to semantically annotate information and information sources. Wache et al. [32] suggest that the works on ontology-based information integration deal with three types of ontology architectures: single ontology approach, multiple ontology approach and hybrid ontology approach. In the single ontology approach, multiple information sources have to conform to a single global ontology. In other words, a system has a mediator which contains a domain model and models of all information sources. The media-

tor also needs to have a set of mappings which relate concepts in models on individual information sources and concepts in a global domain model. An example of this architecture is SIMS [2] in which case queries are written using concepts from a global ontology. It is a mediator's job to decompose a query and relate its part to relevant information sources. Similar work was done by Cruz et al. [10] who proposed a peer to peer data management architecture for data integration. Their architecture consists of peers with local schemas and one super peer which contains the global ontology which defines the global vocabulary of a system [10]. The RoboEarth approach [33] can also be categorized into the single ontology approach as it consists of a single database which contains data and its semantic information.

In the multiple ontologies approach each source has its own local ontology instance which is used to capture the semantics of the information source. As Wache et al. [32] argue an advantage of this approach is the independence of information sources, meaning that changes in a local ontology do not influence other local ontologies. However, this independence also makes it difficult to compare information from different sources [32]. To deal with this problem it is necessary to establish semantic mappings between concepts in different distributed ontologies. An example of this architecture is the work done by Mena et al. [23] on the OBSERVER system. The OBSERVER system consists of multiple heterogeneous platforms connected with semantic mappings. Semantic mappings are based on lexical relations and represent mappings such as synonym, hyponym, hypernym, etc. Other approaches to semantic mappings such as those proposed by Bouquet et al. [6] and Serafini and Tamilin [26] allow mappings based on semantic correspondence. In this case it is possible to define relations such as equivalence, "more general" and "less general" between entities in different ontologies.

Finally, in the hybrid approach each information source has its own local ontology, however each of these ontologies is built upon some global vocabulary [32]. The fact that ontologies are built upon some global vocabulary makes it possible to compare information from different sources without the need for semantic mappings. A global vocabulary describes basic terms while local ontologies combine these terms to form complex terms. Work done by Stuckenschmidt et al. [30] proposes an approach in which a global vocabulary is represented in the form of an ontology which describes primitives of the vocabulary. However, similar to the single ontology approach queries which require information from multiple sources need to be written using the global vocabulary.

It is important to note that ontologies used for these data integration approaches can either be developed for a specific domain similar to our solution to the semantic matching problem or a relevant existing ontologies can be used. For example, OpenCyc ¹ ontology is an upper ontology meaning that it is used to encode generic knowledge about a number of domains. Another example are Semantic Web for Earth and Environmental

¹<http://www.cyc.com/opencyc>

Terminology (SWEET)² ontologies which are "middle-level" ontologies and are used in a combination with a domain specific ontology. This means that concepts from a domain specific ontology are related to concepts in SWEET ontologies thus allowing the extraction of new knowledge about concepts (from a domain specific ontology).

Our approach for information integration relates to the multiple ontology approach. Each platform has its own ontology and the platforms' ontologies are connected with semantic mappings. Semantic mappings in our approach are similar to the one proposed by Bouquet et al. [6]. The main reason for the choice of this approach was to allow greater independence between platforms thus allowing the platforms to define ontologies without the need for conforming to some global vocabulary (ontology). This independence also simplifies individual platforms as platforms only need to know about concepts in their local ontology. When it comes to querying distributed sources with this approach, platforms use concepts from the local ontology in queries. These concepts are then mapped to concepts in ontologies on distributed sources in order to extract relevant information.

²<http://sweet.jpl.nasa.gov/>

Chapter 5

Implementation

The chapter provides the implementation details for the Knowledge Manager (KM). This includes an overview of the design and components used in the implementation.

5.1 Introduction

The main task of the KM is the process of matching features in a logical formula to content of streams in the system. It accepts a formula as an input and provides relevant streams for each extracted feature from the formula. In order to support this service the KM needs to be provided with an ontology which represents the domain model of a platform and topic specifications represented in SSL_T . The KM also provides support for specifying new topics.

Currently, the KM supports reuse of data from distributed platforms. However, in order to do so the KM has to have access to ontologies and topic specifications on these distributed platforms together with semantic mappings which map concepts on a local platform with concepts on distributed platforms.

5.2 Proposed solution

The main characteristics of our implementation are:

- The implementation of the KM is written in Java
- The Jena Semantic Web Framework ¹ and its Ontology API is used to deal with ontologies

¹<http://jena.sourceforge.net/>

- The lexer and parser code for the topic specification language is generated using Another Tool for Language Recognition (ANTLR)²
- Pellet³ is used as a reasoning mechanism for the ontologies
- JNI implementation of rosjava⁴ is used for the integration with the Stream Processor and the Formula Progressor in ROS implementation of DyKnow

5.3 Design

The KM consists of four main components:

- Ontology component
- Topic Specification component
- Formula processing component
- KM interface

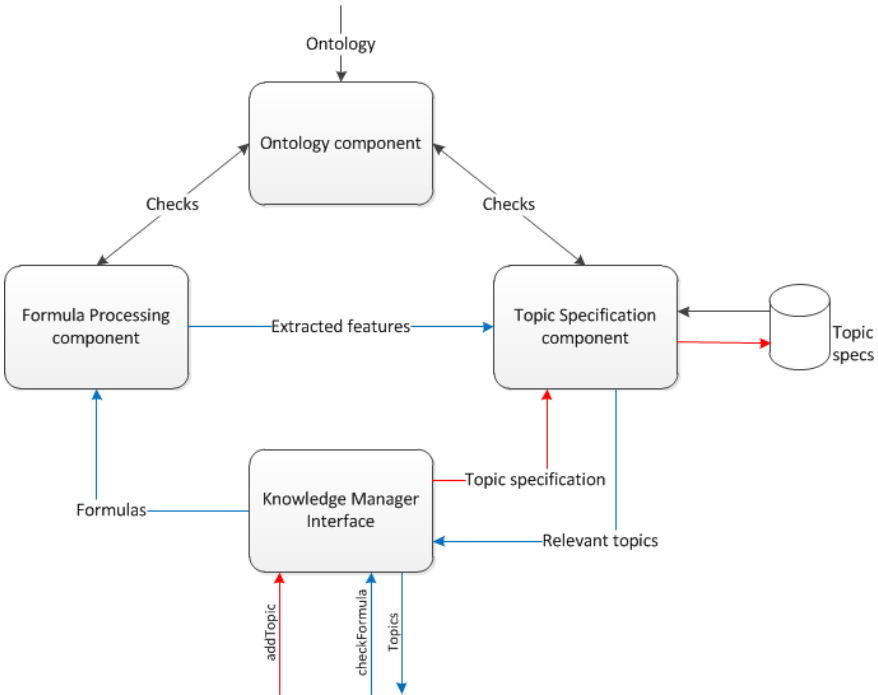


Figure 5.1: Components in ROS implementation of DyKnow.

²<http://www.antlr.org/>

³<http://clarkparsia.com/pellet/>

⁴<http://www.ros.org/wiki/rosjava>

Figure 5.1 gives an overview of relations between different components. The blue arrows show the process of matching formulas to a variable while the red arrows show the process of adding new topic specifications. The following sections describe the aforementioned components in more details.

Knowledge Manager interface

The KM interface provides a high level interface consisting of a number of methods which cover the main functionality of the KM:

- `checkFormula` - returns relevant topics for features in a logical formula
- `addTopic` - addition of new topic specifications
- `removeTopic` - removal of topic specification
- `getTopicSpecification` - returns a topic specification for some topic
- `listTopicSpecifications` - list all topic specifications in a system
- `listTopics` - list all topics in a system
- `listSort` - lists all objects of some sort

To provide these high level services the KM interface reuses the functionalities from other components. The interface is implemented as a façade pattern which simplifies the use of the KM while hiding the implementation details for other components.

Ontology component

The ontology component provides the necessary classes needed for the use of ontologies in the KM. The implementation mostly relies on the Jena Semantic Framework. To simplify the topic specifications and the processing of formulas we have disregarded the URIs in the resource names in the ontology. This was possible because the local names have to be unique on the ontology level and therefore uniquely reference the resources. This required the extension of the ontology model in the Jena Semantic Framework with methods which only deal with local names of resources.

The ontology component also implements methods which are used to check if some feature or features are present in the ontology and are correctly defined with respect to the ontology. As stated before DyKnow deals with features and objects. The same applies to domain ontologies in the KM. The domain ontology consists of a hierarchy of classes representing sorts(types) of objects. The actual object instances are represented as individuals of classes in the ontology. KM currently supports sort hierarchies which are represented as taxonomies meaning that the hierarchies are organized using the is-a (subclass/superclass) relationships between classes. The features,

on the other hand, are represented as intersection classes. Components of these intersection classes include the type of the relation (e.g. unary, binary) and classes which encode the accepted types of arguments of the relation.

The following two axioms give an example of the representation of features *Altitude* and *Behind* in the ontology.

$$\begin{aligned} \textit{Altitude} &\subseteq \textit{UnaryRelation} \cap \forall \textit{arg1}. \textit{UAV} \\ \textit{Behind} &\subseteq \textit{BinaryRelation} \cap \forall \textit{arg1}. (\textit{UAV} \cup \textit{Car}) \cap \forall \textit{arg2}. (\textit{UAV} \cup \textit{Car}) \end{aligned}$$

Altitude represents a unary relation whose argument is of sort *UAV* while *Behind* is defined as a binary relation whose arguments can either be of sort *UAV* or *Car*. As the example shows, the ontology has to have predefined classes which represent the arity of the relation (*UnaryRelation* and *BinaryRelation*) together with object properties (*arg1* and *arg2*) which are used to enumerate sorts of objects which can appear as arguments to a relation. This representation is similar to the predicate representation found in CycL⁵ which unlike OWL supports predicates with arity higher than two. The current implementation of the KM supports unary, binary and ternary relations.

Formula processing component

The task of the formula processing component is extraction of features from a logical formula. The formulas are represented in the metric temporal logic and the current implementation uses the metric temporal logic parser and lexer in order to extract the data needed for processing. This data includes types of quantifiers, quantified variables, domains of quantified variables and predicates/terms with arguments. Predicates or terms in formulas are treated as features in the ontology while the arguments are treated as objects (instances of classes). Domains of quantified variables are represented as classes in the ontology. An Abstract Syntax Tree (AST) containing this data is constructed during the parsing and lexical analysis and is used for extracting features. These features are then checked against the ontology using the ontology component to determine if features map to relations in the ontology. For example, if we have the feature *Altitude*[*uav1*] it is necessary to determine if a relation *Altitude* exists in the ontology and if *uav1* is a proper argument for this relation.

Topic Specification component

This component deals with the topic specifications. It has two main functionalities:

- It provides an interface for addition of semantic specifications for topics

⁵<http://www.cyc.com/>

- It provides an interface for searching the topic specifications based on the semantics of the topics

In order to implement these two functionalists the KM has to be able to determine the content of each topic. As discussed in the previous chapter, SSL_T provides support for encoding the content of topics. The topic specifications are saved in an XML file using the structure defined in 4.2. The XML file is queried using XQuery.

When adding new specifications each specification is checked in the ontology to determine if the feature exists in the ontology and its arguments satisfy the definition in the ontology. In our implementation more specific arguments are allowed, for example if feature Altitude is defined for aerial vehicles then the topic specification for altitude of unmanned aerial vehicle (subclass of aerial vehicle) is a valid specification.

The following section gives more details about the actual process of matching the features in a formula to topics in the KM.

5.4 Matching a formula to topics

The KM interface provides an entry point to the underlying system through a number of methods. This section focuses on the `checkFormula` method which is used for matching features in a formula to topics. The input to the `checkFormula` method is a string which represents the formula in the metric temporal logic. The formulas are first passed through the metric temporal logic parser which extracts the AST. This tree is used in the next step to extract all features from the formula. Constants in the formula are considered to be objects in the domain while in the case of variables we take that the domain of the variable represents some sort in the ontology. For example, for the following formula:

$$\forall x \text{ in } Car(Behind[car1, x])$$

car1 is considered an object in the ontology while the second argument of the feature would range over all instances of the class *Car* in the domain.

When all of the features in the formula have been extracted the next step is to check if these features correspond to the domain definition. In this step the domain ontology is checked to see if all features satisfy the requirements defined in the ontology. The requirements include name of the feature, arity and types of arguments. If the argument is a constant, checks are made to determine if the object with this name exists in the ontology. If this is the case, the sort of this object is determined in the ontology and used to determine if the argument type is correct. In the case of a variable, the domain of the variable (sort) is used for argument checks.

If the tests were successful and the formula is correct with respect to the ontology the set of features is sent to the Topic Specification component which needs to find relevant topics. The first task of the Topic Specification

component is to expand features which have sorts as arguments. To do this, the ontology is queried to get all objects in the domain which have these sorts as their classes. The next step is to make all possible combinations of the feature using the acquired objects. If we take the formula from the earlier example the following feature is sent to the Topic Specification component:

$$\textit{Behind}[\textit{car1}, \textit{Car}]$$

where Car represents the sort as described earlier. Given that the sort Car has two instances, car1 and car2, then the feature would be expanded into following features:

$$\begin{aligned} &\textit{Behind}[\textit{car1}, \textit{car1}] \\ &\textit{Behind}[\textit{car1}, \textit{car2}] \end{aligned}$$

The expanded features are added to the list of features which need to be synchronized to provide the necessary data used in the formula progression. The next step is to find relevant topics for these features. As stated in the previous chapter, all topic specifications are stored in the XML file. Given that all features in this set are on the object level (the arguments are objects) we consider that the relevant topics for some feature are those which have these objects or the sorts of these objects as arguments. This process was described in more detail in section 4.2.

