


Master's thesis

Extending semantic matching in DyKnow to handle indirectly-available streams

Department of Information and Computing Sciences
Utrecht University

Author:
D.N. de Leng
ICA-3220540

Supervisors:
Prof J-J.Ch. Meyer (UU)
Prof P. Doherty (LiU)
Dr F. Heintz (LiU)

17th November 2013

Abstract

Stream reasoning is incremental reasoning over streams, i.e. sequences of time-stamped values. Our stream reasoning approach uses a metric temporal logic. In order to reason about the physical world, knowledge represented in a system has to originate from physical sensors, often providing noisy and incomplete quantitative data. There exists a sense-reasoning gap between this low-level sensor data and the symbolic knowledge necessary for stream reasoning. DyKnow can be used for bridging this gap. In order to find streams based on their semantics, DyKnow employs semantic web technologies that enable it to do semantic matching.

This thesis focuses on indirectly-available information, and seeks to make available this information as streams through the use of transformations in DyKnow. Because the set of available streams may be time-dependent, the availability of desired information changes over time. By utilising transformations, such a dynamic set of available streams may be better handled. To these ends, this thesis further extends DyKnow and its implementation in ROS to support transformations by looking at knowledge processes, an improved semantic matching procedure, and improvements to the declarative languages used to specify DyKnow instances.

Acknowledgements

First and foremost I want to thank Fredrik for his daily supervision, which included reading this thesis time and time again for feedback. I realise you have a busy schedule, yet you made time in order to provide feedback and have discussions on DyKnow, of which I hope many more will follow. You have been a wonderful supervisor for my Master's thesis, and I appreciate the trust you have in my ability to perform well.

I also want to thank both Fredrik and Prof Doherty for allowing me to attend the FUSION 2013 conference to present a paper [21] that covered intermediate results made in the pursuit of this thesis. This was a very educational experience for me, both from an academic and a travel experience considering the ongoing protests in Istanbul at the time of the conference. Presenting is one of my weak points, so taking this step was of great importance to me.

I am grateful for the help from Prof Meyer towards getting me in contact with AIICS and for his supervision during my thesis period. Your support helped make it possible for me to contribute here at AIICS, which is something I have been wanting to do since my freshman year, as evidenced by the number of queue points I had accumulated for an apartment by the time I was able to go.

Additionally I want to thank my parents, my 'tree friends', and Riley for the support in these past 10 months of living in Sweden. It can be difficult to be so far away, for everyone involved. Emigration is something that puts a huge burden on those you leave behind, so I am grateful for you guys putting up with me and helping me out so much from over there. You may not understand what is written in this thesis, but you helped getting it there. *Ontzettend bedankt!*

Lastly, big thanks to Marco for showing me around the university during the first part of my thesis, Marc and Karin for helping me with a bunch of questions in the lead-up to moving to Sweden, Tiemen and Chananja for helping me out with my student room in Utrecht during my absence, as well as Sam and Richard for having me over for so long.

Contents

1 Introduction

This chapter gives an introduction to the problem of indirectly-available information and introduces the stream-based middleware DyKnow. The objectives of this thesis as well as its contributions are covered. Finally, a thesis outline is given, introducing the upcoming chapters and their respective topics.

1.1 Introduction

Advances in the field of autonomous (aerial) systems can be seen in recent achievements both in civilian and military applications. Examples include the now concluded Wallenberg Laboratory for Information Technology and Autonomous Systems[1] (WITAS) project at Linköping University with the goal of producing high-level autonomy in unmanned aerial vehicles (UAVs), and the U.S. Navy Northrop Grumman X-47B prototype for autonomous carrier operations. Figure 1.1 shows a modified Yamaha RMAX used in the WITAS project by the Artificial Intelligence and Integrated Computer Systems (AIICS) division. The ongoing EU Smart Collaboration between Humans and Ground-Aerial Robots for Improving Rescuing Activities in Alpine Environments[2] (SHERPA) project involving multiple partners including seven universities shows continued interest in the field. These contemporary autonomous systems rely increasingly on sensory information and low-level fusion techniques to generate a representation of the environment they reside in. The semantic interpretation of this model in combination with high-level reasoning support makes it possible for these intelligent agents to operate within the environment by for example formulating plans to achieve certain goals or reasoning about the effects of actions on the environment.

[1] `http://www.ida.liu.se/ext/witas/`
[2] `http://www.sherpa-project.eu`

Figure 1.1: One of two Yamaha RMAX UAVs used at Linköping University

1.2 Problem overview

We want to handle the general case of supporting the semantic interpretation of a world model in combination with high-level reasoning support. This leads us to numerous interesting problem scenarios that need to be handled. One such case is considered in this section and serves as the introduction to the problems central to this thesis.