Listing 5.1: Possible topics for feature Behind[car1, car2].

```
<Topic msgtype="CarMsg" name="topic1">
  <feature name="Behind" value="msg.behind">
    <sort name="Car" value="msg.id" />
    <sort name="Car" value="msg.id2" />
  </feature>
</Topic>
<Topic msgtype="CarMsg" name="topic1">
  <feature name="Behind" value="msg.behind">
    <sort name="Car" value="msg.id" />
    <sort name="car2" value="msg.id2" />
  </feature>
</Topic>
<Topic msgtype="CarMsg" name="topic1">
  <feature name="Behind" value="msg.behind">
    <sort name="car1" value="msg.id" />
    <sort name="Car" value="msg.id2" />
  </feature>
</Topic>
<Topic msgtype="CarMsg" name="topic1">
  <feature name="Behind" value="msg.behind">
    <sort name="car1" value="msg.id" />
    <sort name="car2" value="msg.id2" />
  </feature>
</Topic>
```

When checking for the relevant topics, the current implementation of the KM also tries to find equivalent features in the ontology and if such a feature is found the ontology is checked in order to find all objects which are the same as objects in the initial feature. If more than one such object is found for an argument then those objects are used to form combinations with objects from other arguments (if any). These combinations are then tested in the Ontology component to make sure that they comply with the ontology. If we take the running example and assume that *car45* is the same as *car1* and that the feature *Behind* has the equivalent feature *Back* which takes the arguments of type *Car* then features

$$\begin{aligned} &Back[car1, car2] \\ &Back[car45, car2] \end{aligned}$$

would be tested for relevant topics while testing for *Behind* [*car1*, *car2*]. It is obvious that in some cases a certain feature will have more relevant topics. With the current implementation all of these topics would be sent to the Stream Processor which should do the merging in order to get one output for each feature. The merging process could be based on relevance (chronology of the messages, ranking of sources, etc.).

5.5 Multiple Platform Scenario

In the Multiple Platform scenario the KM deals with multiple ontologies and multiple sources with topic specifications. In order to make use of data on different platforms there must exist some kind of relation between concepts in the local ontology and concepts in the distributed ontology. Therefore, it is required to specify mappings from local concepts in the ontology to concepts in the distributed ontology. This is done using the COWL mappings described in section 3.3.2. The source concepts in these mappings are the concepts in the local ontology while the concepts in distributed ontologies are target concepts. The mappings include both mappings between concepts and individuals where the mappings between concepts define subclass, superclass or equivalence relation while the mappings between individuals can only define equivalence.

The process of matching topics to formulas described in section 5.4 is extended with the support for checking topic specifications on distributed platforms. 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. To do this the KM first constructs an ontology which combines the local ontology and the distributed one. The new ontology model also includes relevant mappings and a reasoner (Pellet) which adds new entailments resulting from the relations in the model. 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 the equivalent feature exists and all arguments of the selected feature can be mapped to objects in the distributed ontology, the feature is added to the list of features that needs to be checked for correctness in the distributed 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 distributed 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 distributed platform and therefore it cannot provide this data. If the features are correct in the distributed ontology, the KM tries 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 distributed ontology.

Let us consider the example presented in the following figure.

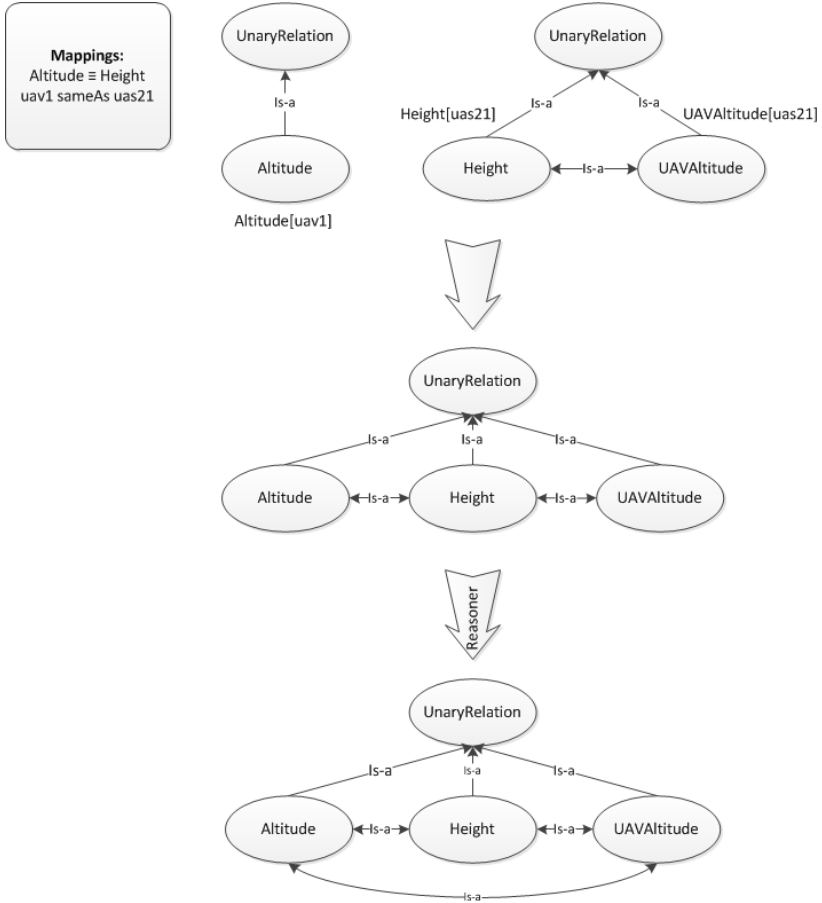


Figure 5.2: Multiple Platform scenario - example.

A mapping shows that the feature *Altitude* in the local ontology is equivalent to the feature *Height* in the distributed ontology. The feature *Height* is equivalent to the feature *UavAltitude* where both concepts are in the distributed ontology. While matching *Altitude* using the aforementioned 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. As described a reasoner adds new entailments to the ontology and in the case of the KM this 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* was also found in the example.

5.6 Integration

The components described so far are not ROS dependent in the sense that they do not use the ROS framework. This was mainly done to simplify the code for ROS integration and allow the use of the KM in other systems which are not based on ROS. Taking this into account the implementation for the integration only requires one class which uses the KM implementation. For the integration with other components in ROS we used the `rosjava.jni` which is based on Java Native Interface (JNI) meaning that the method calls are done by wrapping calls to C++ implementation of the client library.

As discussed earlier, the KM extracts an initial set of features from the formula and expands this set to form the final list of features (on the object level) that need to be synchronized. Each feature from this set has zero or more relevant topics which need to be merged into one output to provide necessary data to the Progressor. To deal with these sets, the KM provides the output in the form of `StateStreamSpecification`. This information is used by the Stream Processor to do the necessary subscriptions. `StateStreamSpecification` is a class that contains an array of `FeatureSpecifications`, one for each feature which needs to be synchronized. The `FeatureSpecification`, on the other hand, contains a set of relevant topics for the feature where each topic is represented with the class `FeatureTopicSpecification` which contains following fields: name, message type, the field containing the data and a set of `FeatureArgumentSpecifications`. `FeatureArgumentSpecification` specifies a single argument of a feature. If an argument is implicit in a topic then `FeatureArgumentSpecification` contains only the object identifier of the argument. On the other hand, if an arguments is computed from a field in a topic then it is necessary to specify the field from which the argument is computed together with the required value for that field.

The communication between ROS components is done using ROS services. The Progressor communicates with the KM via the Formula service provided by the KM.

Listing 5.2: CreateGroundingService service and relevant messages.

```

CreateGroundingContext.srv:
string formula
duration delay_before_approximation
StreamConstraints stream_constraints

bool success
string error_message
string stream_topic
string stream_constraint_violation_topic

StreamConstraints.msg:
time start_time           # 0 = now
time end_time             # 0 = forever
duration sample_period    # 0 = no sample period
duration sample_period_deviation # 0 = no deviation allowed
duration max_delay        # 0 = infinite
bool ordered_on_valid_time # do samples arrive ordered
                           # by valid time?
bool unique_valid_time    # are valid times unique?

```

As listing 5.2 shows the KM takes the string which represents the formula and returns the name of the stream topic which contains the data if the call was successful. Moreover, if a call is not successful then `error_message` is used for error reporting. The service call also includes the `StreamConstraints` which specify constraints on a stream and `delay_before_approximation` which determines how long the Stream Processor can wait before approximating streams. Given that the stream constraints can be violated the reply from the `CreateGroundingContext` contains the name of a topic where these constraint violations are published.

The Stream Processor is also implemented as a server and its main task is to provide the support for setting up streams. The service message (`CreateStateStream.srv`) is represented in listing 5.3. Request to this service is in the form of `StateStreamSpecification` message which represents a set of topics for each feature. The task of the ROS implementation of the KM is to take the output represented in the form of `StateStreamSpecification` and adapt it into the ROS version of this class.

Listing 5.3: State Processor service and relevant messages.

```

CreateStateStream.srv:
string stream_name
StateStreamSpecification state_stream_spec

bool success
string error_message

```

```

string stream_topic
string stream_constraint_violation_topic

StateStreamSpecification.msg
string state_name
FeatureSpecification[] features # Synchronize these features
duration delay_before_approximation
StreamConstraints stream_constraints

FeatureSpecification.msg
string feature_name
FeatureTopicSpecification[] topics # Merge these topics

FeatureTopicSpecification.msg:
string topic_name
string topic_msg_type
string topic_field
FeatureArgumentSpecification[] arguments

FeatureArgumentSpecification.msg:
bool constant
string object_identifier
string object_identifier_prefix
string field
string required_field_value
StringPair[] renaming_map

```

5.7 Summary

This chapter has presented a design of the Knowledge Manager (KM) which provides the support for semantic matching in the ROS implementation of DyKnow. It consists of four components: Knowledge Manager Interface, Ontology component, Topic Specification component and Formula Processing component. The Knowledge Manager Interface provides a set of high level methods which capture the main functionality of the KM. The Ontology component is used when dealing with ontologies, for example querying for objects of a certain sort or checking if a feature is correctly defined with respect to an ontology. The main task of the Formula Processing component is the extraction of features from a logical formula. Finally, the Topic Specification component allows adding new topic specifications and querying the existing set of topic specifications.

The KM supports use of information from distributed heterogeneous sources. In this case the process of semantic matching requires the specification of semantic mappings between the ontology on a local platform and

ontologies on distributed platforms. When searching for equivalent features on distributed platforms the KM makes use of the reasoning mechanism in order to allow discovery of equivalent classes which were not explicitly defined through semantic mappings.

The integration with the ROS implementation of DyKnow is done through ROS services where the KM defines a service which takes a formula and provides a set of relevant topic specifications for features in the formula.

Chapter 6

Case studies

This chapter presents two case studies which cover the functionality of the Knowledge Manager (KM). The first case study deals with a single platform scenario meaning that the platform uses only knowledge which is directly available to it. The second case study covers the multiple platform scenario. In this case the distributed system contains three platforms, one of which is the same as in the Single Platform Scenario.

6.1 Single platform scenario

This case study covers the Single Platform scenario covered in the previous chapters. The scenario deals with the situation where there is only one platform in the system doing execution monitoring by evaluating control formulas. The platform is provided with the set of topic specifications available in the system which it uses to determine relevant topics for the formula. The vocabulary of the topic specifications is provided in the form of an ontology.

Ontology

The ontology used in the Single Platform Scenario is presented in figure 6.1. As the figure shows the domain deals with two types of objects, static and moving objects. Static objects are some points of interest (house, road, crossing) while the moving objects define different types of vehicles (aerial vehicle, car, UAV).

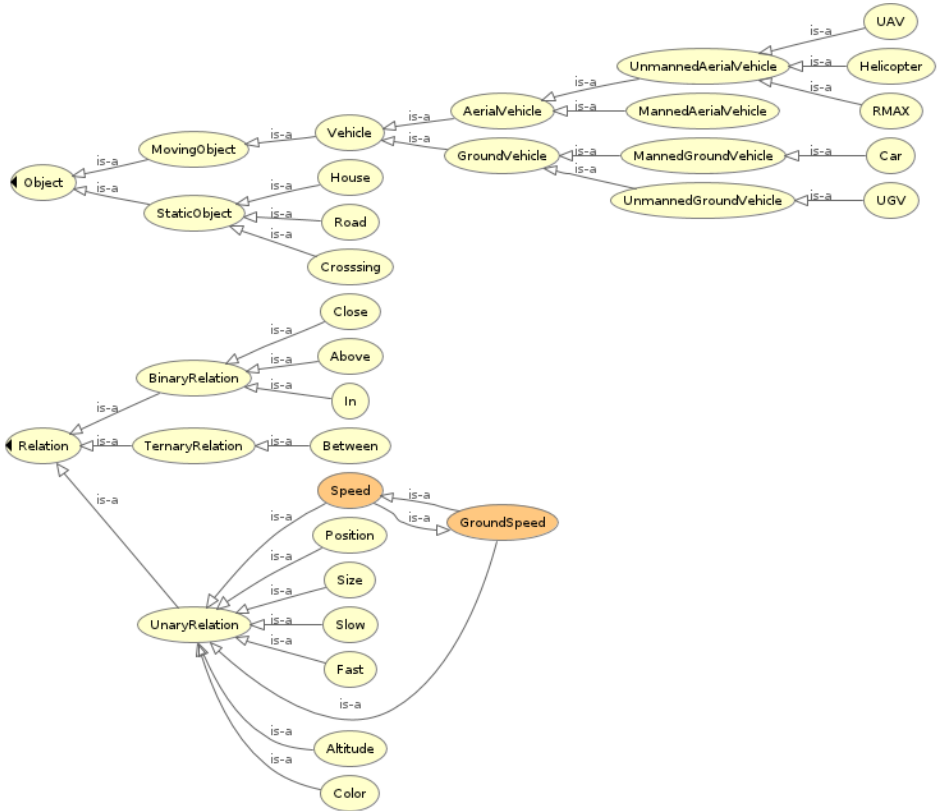


Figure 6.1: Visualization of the ontology for the single platform scenario.

The ontology also defines a number of relations differing in the number of arguments. Tables 6.1, 6.2 and 6.3 give an overview of the defined relations with types of their arguments.

Relation	Argument 1 type
<i>Altitude</i>	<i>AerialVehicle</i>
<i>Fast</i>	<i>MovingObject</i>
<i>Color</i>	<i>Vehicle</i>
<i>GroundSpeed</i>	<i>UAV</i>
<i>Slow</i>	<i>MovingObject</i>
<i>Size</i>	<i>Object</i>
<i>Speed</i>	<i>MovingObject</i>
<i>Position</i>	<i>Object</i>

Table 6.1: Unary relations and argument types.

Relation	Argument 1 type	Argument 2 type
<i>Above</i>	<i>Object</i>	<i>Object</i>
<i>In</i>	<i>Object</i>	<i>Object</i>
<i>Close</i>	<i>Object</i>	<i>Object</i>

Table 6.2: Binary relations and argument types.

Relation	Argument 1 type	Argument 2 type	Argument 3 type
<i>Between</i>	<i>Object</i>	<i>Object</i>	<i>Object</i>

Table 6.3: Ternary relation and argument types.

It is important to note that relations *Speed* and *GroundSpeed* are defined to be equivalent. However, they do not accept the same types of arguments.

This domain also defines a number of instances of classes which represent objects in the domain. We have the following objects in this domain:

UAVs **uav1** and **uav2**

Cars **car1**, **car2** and **car3**

Topic specifications

In this case study we define nine topics. Topic specifications with the syntax in SSL_T are presented in listing 6.1.

Listing 6.1: Topic specifications in SSL_T .

```

topic topic1:UAVMsg contains Altitude(uav1) = msg.alt for
    uav1 = msg.id, Speed(uav1)=msg.spd for uav1 = msg.id

topic topic2:UAVMsg contains Altitude(UAV) = msg.alt for
    some UAV = msg.id

topic topic3:CarMsg contains Speed(car1) = msg.spd for
    car1 = msg.id

topic topic4:CarMsg contains Speed(car2) = msg.spd for
    car2 = msg.id

topic topic5:CarMsg contains Speed(car3) = msg.spd for
    car3 = msg.id

topic topic6:UavMsg contains Close(uav1, uav2) = msg.close
    for uav1 = msg.id1, uav2 = msg.id2

topic topic7:UavMsg contains Close(uav2, uav1) = msg.close
    for uav1 = msg.id2, uav2 = msg.id1

```

```

topic topic8:UavMsg contains Close(uav1, UAV) = msg.close
    for uav1 = msg.id1, some UAV = msg.id2

topic topic9:UavMsg contains GroundSpeed(uav2) = msg.spd
    for uav2 = msg.id

```

Topic **topic1** contains altitude (field *alt*) and speed (field *spd*) for the object **uav1**. Similarly, **topic2** also contains altitude, however in this case for some objects of the sort *UAV*. Topics, **topic3**, **topic4** and **topic5**, contain the feature *Speed* for objects **car1**, **car2** and **car3** respectively. The next three topics contain the feature *Close* with different combinations of the arguments. Finally, **topic9** contains the feature *GroundSpeed* for the object **uav2**.

The XML representation of the topic specifications is given in Appendix C.1.

Formulas

We are going to test three formulas in this scenario. The formulas should show the main functionalities of the KM. The structure of the synchronization request will be provided for each formula.

Formula 1: $\Box((Altitude[uav1] > Altitude[uav2]) \text{ and } (Speed[uav1] > 50))$

The given formula covers the basic functions of the KM. It does not require expansions because the formula does not have any quantified variables. Therefore we only need to test the three features that appear in the formula.

Listing 6.2: Output from the KM for formula 1.

```

Feature: Altitude[uav1]
  Topics to merge:
    Topic: topic1, Message type: UavMsg, Id: msg.id = 1,
      Field: msg.alt

    Topic: topic2, Message type: UavMsg, Id: msg.id = 1,
      Field: msg.alt

Feature: Altitude[uav2]
  Topics to merge:
    Topic: topic2, Message type: UavMsg, Id: msg.id = 2,
      Field: msg.alt

Feature: Speed[uav1]
  Topics to merge:
    Topic: topic1, Message type: UavMsg, Id: msg.id = 1,
      Field: msg.spd

```

The feature *Altitude[uav1]* requires the merging of two topics because both *topic1* and *topic2* are relevant for this feature. On the other hand, *Altitude[uav2]* has only one candidate, *topic2*.

Topic *topic1* is the only relevant topic for feature *Speed[uav1]* because it is the only topic which contains feature *Speed* for the object **uav1**.

Formula 2: *forall x in UAV y in Car \square (Altitude[x] > 10 and Speed[x] > Speed[y])*

In this case we are dealing with two quantified variables, x with *UAV* as the domain and y which has *Car* as the domain. Therefore the expansion is needed and as listing 6.3 shows the features were expanded to cover all objects of the aforementioned domains. The output for features *Altitude[uav1]*, *Altitude[uav2]* and *Speed[uav1]* is the same as in the previous example. However, this formula also requires feature *Speed[uav2]* which is not explicitly defined with any topic in the system. The only possible candidate for this feature is *topic9* which defines an equivalent feature *GroundSpeed* for **uav2**.

Finally, topics *topic3*, *topic4* and *topic5* are relevant for features *Speed[car1]*, *Speed[car2]* and *Speed[car3]*.

Listing 6.3: Output from the KM for formula 2.

```
Feature: Altitude[uav2]
Topics to merge:
  Topic: topic2, Message type: UavMsg, Id: msg.id = 2,
    Field: msg.alt

Feature: Altitude[uav1]
Topics to merge:
  Topic: topic1, Message type: UavMsg, Id: msg.id = 1,
    Field: msg.alt

  Topic: topic2, Message type: UavMsg, Id: msg.id = 1,
    Field: msg.alt

Feature: Speed[uav2]
Topics to merge:
  Topic: topic9, Message type: UavMsg, Id: msg.id = 2,
    Field: msg.spd

Feature: Speed[uav1]
Topics to merge:
  Topic: topic1, Message type: UavMsg, Id: msg.id = 1,
    Field: msg.spd

Feature: Speed[car3]
Topics to merge:
```

Topic: topic5, Message type: CarMsg, Id: msg.id = 3,
Field: msg.spd

Feature: Speed[car2]

Topics to merge:

Topic: topic4, Message type: CarMsg, Id: msg.id = 2,
Field: msg.spd

Feature: Speed[car1]

Topics to merge:

Topic: topic3, Message type: CarMsg, Id: msg.id = 1,
Field: msg.spd

Formula 3: $\text{forall } x \text{ in UAV } \square(\text{Close}[uav1, x])$

In this formula we have a binary feature *Close*. As the formula shows, one of the arguments to the feature is quantified therefore we need to expand the feature.

The output from the KM in listing 6.4 shows that both **topic6** and **topic8** might contain data for feature *Close*[uav1, uav2] while **topic8** is the only candidate for feature *Close*[uav1, uav1].

Listing 6.4: Output from the KM for formula 3.

Feature: Close[uav1, uav2]

Topics to merge:

Topic: topic6, Message type: UavMsg, Id: msg.id1 = 1,
Id: msg.id2 = 2, Field: msg.close

Topic: topic8, Message type: UavMsg, Id: msg.id1 = 1,
Id: msg.id2 = 2, Field: msg.close

Feature: Close[uav1, uav1]

Topics to merge:

Topic: topic8, Message type: UavMsg, Id: msg.id1 = 1,
Id: msg.id2 = 1, Field: msg.close

6.2 Multiple platform scenario

In this scenario there are three platforms in the system including the platform from the previous scenario. An example of this scenario was presented in section 1.2 which covered a situation where multiple platforms are monitoring a road for traffic violations.

In order to show the differences between different scenarios the formulas are again checked on the platform from the previous scenario however in this case this platform has the possibility to use information from two distributed heterogeneous platforms.

Ontologies

As mentioned before the platform on which we are testing formulas is the same one we used in section 6.1 (named **platform 1** in this scenario). The ontology is given in figure 6.1. The system also contains two other platforms, **platform 2** which deals only with cars in the domain and **platform 3** which deals only with Aircraft Systems.

The ontology used on **platform 2** is given in figure 6.2.

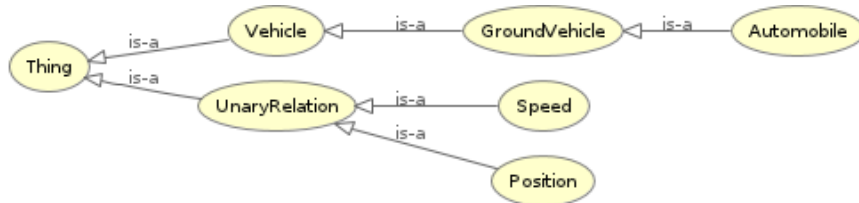


Figure 6.2: Visualization of the ontology for platform 2.

Two unary relations, *Position* and *Speed*, are defined in the ontology. The table 6.4 gives an overview of the argument types for the relations.

Relation	Argument 1 type
<i>Speed</i>	<i>Automobile</i>
<i>Position</i>	<i>Vehicle</i>

Table 6.4: Unary relations and argument types for platform 2.

The ontology also defines two objects, **car11** and **car12** which are of sort *Automobile*.

Platform 3 has an ontology defining Unmanned Aircraft Systems.

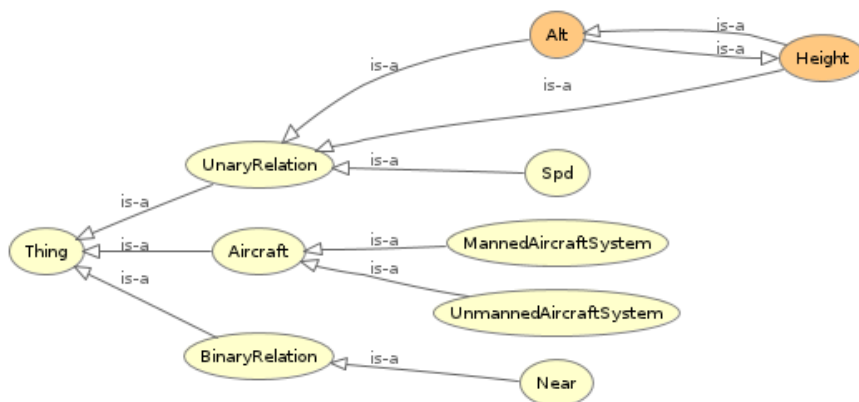


Figure 6.3: Visualization of the ontology for platform 3.

As shown in figure 6.3 the ontology for **platform 3** defines two types of aircrafts, manned and unmanned aircraft system. The ontology also defines a number of relations presented in listings 6.5 and 6.6. Features *Alt* and *Height* are defined to be equivalent.

Relation	Argument 1 type
<i>Alt</i>	<i>Aircraft</i>
<i>Height</i>	<i>Aircraft</i>
<i>Spd</i>	<i>Aircraft</i>

Table 6.5: Unary relations and argument types for platform 3.

Relation	Argument 1 type	Argument 2 type
<i>Near</i>	<i>Aircraft</i>	<i>Aircraft</i>

Table 6.6: Binary relations and argument types for platform 3.

There are 5 defined objects in the ontology: **uas20**, **uas21** and **uas22** of sort *UnmannedAircraftSystem* and **heli1** and **heli2** of sort *MannedAircraftSystem*.

Mappings

In order for **platform 1** to reuse knowledge from **platform 2** and **platform 3** we need to specify mappings between concepts in these ontologies. Listing 6.5 gives the mappings used in this scenario. In this solution we only used mappings which show equivalence between classes (bridge type equiv) and equivalence between individuals (bridge type same), however other mappings are possible.

Listing 6.5: Mappings in XML.

```
<owl:mapping>
  <owl:bridgeRule owl:br-type="equiv">
    <owl:sourceConcept rdf:resource="http://www.example.com/ontology#
      GroundVehicle"/>
    <owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
      /2011/5/Distributed.owl#GroundVehicle"/>
  </owl:bridgeRule>
  <owl:bridgeRule owl:br-type="equiv">
    <owl:sourceConcept rdf:resource="http://www.example.com/ontology#Car"/>
    <owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
      /2011/5/Distributed.owl#Automobile"/>
  </owl:bridgeRule>
  <owl:bridgeRule owl:br-type="equiv">
    <owl:sourceConcept rdf:resource="http://www.example.com/ontology#Position"/>
    <owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
      /2011/5/Distributed.owl#Position"/>
  </owl:bridgeRule>
  <owl:bridgeRule owl:br-type="equiv">
    <owl:sourceConcept rdf:resource="http://www.example.com/ontology#Speed"/>
    <owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
      /2011/5/Distributed.owl#Speed"/>
  </owl:bridgeRule>
  <owl:bridgeRule owl:br-type="equiv">
    <owl:sourceConcept rdf:resource="http://www.example.com/ontology#UAV"/>
    <owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
      /2011/6/distributed_uavs.owl#UnmannedAircraftSystem"/>
  </owl:bridgeRule>
  <owl:bridgeRule owl:br-type="equiv">
    <owl:sourceConcept rdf:resource="http://www.example.com/ontology#Close"/>
    <owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
      /2011/6/distributed_uavs.owl#Near"/>
  </owl:bridgeRule>
</owl:bridgeRule owl:br-type="equiv">
```



```

<owl:sourceConcept rdf:resource="http://www.example.com/ontology#Altitude"/>
<owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
/2011/6/distributed_uavs.owl#Height"/>
</owl:bridgeRule>
<owl:bridgeRule owl:br-type="equiv">
<owl:sourceConcept rdf:resource="http://www.example.com/ontology#Speed"/>
<owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
/2011/6/distributed_uavs.owl#Spd"/>
</owl:bridgeRule>
<owl:bridgeRule owl:br-type="same">
<owl:sourceConcept rdf:resource="http://www.co-ode.org/ontologies/ont.owl#
uav1"/>
<owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
/2011/6/distributed_uavs.owl#uas20"/>
</owl:bridgeRule>
<owl:bridgeRule owl:br-type="same">
<owl:sourceConcept rdf:resource="http://www.co-ode.org/ontologies/ont.owl#
uav2"/>
<owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
/2011/6/distributed_uavs.owl#uas21"/>
</owl:bridgeRule>
<owl:bridgeRule owl:br-type="same">
<owl:sourceConcept rdf:resource="http://www.co-ode.org/ontologies/ont.owl#
car1"/>
<owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
/2011/5/Distributed.owl#car11"/>
</owl:bridgeRule>
<owl:bridgeRule owl:br-type="same">
<owl:sourceConcept rdf:resource="http://www.co-ode.org/ontologies/ont.owl#
car2"/>
<owl:targetConcept rdf:resource="http://www.semanticweb.org/ontologies
/2011/5/Distributed.owl#car12"/>
</owl:bridgeRule>
</owl:mapping>

```

Topic specifications

There are 3 topics defined for platform 2 and they are presented in listing 6.6

Listing 6.6: Topic specifications for platform 2 in SSL_T .

```

topic carstopic1:AutomobileMsg contains Speed(Automobile)
    = msg.speed for some Automobile = msg.id

topic carstopic2:AutomobileMsg contains Speed(car11) =
    msg.speed for car11 = msg.id

topic carstopic3:AutomobileMsg contains
    Position(Automobile) = msg.speed for every Automobile
    = msg.id

```

As the listing shows, **carstopic1** and **carstopic2** define feature *Speed*. **carstopic1** contains data about *Speed* for some objects of sort *Automobile* while **carstopic2** contains feature *Speed* for **car11**. **carstopic3** contains feature *Position* for all objects of the sort *Automobile*.