Consider a scenario where we want to use autonomous systems for monitoring traffic. We are interested in detecting traffic violations, and we use heterogeneous autonomous UAVs to achieve this. Some of these UAVs may be controlled by third parties, but a common language defined by an ontology exists between the UAVs. Because the environment changes over time, we consider a snapshot where we have one UAV that is sharing information regarding the speed of cars in kilometres per hour, as well as other potentially relevant information. The other UAVs in this heterogeneous group only share the coordinates of (generic) vehicles in the environment. In this snapshot, it is possible for the group of UAVs to verify whether specific constraints on speed are met by the cars. For example, a mission operator may want to set an upper and lower bound on the speeds depending on the local traffic laws. These constraints can be described using a metric temporal logic, which is covered in the next chapter.

Moving forward in time, we take another snapshot where the UAV providing speeds of cars has left the group. Only the UAVs providing coordinates of vehicles remain. This poses a problem for the evaluation of the formulas, since there exists no flow of information regarding the speeds of

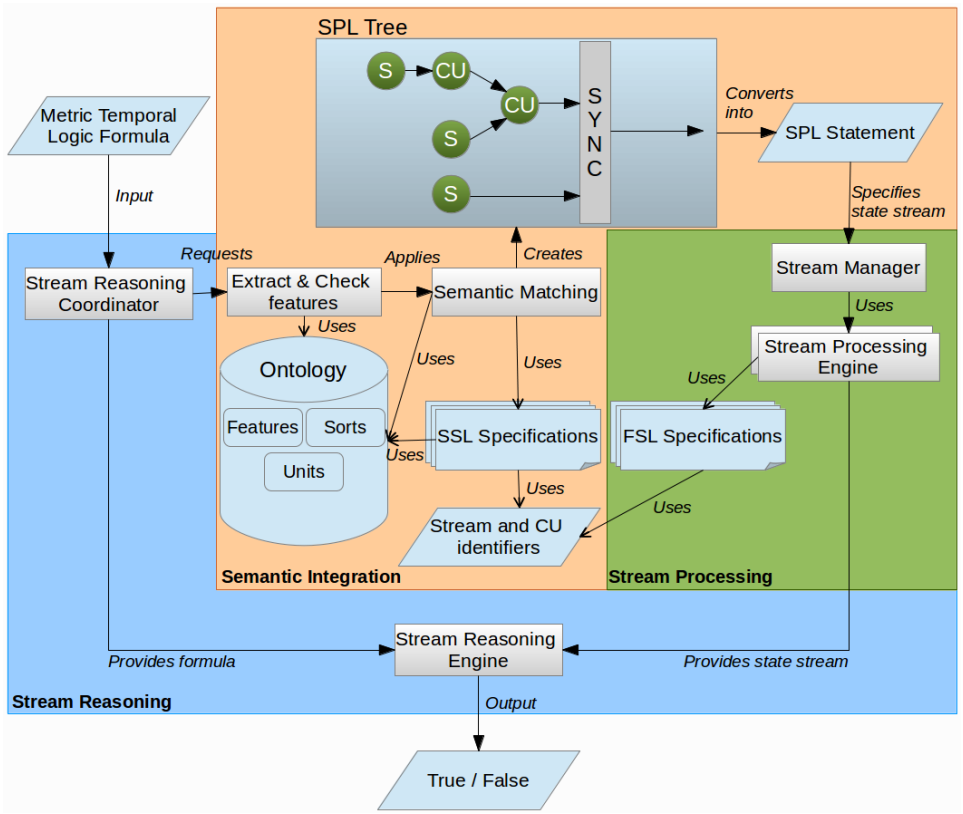

Figure 1.2: DyKnow system flow overiew

cars. However, one of the UAVs has the ability to take the coordinates of vehicles and compute the speeds of these vehicles. This means that the desired information is indirectly available, and can be made directly available through this ability.

Assuming an autonomous system has knowledge of its abilities with regards to this *transformation* of information from one type to another, it is able to automatically generate indirectly-available information as described above. This is the subject of interest in this thesis.