Listing 6.7: Topic specifications for platform 3 in SSL_T .

```

topic uastopic1:UasMsg contains
    Spd(UnmannedAircraftSystem) = msg.speed for every
    UnmannedAircraftSystem = msg.id

topic uastopic2:UasMsg contains Spd(uas20) = msg.speed for
    uas20 = msg.id

```

```

topic uastopic3:UasMsg contains Spd(uas22) = msg.speed for
    uas22 = msg.id

topic uastopic4:UasMsg contains Height(uas20) = msg.height
    for uas20 = msg.id

topic uastopic5:UasMsg contains Alt(uas21) = msg.alt for
    uas21 = msg.id

topic uastopic6:UasMsg contains
    Near(UnmannedAircraftSystem, uas22) = msg.near for
    every UnmannedAircraftSystem = msg.id1, uas22 = msg.id2

topic uastopic7:UasMsg contains
    Near(UnmannedAircraftSystem, uas21) = msg.near for
    every UnmannedAircraftSystem = msg.id1, uas21 = msg.id2

topic uastopic8:UasMsg contains Near(uas21,
    UnmannedAircraftSystem) = msg.near for uas21 =
    msg.id2, every UnmannedAircraftSystem = msg.id2

```

First three topics contain feature *Spd* for the whole sort *UnmannedAircraftSystem*, **uas20** and **uas22** respectively. Topics **uastopic4** and **uastopic5** contain features *Height* for **uas20** and *Alt* for **uas21**. Finally, topics **uastopic6**, **uastopic7** and **uastopic8** contain feature *Near* with different objects (sorts) in the argument lists.

Formulas

In this section we are going to test the formulas from the single platform scenario. The section will highlight which topics are acquired from the distributed sources.

Formula 1: $\Box((\text{Altitude}[uav1] > \text{Altitude}[uav2]) \text{ and } (\text{Speed}[uav1] > 50))$

The listing 6.8 shows the output from the KM when the platform uses knowledge from distributed sources. Given that in this formula we only use objects of sort *UAV* the output does not contain any topics from **platform 2**. It is important to note that **uastopic4** was also included even though there is no direct mapping between *Altitude* on **platform 1** and *Height* on **platform 3**. As mention earlier this was possible because the ontology reasoner includes all entailments resulting from the transitive property of equivalence relation.

Listing 6.8: Output from the KM for formula 1 in distributed system.

```

Feature: Altitude[uav1]
Topics to merge:

```

Topic: topic1, Message type: UavMsg, Id: msg.id = 1,
Field: msg.alt

Topic: topic2, Message type: UavMsg, Id: msg.id = 1,
Field: msg.alt

Topic: uastopic4, Message type: UasMsg, Id: msg.id =
20, Field: msg.height

Feature: Altitude[uav2]

Topics to merge:

Topic: topic2, Message type: UavMsg, Id: msg.id = 2,
Field: msg.alt

Topic: uastopic5, Message type: UasMsg, Id: msg.id =
21, Field: msg.alt

Feature: Speed[uav1]

Topics to merge:

Topic: topic1, Message type: UavMsg, Id: msg.id = 1,
Field: msg.spd

Topic: uastopic1, Message type: UasMsg, Id: msg.id =
20, Field: msg.speed

Topic: uastopic2, Message type: UasMsg, Id: msg.id =
20, Field: msg.speed

Formula 2: *forall x in UAV y in Car* $\square(Altitude[x] > 10 \text{ and } Speed[x] > Speed[y])$

The output in listing 6.9 is very similar to the output in the previous example. However, in this situation the formula also includes features with arguments of type *Car*. As it was previously suggested **platform 2** contains data on objects of sort *Automobile* which directly map to objects of sort *Car* on **platform 1**.

Listing 6.9: Output from the KM for formula 2 in distributed system.

Feature: Altitude[uav2]

Topics to merge:

Topic: topic2, Message type: UavMsg, Id: msg.id = 2,
Field: msg.alt

Topic: uastopic5, Message type: UasMsg, Id: msg.id =
21, Field: msg.alt

Feature: Altitude[uav1]

Topics to merge:

Topic: topic1, Message type: UavMsg, Id: msg.id = 1,
Field: msg.alt

Topic: topic2, Message type: UavMsg, Id: msg.id = 1,
Field: msg.alt

Topic: uastopic4, Message type: UasMsg, Id: msg.id =
20, Field: msg.height

Feature: Speed[uav2]

Topics to merge:

Topic: topic9, Message type: UavMsg, Id: msg.id = 2,
Field: msg.spd

Topic: uastopic1, Message type: UasMsg, Id: msg.id =
21, Field: msg.speed

Feature: Speed[uav1]

Topics to merge:

Topic: topic1, Message type: UavMsg, Id: msg.id = 1,
Field: msg.spd

Topic: uastopic1, Message type: UasMsg, Id: msg.id =
20, Field: msg.speed

Topic: uastopic2, Message type: UasMsg, Id: msg.id =
20, Field: msg.speed

Feature: Speed[car3]

Topics to merge:

Topic: topic5, Message type: CarMsg, Id: msg.id = 3,
Field: msg.spd

Feature: Speed[car2]

Topics to merge:

Topic: topic4, Message type: CarMsg, Id: msg.id = 2,
Field: msg.spd

Topic: carstopic1, Message type: AutomobileMsg, Id:
msg.id = 12, Field: msg.speed

Feature: Speed[car1]

Topics to merge:

Topic: topic3, Message type: CarMsg, Id: msg.id = 1,
Field: msg.spd

Topic: carstopic1, Message type: AutomobileMsg, Id:
msg.id = 11, Field: msg.speed

```
Topic: carstopic2, Message type: AutomobileMsg, Id:
      msg.id = 11, Field: msg.speed
```

Formula 3: *forall x in UAV* $\square(\text{Close}[\text{uav1}, x])$

Similar to formula 1, formula 3 also does not contain any objects of sort *Car* and therefore can not use any data from platform 2. `uastopic7` is the only topic included from topic specifications on platform 3.

Listing 6.10: Output from the KM for formula 3 in distributed system.

```
Feature: Close[uav1, uav2]
```

```
Topics to merge:
```

```
Topic: topic6, Message type: UavMsg, Id: msg.id1 = 1,
      Id: msg.id2 = 2, Field: msg.close
```

```
Topic: topic8, Message type: UavMsg, Id: msg.id1 = 1,
      Id: msg.id2 = 2, Field: msg.close
```

```
Topic: uastopic7, Message type: UasMsg, Id: msg.id1 =
      20, Id: msg.id2 = 21, Field: msg.near
```

```
Feature: Close[uav1, uav1]
```

```
Topics to merge:
```

```
Topic: topic8, Message type: UavMsg, Id: msg.id1 = 1,
      Id: msg.id2 = 1, Field: msg.close
```

6.3 Discussion

The case studies presented in this chapter covered the main functionalities of the KM. The first test covered the Single Platform Scenario which dealt with a single platform implementing execution monitoring. Three formulas with different levels of complexity were tested. The KM managed to extract features from the formulas and find relevant streams for them based on the streams' content.

In the Multiple Platform Scenario the distributed system consists of three platforms covering different parts of the domain. In this test we used the same formulas in order to emphasize differences between outputs in these two tests. This scenario showed how the KM can reuse data from multiple distributed and heterogeneous platforms. The interoperability between the platforms was supported by semantic mappings between concepts in different ontologies.

The output also showed that with the current implementation of the KM the expansion of features with quantified arguments can significantly increase the number of features that need to be checked against topic specifications. If we also include features which are acquired through reasoning then

this increase can not be ignored. Another way to deal with this problem is to query the topic specifications with non-expanded features. In other words if we extract the feature *Altitude*[*UAV*] from a formula where *UAV* is a sort then this feature should also be used to query topic specifications. With this approach querying topic specifications which contain features which have sorts as arguments is trivial. However, if we assume that *UAV* has two instances, **uav1** and **uav2** then topics specifying features *Altitude*[*uav1*] and *Altitude*[*uav2*] are also relevant for feature *Altitude*[*UAV*] and should be included in the solution. In order to support this the queries would require a number of sub-queries which would be used to determine objects of a sort. Therefore, with this approach the semantic matching process would deal with a smaller number of features but the actual queries against topic specifications would be more complex.

Chapter 7

Performance evaluation

This chapter presents a performance evaluation of the Knowledge Manager (KM) in the ROS implementation of DyKnow. The goal of the performance evaluation is to see which aspects of semantic matching affect the performance of the KM and in what way.

7.1 Test cases

This performance evaluation is based on execution time. In this case we consider the execution time to be the time it takes to produce a call to the Stream Processor after acquiring a formula. Given that the semantic matching consists of a number of steps, the execution time is also divided in five phases:

- Preprocessing – includes loading of ontologies and topic specifications into memory
- Extracting features – includes feature extraction from a logical formula
- Checking features – includes the process of checking extracted features against an ontology
- Matching topic specifications – process of querying for relevant topic specifications
- ROS integration – transforming classes used in the KM to classes used in ROS implementation of the KM

The test cases used in the performance evaluation are based on four different aspects of semantic matching:

1. Size of an ontology – number of concepts, number of relevant and irrelevant ¹ individuals in an ontology

¹Relevant individuals are individuals which are of the same class as objects in the formula

2. Number of topic specifications – number of relevant and irrelevant topic specifications
3. Size of a formula – number of features in a formula
4. Type of features in the formula – quantified and non-quantified arguments

Taking into account these different aspects we have designed the following test cases for performance evaluation:

Test case 1.1 – the number of concepts in an ontology is varied (0 to 200) while the number of individuals and topic specifications is kept constant

Test case 1.2 – the number of irrelevant individuals is varied (0 to 200) while the number of concepts and topic specifications is kept constant

Test case 1.3 – the number of relevant individuals is varied (0 to 200) while the number of concepts and topic specifications is kept constant

Test case 2.1 – the number of irrelevant topic specifications is varied (0 to 500) while the number of concepts, individuals and relevant topic specifications is kept constant

Test case 2.2 – the number of relevant topic specifications is varied (100 to 500) while the number of concepts, individuals and irrelevant topic specifications is kept constant

Test case 3 – the number of features in the formula is varied (10 to 50) while the number of concepts, individuals and topic specifications is kept constant

Test case 4 – two versions of the same formula are tested, one with quantifiers and one without. This is done for three formulas with varying number of features (3, 9, 27).

7.2 Test setup

The test cases were run on a Dell Studio 1558 equipped with a 1.6 GHz six-core Intel i7-720QM processor and 4 gigabytes of memory running Ubuntu 11.04 with ROS 1.4.9 (Diamondback).

Ontologies and topic specifications used in the test cases were randomly generated. Each generated ontology contains at least concepts A, B, C, D and E where concepts B, C, D and E are subclasses of A. Ontologies also need to include at least 3 individuals (b1, b2, b3 of class B) which are relevant for logical formulas used in the test cases and therefore can not change throughout the tests. Ontologies define three relations: X, Y and

Z where X is a unary, Y is a binary and Z is a ternary relation. All of the aforementioned relations accept arguments of sort A. Other concepts in ontologies were generated randomly and were either defined to be a top concept or a subconcept of a randomly chosen previously generated concept.

7.3 Results

Test case 1.1 - Varying number of concepts

An ontology with 200 concepts and 200 individuals was generated for this test case. Out of 200 individuals, 3 are relevant (b1, b2, b3) and the rest are irrelevant. The following formula with 9 features was used in the test case:

$$X[b1] \text{ and } X[b2] \text{ and } X[b3] \text{ and } Y[b1, b1] \text{ and } Y[b1, b2] \text{ and } Y[b1, b3] \\ \text{and } Z[b1, b2, b1] \text{ and } Z[b1, b2, b2] \text{ and } Z[b1, b2, b3]$$

The setup of the test case included in total 20 topic specifications. Each feature had exactly one relevant topic specification and the rest were irrelevant for features in the formula.

In each iteration of the test we added 25 concepts to the ontology while keeping the number of individuals and topic specifications constant.

Number of concepts	25	50	75	100	125	150	175	200
Preprocessing	197	202	173	181	183	203	176	189
Extracting features	28	32	28	32	30	26	28	28
Checking features	1807	2379	2937	4651	5308	5595	5978	6426
Matching topic specs	546	542	554	544	558	562	551	547
Integrating with ROS	2	2	2	2	2	2	2	2
Total	2580	3157	3694	5410	6081	6388	6735	7192

Table 7.1: Varying number of concepts.

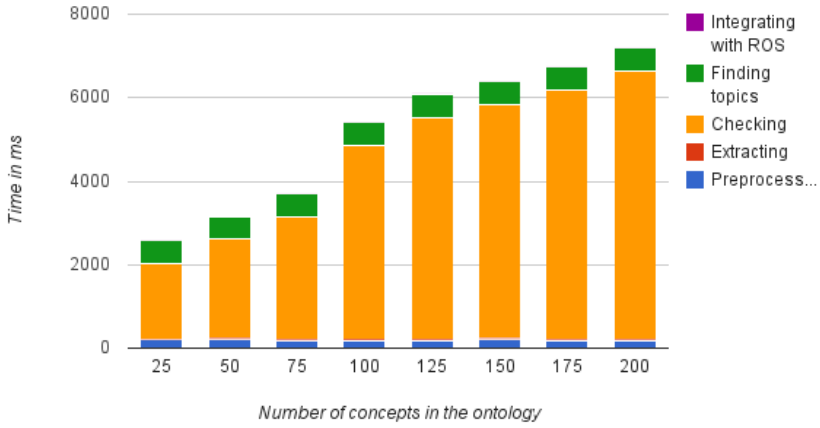


Figure 7.1: Varying number of concepts.

Figure 7.1 shows a linear increase in time when increasing number of concepts in the ontology from 25 to 75 and from 100 to 200. However, there is a sudden increase in execution time when going from 75 to 100 concepts in the ontology. Given that this increase is more obvious in the phase of feature checking one could accredit this to the internals of the Jena Semantic Web Framework. As expected, an increase in the number of concepts in the ontology has the highest impact on the phase of feature checking while the phase of matching topic specifications is constant.

Test case 1.2 - Varying number of irrelevant individuals

This test case uses the same formula, ontology and topic specifications from the previous test case. However, in this case in each iteration of the test case 25 individuals were added to the ontology while keeping number of concepts constant (200 concepts from the previous test case).

Num. of irrel. ind.	0	25	50	75	100	125	150	175	200
Preprocessing	188	186	206	184	172	191	185	178	189
Extracting features	29	29	29	30	29	29	29	29	27
Checking features	4944	5219	5465	5682	5805	6032	6205	6305	6450
Matching topic specs	575	560	557	546	550	561	567	558	570
Integrating with ROS	3	2	2	2	2	3	2	2	3
Total	5739	5996	6259	6444	6558	6816	6988	7072	7239

Table 7.2: Varying number of irrelevant individuals.

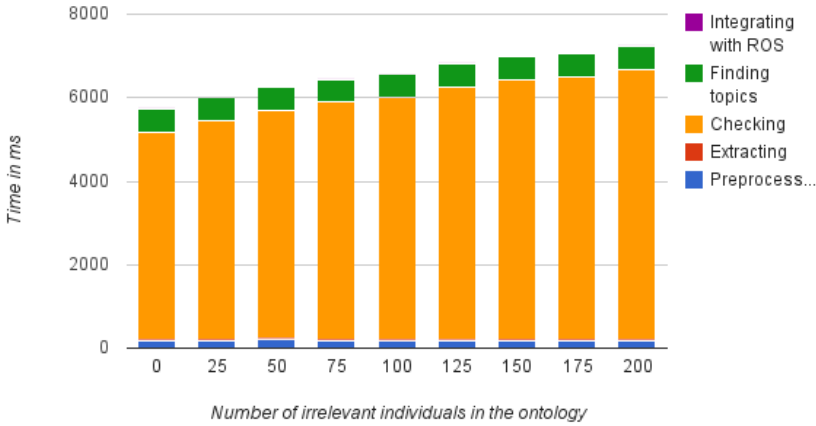


Figure 7.2: Varying number of irrelevant individuals.

Figure 7.2 shows that an increase in irrelevant individuals impacts the execution time far less than an increase in the number of concepts (an increase of approximately 1.5 seconds when going from 0 to 200 individuals compared to approximately 5 seconds when going from 25 - 200 concepts). Similar to the previous test case, table 7.2 shows that the increase in time is most obvious in the feature checking phase.

Test case 1.3 - Varying number of relevant individuals

Individuals in the ontology used in previous test cases are replaced with 200 relevant individuals (individuals of the same class as b1, b2 and b3). The ontology also contains 200 concepts from the previous test case. The test case reuses topic specifications and the formula from earlier tests.

Similar to the previous test case 25 individuals are added in every iteration of the test.

Num. of rel. ind.	0	25	50	75	100	125	150	175	200
Preprocessing	178	176	183	203	190	198	191	191	176
Extracting features	28	30	30	29	30	30	29	32	29
Checking features	5086	5164	5400	5657	5715	5984	6060	6295	6514
Matching topic specs	565	549	555	556	721	567	791	574	582
Integrating with ROS	2	2	2	2	2	3	2	2	2
Total	5859	5921	6170	6447	6658	6782	7073	7094	7303

Table 7.3: Varying number of relevant individuals.

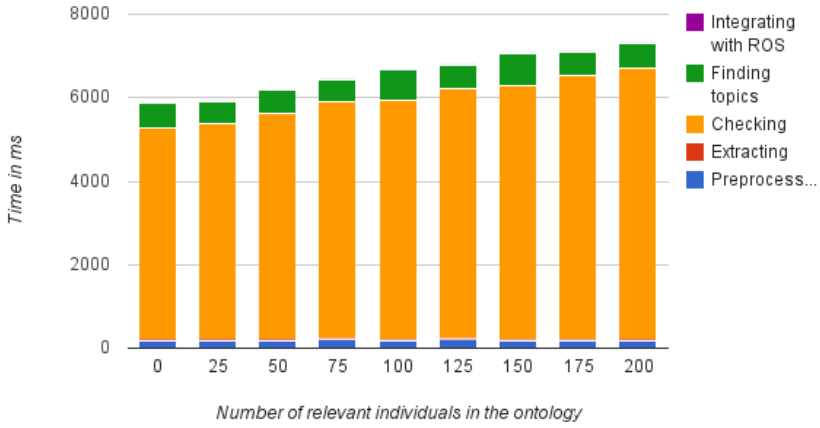


Figure 7.3: Varying number of relevant individuals.

Figure 7.3 shows that the increase in execution time is almost the same as in the previous test. Therefore, we can conclude that the execution of the KM is only influenced by the number of individuals regardless of whether they are relevant or irrelevant.

Test case 2.1 - Varying number of irrelevant topic specifications

The test case uses an ontology with 50 concepts and 50 relevant individuals and the formula from earlier tests. The topic specifications contain 9 relevant topic specifications (one for each feature in the formula) and a varying number of irrelevant topic specifications (from 0 to 500). Irrelevant topic specifications were randomly generated and differ only in one argument from the relevant topic specifications.

Number of irrelevant topic specs	0	100	200	300	400	500
Preprocessing	188	184	183	193	184	185
Extracting features	28	27	29	29	28	28
Checking features	1969	2034	2038	2040	2101	2200
Matching topic specs	585	831	1026	1243	1345	1345
Integrating with ROS	2	3	4	3	3	3
Total	2772	3079	3280	3508	3661	3761

Table 7.4: Varying number of irrelevant topic specifications.

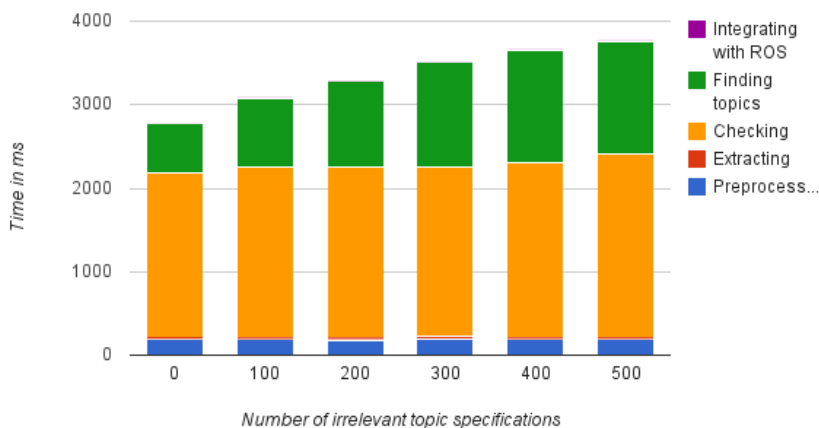


Figure 7.4: Varying number of irrelevant topic specifications.

As expected Table 7.4 shows that the phase of feature checking is not influenced by an increase in the number of topic specifications. The increase is visible in the phase of matching topic specification and amounts to up to 200 ms for every 100 new topic specifications.

Test case 2.2 - Varying number of relevant topic specifications

This test case uses the same setup as test case 2.1 with respect to the ontology and the formula. However, in this case the topic specification contains 9 irrelevant topic specifications and a varying number of relevant topic specifications (0 to 500). In each iteration of the test number of relevant topic specifications is increased by 100.

Number of relevant topic specs	0	100	200	300	400	500
Preprocessing	199	186	186	192	191	191
Extracting features	28	28	30	29	29	29
Checking features	2003	1925	2063	1949	2040	2015
Matching topic specs	574	818	1064	1258	1433	1490
Integrating with ROS	2	4	4	4	6	4
Total	2806	2961	3347	3432	3699	3729

Table 7.5: Varying number of relevant topic specifications.

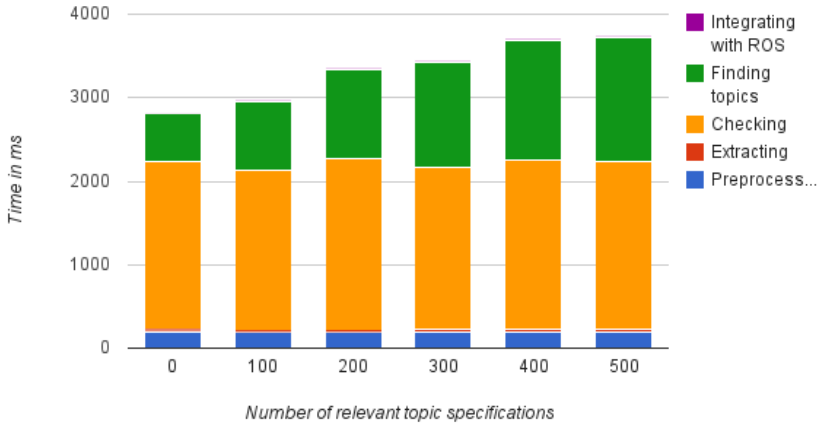


Figure 7.5: Varying number of relevant topic specifications.

The results presented in Table 7.5 show very similar results to the ones presented in Table 7.4 which implies that the process of finding relevant topic specifications is only influenced by an increase in the number of topic specifications and is the same regardless of whether topic specifications are relevant or irrelevant.

Test case 3 - Varying number of features in a formula

In this test case the number of features in the formula in each iteration of the test is increased by 10. The test setup also included 50 topic specifications (one for each feature) and an ontology with 50 concepts and 51 relevant individuals.

Number of features	10	20	30	40	50
Preprocessing	181	181	177	198	178
Extracting features	28	30	31	34	36
Checking features	1929	2090	2355	2746	3089
Matching topic specs	633	1031	1172	1325	1486
Integrating with ROS	2	4	3	3	4
Total	2773	3336	3738	4306	4793

Table 7.6: Varying number of features in a formula.

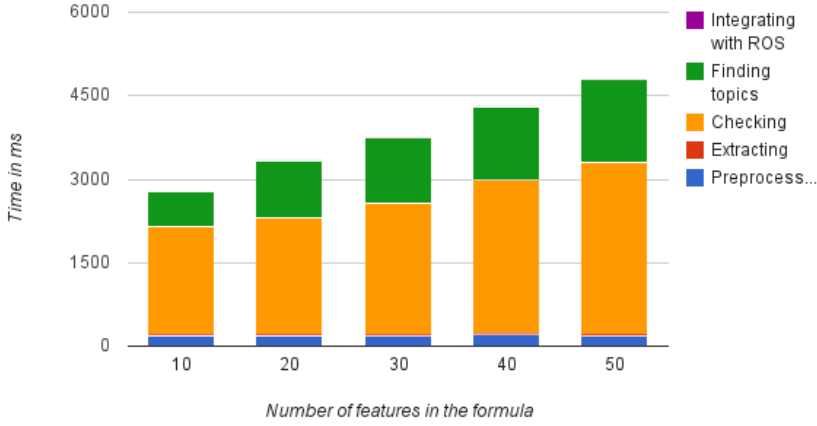


Figure 7.6: Varying number of features in the formula.

Table 7.6 and Figure 7.6 show that the increase in execution time is linear when increasing number of features in a formula. The results also show that both the phase of feature checking and the phase of matching topic specifications are impacted by an increase in the number of features in a formula. This is expected as higher number of features mean that more checks need to be done against the ontology together with additional queries against topic specifications. It was also expected that the phase of feature extraction will be impacted as more features have to be extracted from the formula. However, the increase in extraction time is negligible with respect to the total execution time.

Test case 4 - Quantified or non-quantified arguments

In this test case we compare quantified to non-quantified versions of the same formula. Test case tested the following formulas:

$$\begin{aligned}
 & \text{forall } x \text{ in } B (Z[x, d1, e1]) \\
 & \text{forall } x \text{ in } B, y \text{ in } D (Z[x, y, e1]) \\
 & \text{forall } x \text{ in } B, y \text{ in } D, z \text{ in } E (Z[x, y, z])
 \end{aligned}$$

against their respective expanded versions. Similar to the earlier test cases, the topic specifications contain only one relevant topic for each expanded feature and the total number of topic specifications does not change throughout the test. The ontology used in this test case has 50 concepts and 50 individuals.

	quant 3	non- quant 3	quant 9	non- quant 9	quant 27	non- quant 27
Preprocessing	194	199	187	186	196	183
Extracting features	27	27	28	29	32	31
Checking features	1586	1697	1580	2268	98	3062
Matching topic specs	485	521	693	630	1332	1041
Integrating with ROS	2	2	2	3	3	4
Total	2294	2446	2490	3116	1661	4321

Table 7.7: Comparing quantified and non-quantified versions of a formula.

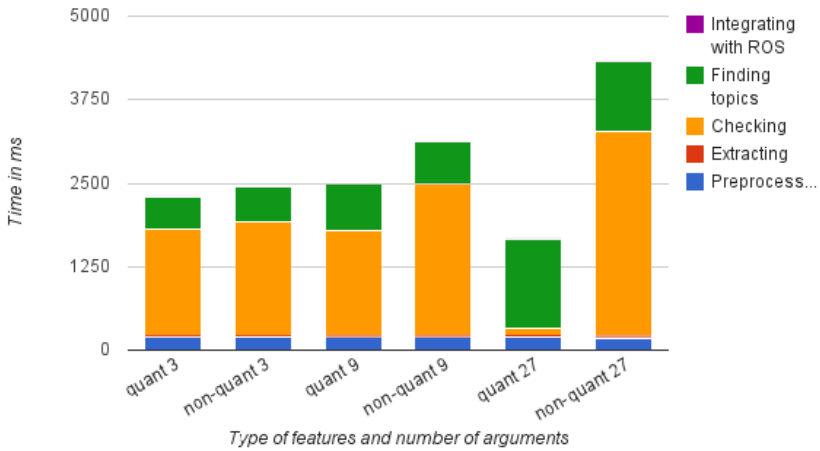


Figure 7.7: Comparing quantified and non-quantified versions of a formula.

Results presented in Figure 7.7 show that formulas with quantified variables require less time for semantic matching. The obvious difference is in the phase of feature checking. However, this is somewhat 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 a relation in the ontology. If one or more arguments of a feature is a sort then the first step is skipped for this argument. This is visible in the test case with 27 quantified features (quant27) where the feature checking phase took only 97 ms while the same phase took around 3 seconds in the test case with 27 non-quantified features. This shows that the queries to acquire arguments' sorts are much more expensive than checking these sorts against a relation in the ontology.

The results also show that the phase of matching topic specifications 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.

7.4 Conclusion

The presented test cases covered different aspects of the semantic matching process. The acquired results were expected and showed that the changes in ontologies have the greatest impact on the phase of feature checking. Similarly, the changes in topic specifications affect the phase of matching topic specifications the most.

The results also show that the phase of feature checking is relatively expensive and has the initial cost of 2 seconds when checking 10 features with object arguments. The problem lies in the fact that we are not using URIs and therefore need to iterate through the whole list of individuals when checking if an object exists in an ontology. However, it was unexpected that an increase with 200 individuals would result in approximately 2 seconds increase in the total execution time. Additional tests were done and they showed that when querying for a list of individuals in the Jena Semantic Web Framework only the first call is expensive while every subsequent call takes less time. For example, in test case 3 with 50 features the first query for a list of individuals takes 1200 ms while every other call takes less than 20 ms. This could imply that Jena Semantic Framework loads the individuals into memory and that this loading is what accounts for the high costs of the feature checking phase.