DyKnow[3] is a stream-based knowledge processing middleware framework [22, 19, 25]. It maintains streams of information that may be generated by sensors or various computational processes. *Streams* may be conceptually considered as sequences of abstract values for specified time-points. Given some temporal logic formula, DyKnow can be used to synchronise streams containing the relevant information necessary for evaluating the formula through progression. This makes DyKnow very useful in situations such as

[3]Initially DyKnow was an acronym for *Dynamic Knowledge Processing*. This was later extended to *Dynamic Knowledge Processing and Object Management*. Presently, DyKnow is a pseudo-acronym.

described in the above scenario. An overview of the evaluation of a metric temporal logic formula is shown in Figure 1.2. Three components can be distinguished; the stream reasoning component, the semantic integration component, and the stream processing component. The details of the inner workings of DyKnow will become clear in future chapters, but an intuitive introduction is provided here.

Suppose we have a constraint represented as a metric temporal logic formula. This formula is fed as input to DyKnow's stream reasoning coordinator, which forwards it to the stream reasoning engine for evaluation. Because the stream reasoning engine needs to receive the information necessary to evaluate the formula, it is also sent to the semantic integration component. Here the symbols in the formula are matched against the available streams, both directly and indirectly available through transformations of existing streams. By using semantic technologies, it is possible to find specific information using a common language described by an ontology [24]. The matching procedure uses semantic specifications provided as inputs, describing streams and transformations by the information they provide. The semantic integration component provides DyKnow with identifiers for the streams that are relevant to the evaluation of the formula, which are then used to generate a stream specification tree. Because information in streams is time-dependent, the stream processing component synchronises the relevant streams. The result is provided to the stream reasoning engine. Now that this engine has both the formula and the necessary information for evaluation, the evaluation process is started, returning either 'true' or 'false'. If the formula for example states that cars are not allowed to go faster than 120 km/h, evaluation of the formula to 'false' corresponds to a speed violation detection that can be reacted to.

1.3 Objectives

The main research question of this thesis is: "How can DyKnow efficiently be extended to handle the semantic matching of indirectly-available streams?" As demonstrated in the example, transformations may be used to this effect. In order to answer this question, two goals have been identified.

The first goal of this thesis is to extend semantic matching in DyKnow to handle indirectly-available information. In order to achieve this goal, we consider the following subgoals:

- Supporting transformations between different *units of measurement* and thereby removing the assumption that the unit of measurement for a given feature is always the same. In this context, a feature says something about an object in the world, e.g. speed. This allows DyKnow to better support third-party sources by making units of

measurement explicit, essentially through providing semantics for the numeric values in streams.

- Supporting transformations between different *features* and thereby allowing DyKnow to indirectly obtain features from streams by automatically constructing those streams. This expands the number of features available to the system during semantic matching, reducing the likelihood of ending up with no streams for some feature.

Since the application area of DyKnow is that of time-sensitive autonomous systems such as UAVs, we seek to achieve high efficiency in formula evaluation. The second goal of this thesis is to evaluate the performance and scalability of DyKnow. Since this thesis focuses mainly on the semantic matching support, this will also be the focus during experimentation. The acceptability of DyKnow's performance for an autonomous system depends on the purpose of that autonomous system. However, by providing performance results, a general overview is given.

1.4 Contributions

Work towards this thesis resulted in a publication at the FUSION 2013 conference [21] and a poster presentation at the CADICS 2013 workshop[4]. The conference paper showcases earlier results in the pursuit of the objectives stated above. Besides the publication and poster presentation, the contributions by this thesis are two-fold.

At a theoretical level, this thesis extends earlier work on semantic information integration for stream reasoning [24] by taking into account indirectly-available streams. This led to numerous necessary improvements in other areas of DyKnow. A different perspective on DyKnow is given by representing it as a model. This makes it possible both to extend the declarative languages used by DyKnow to support transformations, and to formally define their semantics with transition rules. Concretely, SPL was extended to handle computational units and nesting, SSL was extended to support the semantic annotation of computational units and sources, and FSL was newly introduced to specify external connections to implementations of computational units and sources. The bulk of the theoretical work has gone into supporting transformations in DyKnow, where transformations between units of measurement and between features are considered.

At an implementation and experimentation level, work on this thesis resulted in the first integrated version of DyKnow-ROS, described in more detail later. Previous Master's theses [13, 34, 27] resulted in the indiviual development of DyKnow implementation components that provided limited

[4] *Semantic Matching for Stream Reasoning* (Heintz and de Leng, 2013) poster available at the CADICS website: `http://cadics.isy.liu.se/`

foundation. These components had to be harmonised, fixed, and often extended in order to get to the stage where work could be done on integrating transformations. The resulting system is the first integrated version of DyKnow-ROS that can be used for empirical experimentation.