One way to deal with this problem is to either introduce URIs into topic specifications or to use predefined URIs in the ontology. The first solution would complicate topic specifications as it would require that a user specifies URI for every resource in a topic specification. On the other hand, the second solution would limit the use of distributed heterogeneous ontologies in the system.

However, even with this additional cost the Knowledge Manager performs well and there is only a linear increase in time when increasing number of topic specifications in a system or number of entities in an ontology. It is also important to note that semantic matching is done only once for each formula at subscription time which makes the cost of semantic matching acceptable given its advantages.

Chapter 8

Conclusion

8.1 Summary

Autonomous systems have a number of sensors at their disposal. Data provided by sensors is incrementally available and can be represented in the form of streams. This makes it possible for autonomous systems to run functionalities which do reasoning over incrementally available data. However, in order to use this data for functionalities such as execution monitoring and spatio-temporal reasoning it is necessary to process the data, as data produced by sensors is often noisy and not suitable for reasoning. The processing adapts noisy data from sensors into exact symbolic information needed for reasoning. DyKnow is one example of a knowledge processing middleware which provides the support for bridging the gap between sensing and reasoning.

The reasoning needed for the aforementioned functionalities is based on evaluating logical formulas. To evaluate these formulas it is necessary to provide relevant data for each symbol in the formula. In DyKnow this is done manually by specifying mappings between symbols in a formula and streams. The goal of this Master's thesis was to analyze and provide a solution to the problem of automatic semantic-based matching of symbols in a logical formula to content of streams. The analysis was focused on how the existing semantic technologies could be used in the process of semantic matching. It showed that ontologies can be used for representing machine-readable domain models and therefore can be used for establishing a common vocabulary for users and autonomous systems. The use of ontologies also makes the system domain independent as different ontologies could be used to define different environments and applications.

The analysis also showed that in order to support semantic matching it is necessary to establish a machine-readable encoding of content of streams. Given that in our solution we wanted to implement semantic matching in the ROS implementation of DyKnow this meant that we had to design a rep-

resentation of content of topics. For this purpose the Semantic Specification Language for Topics (SSL_T) was developed which provides the means for specifying content of three different categories of topics, topics containing objects, sorts and features.

The semantic matching in the ROS implementation of DyKnow was implemented in the Knowledge Manager (KM) and tested in two scenarios, the Single Platform and the Multiple Platform scenario. The use case studies show that the KM is able to use data from both homogeneous and heterogeneous distributed sources. To support this interoperability between platforms it is necessary to provide semantic mappings between concepts in ontologies on different platforms. The implementation also includes a reasoning mechanism for ontologies which allows for the discovery of equivalent features not explicitly defined.

Finally, we have also presented a performance evaluation of our implementation. The performance evaluation showed how different aspects of a system (number of entities in an ontology, number of topic specifications, etc.) impact the execution time. An analysis of the test results shows that the fact that we do not use URIs in topic specifications has a significant impact on the execution time as searching for an entity in an ontology requires iterating through a list of all entities. The URIs were not included in our solution as that would mean that a user needs to know and specify an URI for every resource in a topic specification thus making the process of topic specifications more complex. However, the semantic matching is done only once for each formula at subscription time which makes this cost acceptable given the advantages that semantic matching offers.

In conclusion, the solution to the semantic matching problem proposed in this thesis represents an important step towards fully automatized semantic-based stream reasoning. Our solution also shows that semantic technologies can be used for establishing machine-readable domain models. The use of these technologies made the semantic matching domain and platform independent as all domain and platform specific knowledge is specified in ontologies. Semantic matching of ontologies is an active research area and there currently exist a number of mechanisms for mapping concepts in distributed ontologies. In our solution the use of semantic mappings made it possible for systems to support reuse streams on distributed heterogeneous platforms.

8.2 Future work

There are a number of opportunities to improve the current implementation and extend it with new functionalities. Some of them are enumerated in the following list:

- Extending the semantic matching to consider the properties of streams
 - in some cases the choice of a relevant stream for some feature could also be based on the properties of that stream such as maximum delay

and sample period. This would require the extension of SSL_T with support for specifying stream properties.

- Extend the semantic matching to support ontologies which do not conform to our proposed model – as discussed, ontologies used in the semantic matching need to have a separate class hierarchy to enumerate possible features in a formula. However, some existing ontologies might use properties for this purpose. One way to support these ontologies is to make some form of an ontology adapter which would accept ontologies of this form and adapt them to one suitable for the semantic matching.
- Dynamic adaptation to changes in streams – the current implementation is static in the sense that changes in streams (addition of new objects to a sort, addition and removal of streams) will only be included in future calls for semantic matching. Dynamic adaptation to changes in streams could be implemented in the KM. In this case the KM would have to monitor changes in streams and adapt active subscriptions to conform to these changes.
- Improving performance of semantic matching – performance evaluation showed that the fact that topic specifications do not use URIs imposes high costs in the feature checking phase. One way to solve this is to either add URIs to topic specifications or to override the Jena Semantic Web Framework methods for fetching entities in an ontology with methods which do not require URIs.

Bibliography

- [1] G. Antoniou and F. Van Harmelen. *A semantic web primer*, volume 24. The MIT Press, December 2004.
- [2] Y. Arens, C.N. Hsu, and C.A. Knoblock. Query processing in the sims information mediator. *Advanced Planning Technology*, 32:78–93, 1996.
- [3] V.R. Benjamins, J. Contreras, O. Corcho, and A. Gomez-Perez. Six challenges for the semantic web. *KR2002 (ISOCO White Paper)*, 2002.
- [4] A. Borgida and L. Serafini. Distributed description logics: Assimilating information from peer sources. *Journal on Data Semantics*, pages 153–184, 2003.
- [5] M. Botts, G. Percivall, C. Reed, and J. Davidson. OGC® sensor web enablement: Overview and high level architecture. *GeoSensor networks*, pages 175–190, 2008.
- [6] P. Bouquet, F. Giunchiglia, F. Harmelen, L. Serafini, and H. Stuckenschmidt. C-OWL: Contextualizing ontologies. *The SemanticWeb-ISWC 2003*, pages 164–179, 2003.
- [7] P. Bouquet, L. Serafini, and S. Zanolini. Semantic coordination: a new approach and an application. *The SemanticWeb-ISWC 2003*, pages 130–145, 2003.
- [8] 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.
- [9] Ken Conley. Ros introduction. Internet: <http://www.ros.org/wiki/ROS/Introduction/>, July 18, 2011 [August 10, 2011].
- [10] I. Cruz, H. Xiao, and F. Hsu. Peer-to-peer semantic integration of xml and rdf data sources. *Agents and Peer-to-Peer Computing*, pages 108–119, 2005.

- [11] Patrick Doherty and John-Jules Meyer. Towards a delegation framework for aerial robotic mission scenarios. In *11th International Workshop on Cooperative Information Agents, 2007*. Springer, 2007.
- [12] 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.
- [13] B.C. Grau, B. Parsia, and E. Sirin. Working with multiple ontologies on the semantic web. *The Semantic Web-ISWC 2004*, pages 620–634, 2004.
- [14] Harry Halpin. The Semantic Web: The origins of artificial intelligence redux. In *Third International Workshop on the History and Philosophy of Logic, Mathematics, and Computation (HPLMC-04 2005)*, 2005.
- [15] F. Heintz, J. Kvarnström, and P. Doherty. Stream reasoning in DyKnow: A knowledge processing middleware system. In *Proc. 1st Intl Workshop Stream Reasoning*, 2009.
- [16] F. Heintz, J. Kvarnström, and P. Doherty. Stream-Based Reasoning Support for Autonomous Systems. In *Proceeding of ECAI 2010: 19th European Conference on Artificial Intelligence*, pages 183–188, 2010.
- [17] Fredrik Heintz. *DyKnow: a stream-based knowledge processing middleware framework*. Ph.d. thesis, Linköping University, 2009.
- [18] Fredrik Heintz and Patrick Doherty. Federated DyKnow, a Distributed Information Fusion System for Collaborative UAVs. In *Proceedings of the 11th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, 2010.
- [19] I. Horrocks. OWL: A description logic based ontology language. *Logic Programming*, pages 1–4, 2005.
- [20] Ian Horrocks. Ontologies and the Semantic Web. *Communications of the ACM*, 51(12):58, December 2008.
- [21] 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*, pages 69–75, 2002.
- [22] Frank Manola and Eric Miller. RDF Primer. Internet: <http://www.w3.org/TR/2004/REC-rdf-primer-20040210/>, February 10, 2004 [April 20, 2011].
- [23] E. Mena, A. Illarramendi, V. Kashyap, and A.P. Sheth. Observer: An approach for query processing in global information systems based on interoperation across pre-existing ontologies. *Distributed and Parallel Databases*, 8(2):223–271, 2000.

- [24] 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.
- [25] D.J. Russomanno, C. Kothari, and O. Thomas. Building a sensor ontology: A practical approach leveraging iso and ogc models. In *The 2005 International Conference on Artificial Intelligence*, pages 17–18, 2005.
- [26] L. Serafini and A. Tamin. Drago: Distributed reasoning architecture for the semantic web. *The Semantic Web: Research and Applications*, pages 361–376, 2005.
- [27] Luciano Serafini and Andrei Tamin. DRAGO: Distributed reasoning architecture for the semantic web. *The Semantic Web: Research and Applications*, pages 361–376, 2005.
- [28] A. Sheth, C. Henson, and S.S. Sahoo. Semantic sensor web. *IEEE Internet Computing*, pages 78–83, 2008.
- [29] Michael K. Smith, Chris Welty, and Deborah L. McGuinness. OWL Web Ontology Language Guide. Internet: <http://www.w3.org/TR/owl-guide/>, February 10, 2004 [April 20, 2011].
- [30] H. Stuckenschmidt, H. Wache, T. Vögele, and U. Visser. Enabling technologies for interoperability. *Information sharing: Methods and Applications*, pages 35–46, 2000.
- [31] M. Tenorth and M. Beetz. Knowrob - knowledge processing for autonomous personal robots. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 4261–4266. IEEE, 2009.
- [32] 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*, volume 2001, pages 108–117, 2001.
- [33] M. Waibel, M. Beetz, J. Civera, R. D’Andrea, J. Elfring, D. Galvez-Lopez, K. Haussermann, R. Janssen, J.M.M. Montiel, A. Perzylo, B. Schiessle, M. Tenorth, O. Zweigle, and R. van de Molengraft. Roboearth. *Robotics Automation Magazine, IEEE*, 18(2):69–82, 2011.

Appendix A

Acronyms

AIICS Artificial Intelligence and Integrated Computer Systems Division

ANTLR Another Tool for Language Recognition

AST Abstract Syntax Tree

COWL Context OWL

DDL Distributed Description Logics

DF Directory Facilitator

DTD Document Type Definition

IDA Department of Computer and Information Science

KM Knowledge Manager

MITL Metric Interval Temporal Logic

OWL Web Ontology Language

RDF Resource Description Framework

RDFS Resource Description Framework Schema

ROS Robot Operating System

SSL_T Semantic Specification Language for Topics

SWEET Semantic Web for Earth and Environmental Terminology

UAV Unmanned Aerial Vehicle

URI Uniform Resource Identifier

XML Extensible Markup Language

W3C World Wide Web Consortium

WWW World Wide Web

Appendix B

Ontologies

B.1 RDF/XML representation of the ontology for platform 1

Listing B.1: RDF/XML representation of the ontology for platform 1.

```
<?xml version="1.0" ?>

<!DOCTYPE rdf:RDF [
  <!ENTITY ontology "http://www.example.com/ontology#" >
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
  <!ENTITY ont "http://www.co-ode.org/ontologies/ont.owl#" >
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>

<rdf:RDF xmlns="http://www.w3.org/2002/07/owl#"
  xml:base="http://www.w3.org/2002/07/owl"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:ont="http://www.co-ode.org/ontologies/ont.owl#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:ontology="http://www.example.com/ontology#">

  <!--
  //////////////////////////////////////
  //
  // Object Properties
  //
  //////////////////////////////////////
  -->

  <!-- http://www.example.com/ontology#arg1 -->
  <ObjectProperty rdf:about="&ontology;arg1" />

  <!-- http://www.example.com/ontology#arg2 -->
  <ObjectProperty rdf:about="&ontology;arg2" />
```

```

<!-- http://www.example.com/ontology#arg3 -->
<ObjectProperty rdf:about="&ontology;arg3" />

<!--
////////////////////////////////////
//
// Classes
//
////////////////////////////////////
-->

<!-- http://www.co-ode.org/ontologies/ont.owl#GroundSpeed -->
<Class rdf:about="&ont;GroundSpeed">
  <equivalentClass rdf:resource="&ontology;Speed" />
  <rdfs:subClassOf>
    <Class>
      <intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="&ontology;
          UnaryRelation" />
        <Restriction>
          <onProperty rdf:resource="&ontology;arg1" />
          <allValuesFrom rdf:resource="&ontology;UAV" />
        >
      </Restriction>
    </intersectionOf>
  </Class>
</rdfs:subClassOf>
</Class>

<!-- http://www.example.com/ontology#Above -->
<Class rdf:about="&ontology;Above">
  <rdfs:subClassOf>
    <Class>
      <intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="&ontology;
          BinaryRelation" />
        <Restriction>
          <onProperty rdf:resource="&ontology;arg1" />
          <allValuesFrom rdf:resource="&ontology;
            Object" />
        </Restriction>
        <Restriction>
          <onProperty rdf:resource="&ontology;arg2" />
          <allValuesFrom rdf:resource="&ontology;
            Object" />
        </Restriction>
      </intersectionOf>
    </Class>
  </rdfs:subClassOf>
</Class>

<!-- http://www.example.com/ontology#AerialVehicle -->
<Class rdf:about="&ontology;AerialVehicle">
  <rdfs:subClassOf rdf:resource="&ontology;Vehicle" />
</Class>

<!-- http://www.example.com/ontology#Altitude -->
<Class rdf:about="&ontology;Altitude">
  <rdfs:subClassOf>
    <Class>
      <intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="&ontology;
          UnaryRelation" />
        <Restriction>
          <onProperty rdf:resource="&ontology;arg1" />
          <allValuesFrom rdf:resource="&ontology;
            AerialVehicle" />
        </Restriction>
      </intersectionOf>
    </Class>
  </rdfs:subClassOf>
</Class>

```

```

        </Restriction>
      </intersectionOf>
    </Class>

  </rdfs:subClassOf>
</Class>

<!-- http://www.example.com/ontology#Between -->
<Class rdf:about="&ontology;Between">
  <rdfs:subClassOf>
    <Class>
      <intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="&ontology;
          TernaryRelation" />
        <Restriction>
          <onProperty rdf:resource="&ontology;arg1"/>
          <allValuesFrom rdf:resource="&ontology;
            Object" />
        </Restriction>
        <Restriction>
          <onProperty rdf:resource="&ontology;arg2"/>
          <allValuesFrom rdf:resource="&ontology;
            Object" />
        </Restriction>
        <Restriction>
          <onProperty rdf:resource="&ontology;arg3"/>
          <allValuesFrom rdf:resource="&ontology;
            Object" />
        </Restriction>
      </intersectionOf>
    </Class>
  </rdfs:subClassOf>
</Class>

<!-- http://www.example.com/ontology#BinaryRelation -->
<Class rdf:about="&ontology;BinaryRelation">
  <rdfs:subClassOf rdf:resource="&ontology;Relation" />
</Class>

<!-- http://www.example.com/ontology#Car -->
<Class rdf:about="&ontology;Car">
  <rdfs:subClassOf rdf:resource="&ontology;MannedGroundVehicle"
    "/>
</Class>

<!-- http://www.example.com/ontology#Close -->
<Class rdf:about="&ontology;Close">
  <rdfs:subClassOf>
    <Class>
      <intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="&ontology;
          BinaryRelation" />
        <Restriction>
          <onProperty rdf:resource="&ontology;arg1"/>
          <allValuesFrom rdf:resource="&ontology;
            Object" />
        </Restriction>
        <Restriction>
          <onProperty rdf:resource="&ontology;arg2"/>
          <allValuesFrom rdf:resource="&ontology;
            Object" />
        </Restriction>
      </intersectionOf>
    </Class>
  </rdfs:subClassOf>
</Class>

<!-- http://www.example.com/ontology#Color -->

```

```

<Class rdf:about="&ontology;Color">
  <rdfs:subClassOf>
    <Class>
      <intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="&ontology;
          UnaryRelation"/>
        <Restriction>
          <onProperty rdf:resource="&ontology;arg1"/>
          <allValuesFrom rdf:resource="&ontology;
            Vehicle"/>
        </Restriction>
      </intersectionOf>
    </Class>
  </rdfs:subClassOf>
</Class>

<!-- http://www.example.com/ontology#Crossing -->
<Class rdf:about="&ontology;Crossing">
  <rdfs:subClassOf rdf:resource="&ontology;StaticObject"/>
</Class>

<!-- http://www.example.com/ontology#Fast -->
<Class rdf:about="&ontology;Fast">
  <rdfs:subClassOf>
    <Class>
      <intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="&ontology;
          UnaryRelation"/>
        <Restriction>
          <onProperty rdf:resource="&ontology;arg1"/>
          <allValuesFrom rdf:resource="&ontology;
            Object"/>
        </Restriction>
      </intersectionOf>
    </Class>
  </rdfs:subClassOf>
</Class>

<!-- http://www.example.com/ontology#GroundVehicle -->
<Class rdf:about="&ontology;GroundVehicle">
  <rdfs:subClassOf rdf:resource="&ontology;Vehicle"/>
</Class>

<!-- http://www.example.com/ontology#Helicopter -->
<Class rdf:about="&ontology;Helicopter">
  <rdfs:subClassOf rdf:resource="&ontology;
    UnmannedAerialVehicle"/>
</Class>

<!-- http://www.example.com/ontology#House -->
<Class rdf:about="&ontology;House">
  <rdfs:subClassOf rdf:resource="&ontology;StaticObject"/>
</Class>

<!-- http://www.example.com/ontology#In -->
<Class rdf:about="&ontology;In">
  <rdfs:subClassOf>
    <Class>
      <intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="&ontology;
          BinaryRelation"/>
        <Restriction>
          <onProperty rdf:resource="&ontology;arg1"/>
          <allValuesFrom rdf:resource="&ontology;
            Object"/>
        </Restriction>
        <Restriction>
          <onProperty rdf:resource="&ontology;arg2"/>

```



```

        <allValuesFrom rdf:resource="&ontology;
            Object" />
    </Restriction>
</intersectionOf>
</Class>
</rdfs:subClassOf>
</Class>

<!-- http://www.example.com/ontology#MannedAerialVehicle -->
<Class rdf:about="&ontology;MannedAerialVehicle">
    <rdfs:subClassOf rdf:resource="&ontology;AerialVehicle" />
</Class>

<!-- http://www.example.com/ontology#MannedGroundVehicle -->
<Class rdf:about="&ontology;MannedGroundVehicle">
    <rdfs:subClassOf rdf:resource="&ontology;GroundVehicle" />
</Class>

<!-- http://www.example.com/ontology#MovingObject -->
<Class rdf:about="&ontology;MovingObject">
    <rdfs:subClassOf rdf:resource="&ontology;Object" />
</Class>

<!-- http://www.example.com/ontology#Object -->
<Class rdf:about="&ontology;Object" />

<!-- http://www.example.com/ontology#Position -->
<Class rdf:about="&ontology;Position">
    <rdfs:subClassOf>
        <Class>
            <intersectionOf rdf:parseType="Collection">
                <rdf:Description rdf:about="&ontology;
                    UnaryRelation" />
                <Restriction>
                    <onProperty rdf:resource="&ontology;arg1" />
                    <allValuesFrom rdf:resource="&ontology;
                        Object" />
                </Restriction>
            </intersectionOf>
        </Class>
    </rdfs:subClassOf>
</Class>

<!-- http://www.example.com/ontology#RMAX -->
<Class rdf:about="&ontology;RMAX">
    <rdfs:subClassOf rdf:resource="&ontology;
        UnmannedAerialVehicle" />
</Class>

<!-- http://www.example.com/ontology#Relation -->
<Class rdf:about="&ontology;Relation" />

<!-- http://www.example.com/ontology#Road -->
<Class rdf:about="&ontology;Road">
    <rdfs:subClassOf rdf:resource="&ontology;StaticObject" />
</Class>

<!-- http://www.example.com/ontology#Size -->
<Class rdf:about="&ontology;Size">
    <rdfs:subClassOf>
        <Class>
            <intersectionOf rdf:parseType="Collection">
                <rdf:Description rdf:about="&ontology;
                    UnaryRelation" />
                <Restriction>
                    <onProperty rdf:resource="&ontology;arg1" />
                    <allValuesFrom rdf:resource="&ontology;
                        Object" />
                </Restriction>
            </intersectionOf>
        </Class>
    </rdfs:subClassOf>
</Class>
```

```

        </Restriction>
      </intersectionOf>
    </Class>
  </rdfs:subClassOf>
</Class>

<!-- http://www.example.com/ontology#Slow -->
<Class rdf:about="&ontology;Slow">
  <rdfs:subClassOf>
    <Class>
      <intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="&ontology;
          UnaryRelation"/>
        <Restriction>
          <onProperty rdf:resource="&ontology;arg1"/>
          <allValuesFrom rdf:resource="&ontology;
            Object"/>
        </Restriction>
      </intersectionOf>
    </Class>
  </rdfs:subClassOf>
</Class>

<!-- http://www.example.com/ontology#Speed -->
<Class rdf:about="&ontology;Speed">
  <rdfs:subClassOf>
    <Class>
      <intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="&ontology;
          UnaryRelation"/>
        <Restriction>
          <onProperty rdf:resource="&ontology;arg1"/>
          <allValuesFrom rdf:resource="&ontology;
            MovingObject"/>
        </Restriction>
      </intersectionOf>
    </Class>
  </rdfs:subClassOf>
</Class>

<!-- http://www.example.com/ontology#StaticObject -->
<Class rdf:about="&ontology;StaticObject">
  <rdfs:subClassOf rdf:resource="&ontology;Object"/>
</Class>

<!-- http://www.example.com/ontology#TernaryRelation -->
<Class rdf:about="&ontology;TernaryRelation">
  <rdfs:subClassOf rdf:resource="&ontology;Relation"/>
</Class>

<!-- http://www.example.com/ontology#UAV -->
<Class rdf:about="&ontology;UAV">
  <rdfs:subClassOf rdf:resource="&ontology;
    UnmannedAerialVehicle"/>
</Class>

<!-- http://www.example.com/ontology#UGV -->
<Class rdf:about="&ontology;UGV">
  <rdfs:subClassOf rdf:resource="&ontology;
    UnmannedGroundVehicle"/>
</Class>

<!-- http://www.example.com/ontology#UnaryRelation -->
<Class rdf:about="&ontology;UnaryRelation">
  <rdfs:subClassOf rdf:resource="&ontology;Relation"/>
</Class>

<!-- http://www.example.com/ontology#UnmannedAerialVehicle -->

```

```

<Class rdf:about="&ontology;UnmannedAerialVehicle">
  <rdfs:subClassOf rdf:resource="&ontology;AerialVehicle" />
</Class>

<!-- http://www.example.com/ontology#UnmannedGroundVehicle -->
<Class rdf:about="&ontology;UnmannedGroundVehicle">
  <rdfs:subClassOf rdf:resource="&ontology;GroundVehicle" />
</Class>

<!-- http://www.example.com/ontology#Vehicle -->
<Class rdf:about="&ontology;Vehicle">
  <rdfs:subClassOf rdf:resource="&ontology;MovingObject" />
</Class>

<!--
////////////////////////////////////
//
//  Individuals
//
////////////////////////////////////
-->

<!-- http://www.co-ode.org/ontologies/ont.owl#car1 -->
<NamedIndividual rdf:about="&ont;car1">
  <rdf:type rdf:resource="&ontology;Car" />
</NamedIndividual>

<!-- http://www.co-ode.org/ontologies/ont.owl#car2 -->
<NamedIndividual rdf:about="&ont;car2">
  <rdf:type rdf:resource="&ontology;Car" />
</NamedIndividual>

<!-- http://www.co-ode.org/ontologies/ont.owl#car3 -->
<NamedIndividual rdf:about="&ont;car3">
  <rdf:type rdf:resource="&ontology;Car" />
</NamedIndividual>

<!-- http://www.co-ode.org/ontologies/ont.owl#uav1 -->
<NamedIndividual rdf:about="&ont;uav1">
  <rdf:type rdf:resource="&ontology;UAV" />
</NamedIndividual>

<!-- http://www.co-ode.org/ontologies/ont.owl#uav2 -->
<NamedIndividual rdf:about="&ont;uav2">
  <rdf:type rdf:resource="&ontology;UAV" />
</NamedIndividual>
</rdf:RDF>

```

B.2 RDF/XML representation of the ontology for platform 2

Listing B.2: RDF/XML representation of the ontology for platform 2.

```

<?xml version="1.0"?>

<!DOCTYPE rdf:RDF [
  <ENTITY owl "http://www.w3.org/2002/07/owl#" >
  <ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
  <ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
  <ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
  <ENTITY Distributed "http://www.semanticweb.org/ontologies/2011/5/Distributed.owl#" >
]

```

```

<rdf:RDF xmlns="http://www.semanticweb.org/ontologies/2011/5/
Distributed.owl#"
  xml:base="http://www.semanticweb.org/ontologies/2011/5/
Distributed.owl"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:Distributed="http://www.semanticweb.org/ontologies
/2011/5/Distributed.owl#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  <owl:Ontology rdf:about="http://www.semanticweb.org/ontologies
/2011/5/Distributed.owl"/>

<!--
////////////////////////////////////
//
// Object Properties
//
////////////////////////////////////
-->

<!-- http://www.semanticweb.org/ontologies/2011/5/Distributed.
owl#arg1 -->
<owl:ObjectProperty rdf:about="&Distributed;arg1"/>

<!-- http://www.semanticweb.org/ontologies/2011/5/Distributed.
owl#arg2 -->
<owl:ObjectProperty rdf:about="&Distributed;arg2"/>

<!--
////////////////////////////////////
//
// Classes
//
////////////////////////////////////
-->

<!-- http://www.semanticweb.org/ontologies/2011/5/Distributed.
owl#Automobile -->
<owl:Class rdf:about="&Distributed;Automobile">
  <rdfs:subClassOf rdf:resource="&Distributed;GroundVehicle"/>
</owl:Class>

<!-- http://www.semanticweb.org/ontologies/2011/5/Distributed.
owl#GroundVehicle -->
<owl:Class rdf:about="&Distributed;GroundVehicle">
  <rdfs:subClassOf rdf:resource="&Distributed;Vehicle"/>
</owl:Class>

<!-- http://www.semanticweb.org/ontologies/2011/5/Distributed.
owl#Position -->
<owl:Class rdf:about="&Distributed;Position">
  <rdfs:subClassOf>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="&Distributed;
UnaryRelation"/>
        <owl:Restriction>
          <owl:onProperty rdf:resource="&Distributed;
arg1"/>
          <owl:allValuesFrom rdf:resource="&
Distributed;Vehicle"/>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </rdfs:subClassOf>

```

```

</owl:Class>

<!-- http://www.semanticweb.org/ontologies/2011/5/Distributed.
owl#Speed -->
<owl:Class rdf:about="&Distributed;Speed">
  <rdfs:subClassOf>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="&Distributed;
          UnaryRelation"/>
        <owl:Restriction>
          <owl:onProperty rdf:resource="&Distributed;
            arg1"/>
          <owl:allValuesFrom rdf:resource="&
            Distributed;Automobile"/>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </rdfs:subClassOf>
</owl:Class>

<!-- http://www.semanticweb.org/ontologies/2011/5/Distributed.
owl#UnaryRelation -->
<owl:Class rdf:about="&Distributed;UnaryRelation"/>

<!-- http://www.semanticweb.org/ontologies/2011/5/Distributed.
owl#Vehicle -->
<owl:Class rdf:about="&Distributed;Vehicle"/>

<!-- http://www.w3.org/2002/07/owl#Thing -->
<owl:Class rdf:about="&owl;Thing"/>

<!--
////////////////////////////////////
//
//  Individuals
//
////////////////////////////////////
-->

<!-- http://www.semanticweb.org/ontologies/2011/5/Distributed.
owl#car11 -->
<owl:NamedIndividual rdf:about="&Distributed;car11">
  <rdf:type rdf:resource="&Distributed;Automobile"/>
</owl:NamedIndividual>

<!-- http://www.semanticweb.org/ontologies/2011/5/Distributed.
owl#car12 -->
<owl:NamedIndividual rdf:about="&Distributed;car12">
  <rdf:type rdf:resource="&Distributed;Automobile"/>
</owl:NamedIndividual>
</rdf:RDF>

```

B.3 RDF/XML representation of the ontology for platform 3

Listing B.3: RDF/XML representation of the ontology for platform 3.

```
<?xml version="1.0"?>
```

```

<!DOCTYPE rdf:RDF [
  <ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
  <ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >

```

```

<!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
<!ENTITY distributed_uavs "http://www.semanticweb.org/ontologies
/2011/6/distributed_uavs.owl#" >
]>

<rdf:RDF xmlns="http://www.w3.org/2002/07/owl#"
  xml:base="http://www.w3.org/2002/07/owl"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:distributed_uavs="http://www.semanticweb.org/ontologies
/2011/6/distributed_uavs.owl#">
  <Ontology rdf:about="http://www.semanticweb.org/ontologies
/2011/6/distributed_uavs.owl"/>

  <!--
  //////////////////////////////////////
  //
  // Object Properties
  //
  //////////////////////////////////////
  -->

  <!-- http://www.semanticweb.org/ontologies/2011/6/
distributed_uavs.owl#arg1 -->
  <ObjectProperty rdf:about="&distributed_uavs;arg1"/>

  <!-- http://www.semanticweb.org/ontologies/2011/6/
distributed_uavs.owl#arg2 -->
  <ObjectProperty rdf:about="&distributed_uavs;arg2"/>

  <!--
  //////////////////////////////////////
  //
  // Classes
  //
  //////////////////////////////////////
  -->

  <!-- http://www.semanticweb.org/ontologies/2011/6/
distributed_uavs.owl#Aircraft -->
  <Class rdf:about="&distributed_uavs;Aircraft"/>

  <!-- http://www.semanticweb.org/ontologies/2011/6/
distributed_uavs.owl#Alt -->
  <Class rdf:about="&distributed_uavs;Alt">
    <equivalentClass rdf:resource="&distributed_uavs;Height"/>
    <rdfs:subClassOf>
      <Class>
        <intersectionOf rdf:parseType="Collection">
          <rdf:Description rdf:about="&distributed_uavs;
UnaryRelation"/>
          <Restriction>
            <onProperty rdf:resource="&distributed_uavs;
arg1"/>
            <allValuesFrom rdf:resource="&
distributed_uavs;Aircraft"/>
          </Restriction>
        </intersectionOf>
      </Class>
    </rdfs:subClassOf>
  </Class>

  <!-- http://www.semanticweb.org/ontologies/2011/6/
distributed_uavs.owl#BinaryRelation -->
  <Class rdf:about="&distributed_uavs;BinaryRelation"/>

```

```

<!-- http://www.semanticweb.org/ontologies/2011/6/
distributed_uavs.owl#Height -->
<Class rdf:about="distributed_uavs; Height">
  <rdfs:subClassOf>
    <Class>
      <intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="distributed_uavs;
          UnaryRelation"/>
        <Restriction>
          <onProperty rdf:resource="distributed_uavs;
            arg1"/>
          <allValuesFrom rdf:resource="distributed_uavs; Aircraft"/>
        </Restriction>
      </intersectionOf>
    </Class>
  </rdfs:subClassOf>
</Class>

<!-- http://www.semanticweb.org/ontologies/2011/6/
distributed_uavs.owl#MannedAircraftSystem -->
<Class rdf:about="distributed_uavs; MannedAircraftSystem">
  <rdfs:subClassOf rdf:resource="distributed_uavs; Aircraft"/>
</Class>

<!-- http://www.semanticweb.org/ontologies/2011/6/
distributed_uavs.owl#Near -->
<Class rdf:about="distributed_uavs; Near">
  <rdfs:subClassOf>
    <Class>
      <intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="distributed_uavs;
          BinaryRelation"/>
        <Restriction>
          <onProperty rdf:resource="distributed_uavs;
            arg1"/>
          <allValuesFrom rdf:resource="distributed_uavs; Aircraft"/>
        </Restriction>
        <Restriction>
          <onProperty rdf:resource="distributed_uavs;
            arg2"/>
          <allValuesFrom rdf:resource="distributed_uavs; Aircraft"/>
        </Restriction>
      </intersectionOf>
    </Class>
  </rdfs:subClassOf>
</Class>

<!-- http://www.semanticweb.org/ontologies/2011/6/
distributed_uavs.owl#Spd -->
<Class rdf:about="distributed_uavs; Spd">
  <rdfs:subClassOf>
    <Class>
      <intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="distributed_uavs;
          UnaryRelation"/>
        <Restriction>
          <onProperty rdf:resource="distributed_uavs;
            arg1"/>
          <allValuesFrom rdf:resource="distributed_uavs; Aircraft"/>
        </Restriction>
      </intersectionOf>
    </Class>
  </rdfs:subClassOf>

```

```

</Class>

<!-- http://www.semanticweb.org/ontologies/2011/6/
distributed_uavs.owl#UnaryRelation -->
<Class rdf:about="distributed_uavs; UnaryRelation" />

<!-- http://www.semanticweb.org/ontologies/2011/6/
distributed_uavs.owl#UnmannedAircraftSystem -->
<Class rdf:about="distributed_uavs; UnmannedAircraftSystem">
  <rdfs:subClassOf rdf:resource="distributed_uavs; Aircraft" />
</Class>

<!--
////////////////////////////////////
//
//  Individuals
//
////////////////////////////////////
-->

<!-- http://www.semanticweb.org/ontologies/2011/6/
distributed_uavs.owl#heli1 -->
<NamedIndividual rdf:about="distributed_uavs; heli1">
  <rdf:type rdf:resource="distributed_uavs;
    MannedAircraftSystem" />
</NamedIndividual>

<!-- http://www.semanticweb.org/ontologies/2011/6/
distributed_uavs.owl#heli2 -->
<NamedIndividual rdf:about="distributed_uavs; heli2">
  <rdf:type rdf:resource="distributed_uavs;
    MannedAircraftSystem" />
</NamedIndividual>

<!-- http://www.semanticweb.org/ontologies/2011/6/
distributed_uavs.owl#uas20 -->
<NamedIndividual rdf:about="distributed_uavs; uas20">
  <rdf:type rdf:resource="distributed_uavs;
    UnmannedAircraftSystem" />
</NamedIndividual>

<!-- http://www.semanticweb.org/ontologies/2011/6/
distributed_uavs.owl#uas21 -->
<NamedIndividual rdf:about="distributed_uavs; uas21">
  <rdf:type rdf:resource="distributed_uavs;
    UnmannedAircraftSystem" />
</NamedIndividual>

<!-- http://www.semanticweb.org/ontologies/2011/6/
distributed_uavs.owl#uas22 -->
<NamedIndividual rdf:about="distributed_uavs; uas22">
  <rdf:type rdf:resource="distributed_uavs;
    UnmannedAircraftSystem" />
</NamedIndividual>
</rdf:RDF>

```


Appendix C

Topic Specifications

C.1 XML representation of topic specifications for platform 1

Listing C.1: Topic specifications in XML.

```
<Topic_specs>
  <Topic msgtype="UavMsg" name="topic1">
    <feature name="Altitude" value="msg.alt">
      <object name="uav1" value="msg.id"/>
    </feature>
    <feature name="Speed" value="msg.spd">
      <object name="uav1" value="msg.id"/>
    </feature>
  </Topic>
  <Topic msgtype="UavMsg" name="topic2">
    <feature name="Altitude" value="msg.alt">
      <sort all_objects="false" name="UAV" value="msg.id"/>
    </feature>
  </Topic>
  <Topic msgtype="CarMsg" name="topic3">
    <feature name="Speed" value="msg.spd">
      <object name="car1" value="msg.id"/>
    </feature>
  </Topic>
  <Topic msgtype="CarMsg" name="topic4">
    <feature name="Speed" value="msg.spd">
      <object name="car2" value="msg.id"/>
    </feature>
  </Topic>
  <Topic msgtype="CarMsg" name="topic5">
    <feature name="Speed" value="msg.spd">
      <object name="car3" value="msg.id"/>
    </feature>
  </Topic>
  <Topic msgtype="UavMsg" name="topic6">
    <feature name="Close" value="msg.close">
      <object name="uav1" value="msg.id1"/>
      <object name="uav2" value="msg.id2"/>
    </feature>
  </Topic>
  <Topic msgtype="UavMsg" name="topic7">
    <feature name="Close" value="msg.close">
```

```

    <object name="uav2" value="msg.id1" />
    <object name="uav1" value="msg.id2" />
  </feature>
</Topic>
<Topic msgtype="UavMsg" name="topic8">
  <feature name="Close" value="msg.close">
    <object name="uav1" value="msg.id1" />
    <sort all_objects="false" name="UAV" value="msg.id2" />
  </feature>
</Topic>
<Topic msgtype="UavMsg" name="topic9">
  <feature name="GroundSpeed" value="msg.spd">
    <object name="uav2" value="msg.id" />
  </feature>
</Topic>
</Topic-specs>

```

C.2 XML representation of topic specifications for platform 2

Listing C.2: Topic specifications for platform 2 in XML.

```

<Topic-specs>
  <Topic msgtype="AutomobileMsg" name="carstopic1">
    <feature name="Speed" value="msg.speed">
      <sort all_objects="false" name="Automobile" value="msg.id" />
    </feature>
  </Topic>
  <Topic msgtype="AutomobileMsg" name="carstopic2">
    <feature name="Speed" value="msg.speed">
      <object name="car11" value="msg.id" />
    </feature>
  </Topic>
  <Topic msgtype="AutomobileMsg" name="carstopic3">
    <feature name="Position" value="msg.speed">
      <sort all_objects="true" name="Automobile" value="msg.id" />
    </feature>
  </Topic>
</Topic-specs>

```

C.3 XML representation of topic specifications for platform 3


Listing C.3: Topic specifications for platform 3 in XML.

```

<Topic-specs>
  <Topic msgtype="UasMsg" name="uastopic1">
    <feature name="Spd" value="msg.speed">
      <sort all_objects="true" name="UnmannedAircraftSystem" value="msg.id" />
    </feature>
  </Topic>
  <Topic msgtype="UasMsg" name="uastopic2">
    <feature name="Spd" value="msg.speed">
      <object name="uas20" value="msg.id" />
    </feature>
  </Topic>
  <Topic msgtype="UasMsg" name="uastopic3">
    <feature name="Spd" value="msg.speed">

```

```
<object name="uas22" value="msg.id" />
</feature>
</Topic>
<Topic msgtype="UasMsg" name="uastopic4">
  <feature name="Height" value="msg.height">
    <object name="uas20" value="msg.id" />
  </feature>
</Topic>
<Topic msgtype="UasMsg" name="uastopic5">
  <feature name="Alt" value="msg.alt">
    <object name="uas21" value="msg.id" />
  </feature>
</Topic>
<Topic msgtype="UasMsg" name="uastopic6">
  <feature name="Near" value="msg.near">
    <sort all_objects="true" name="UnmannedAircraftSystem" value="msg
      .id1" />
    <object name="uas22" value="msg.id2" />
  </feature>
</Topic>
<Topic msgtype="UasMsg" name="uastopic7">
  <feature name="Near" value="msg.near">
    <sort all_objects="true" name="UnmannedAircraftSystem" value="msg
      .id1" />
    <object name="uas21" value="msg.id2" />
  </feature>
</Topic>
<Topic msgtype="UasMsg" name="uastopic8">
  <feature name="Near" value="msg.near">
    <object name="uas21" value="msg.id2" />
    <sort all_objects="true" name="UnmannedAircraftSystem" value="msg
      .id2" />
  </feature>
</Topic>
</Topic_specs>
```


		Avdelning, Institution Division, Department AIICS, Dept. of Computer and Information Science 581 83 Linköping	Datum Date 2011-10-03
Språk Language <input type="checkbox"/> Svenska/Swedish <input checked="" type="checkbox"/> Engelska/English <input type="checkbox"/> _____	Rapporttyp Report category <input type="checkbox"/> Licentiatavhandling <input checked="" type="checkbox"/> Examensarbete <input type="checkbox"/> C-uppsats <input type="checkbox"/> D-uppsats <input type="checkbox"/> Övrig rapport <input type="checkbox"/> _____	ISBN - ISRN LIU-IDA/LITH-EX-A-11/041-SE Serietitel och serienummer ISSN Title of series, numbering -	
URL fr elektronisk version http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-71669		Linköping Studies in Science and Technology Thesis No. LIU-IDA/LITH-EX-A-11/041-SE	
Titel Title Semantic Matching for Stream Reasoning			
Författare Author Zlatan Dragisic			
Sammanfattning Abstract <p>Autonomous system needs to do a great deal of reasoning during execution in order to provide timely reactions to changes in their environment. Data needed for this reasoning process is often provided through a number of sensors. One approach for this kind of reasoning is evaluation of temporal logical formulas through progression. To evaluate these formulas it is necessary to provide relevant data for each symbol in a formula. Mapping relevant data to symbols in a formula could be done manually, however as systems become more complex it is harder for a designer to explicitly state and maintain this mapping. Therefore, automatic support for mapping data from sensors to symbols would make system more flexible and easier to maintain. DyKnow is a knowledge processing middleware which provides the support for processing data on different levels of abstractions. The output from the processing components in DyKnow is in the form of streams of information. In the case of DyKnow, reasoning over incrementally available data is done by progressing metric temporal logical formulas. A logical formula contains a number of symbols whose values over time must be collected and synchronized in order to determine the truth value of the formula. Mapping symbols in formula to relevant streams is done manually in DyKnow. The purpose of this matching is for each variable to find one or more streams whose content matches the intended meaning of the variable. This thesis analyses and provides a solution to the process of semantic matching. The analysis is mostly focused on how the existing semantic technologies such as ontologies can be used in this process. The thesis also analyses how this process can be used for matching symbols in a formula to content of streams on distributed and heterogeneous platforms. Finally, the thesis presents an implementation in the Robot Operating System (ROS). The implementation is tested in two case studies which cover a scenario where there is only a single platform in a system and a scenario where there are multiple distributed heterogeneous platforms in a system. The conclusions are that the semantic matching represents an important step towards fully automatized semantic-based stream reasoning. Our solution also shows that semantic technologies are suitable for establishing machine-readable domain models. The use of these technologies made the semantic matching domain and platform independent as all domain and platform specific knowledge is specified in ontologies. Moreover, semantic technologies provide support for integration of data from heterogeneous sources which makes it possible for platforms to use streams from distributed sources.</p>			
Nyckelord Keywords Semantic Matching, Stream Reasoning, DyKnow, Semantic Web, OWL, ROS			