1.5 Thesis outline

The thesis is structured as follows.

- Chapter 2 gives an overview of background information on previous work, stream reasoning, and semantic web technologies used.
- Chapter 3 discusses DyKnow both as a model and an implementation. For the implementation, a critical analysis is provided of the work done by previous students.
- Chapter 4 covers the semantic matching approach and the contributions in this area by this thesis.
- Chapter 5 presents the performance evaluation of DyKnow and analyses the results.
- Finally, Chapter 6 draws the conclusions and discusses potential future work.

2 Background

In this chapter, we look at previous work on DyKnow. An introduction is given to stream reasoning, followed by a look at the Web Ontology Language, its basis in Description Logics, and how we can use available semantic technologies to benefit stream reasoning. Finally, semantic technologies are discussed. Upon completion of this chapter, the reader has a general understanding of semantic technologies, including OWL ontologies. This serves as a preparation for later chapters, in which ontologies play a key role.

2.1 Previous work on DyKnow

DyKnow is a stream-based middleware for knowledge processing, and can be used to bridge the sense-reasoning gap [25] that exists between symbolic and subsymbolic techniques in artificial intelligence (AI). It allows for reasoning at different levels of abstraction, and the interaction between these layers. This interaction makes it possible not just to move information upwards from raw signals to logical symbols, but it also makes it possible for high-level reasoning to affect low-level processes by e.g. shrinking the search space. DyKnow was initially proposed by Heintz and Doherty [22] and implemented using the Common Object Request Broker Architecture (CORBA), which will be referred to as DyKnow-CORBA. A formal model of DyKnow and its Knowledge Processing Language (KPL) is presented by Heintz et al [25]. Heintz and Doherty [23] proposed a mechanism for distributed information fusion that allows DyKnow to operate in a multi-agent system.

Recent developments as the result of three Master's theses reimplemented DyKnow in the Robot Operating System (ROS). The ROS implementation will be referred to as DyKnow-ROS. Dragisic [13] introduced semantic matching in order to tackle the problem of semantic information integration for stream reasoning. As part of his thesis work, he developed a *Knowledge Manager* for DyKnow integrated in ROS with the task of carrying

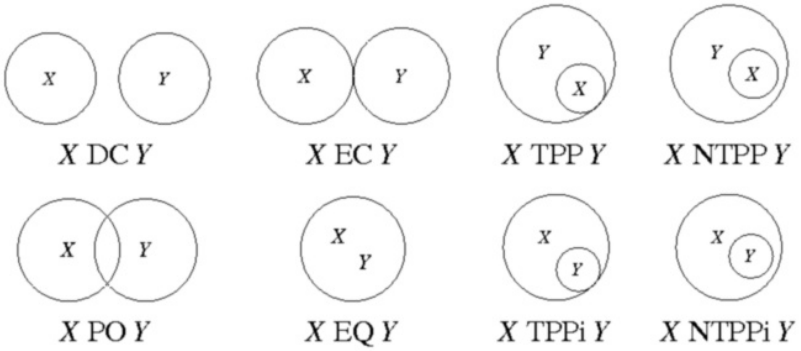

Figure 2.1: RCC8 relations

out semantic matching given a temporal logic formula, where ROS topics were annotated with the Semantic Specification Language for Topics (SSL_T). Additionally, a method for combining different ontologies in order to create a common language is considered, which is extremely useful in situations where multiple instances of DyKnow exist, as it supports federations as proposed by Heintz and Doherty [23]. The resulting work was presented in a conference paper by Heintz and Dragisic [24]. Lazarovski [34] extended the metric temporal logic used by DyKnow's *Stream Reasoning Engine* by incorporating *spatial reasoning* and integrated this in ROS. The resulting spatio-temporal reasoning support relied on *region connection calculus* (RCC8), which consists of eight relations between regions in Euclidian space, as shown in Figure 2.1. Finally, Hongslo [27] used the work by Heintz [19] on DyKnow-CORBA to create stream processing support for DyKnow-ROS. This resulted in the introduction of the Stream Processing Language (SPL), which is an SQL-inspired language used for stream processing purposes. This thesis is based on these recent developments, in particular the work by Heintz and Dragisic [24], which relies on *Semantic Web* technologies.

Early results from this thesis work can be found in [21], and are expanded upon in this thesis. Lastly, Heintz [20] provides an architectural overview of DyKnow-ROS that was obtained in part due to progress made within the scope of this thesis.

2.2 Related work

The term Semantic Sensor Web (SSW) was coined by Sheth et al. [47], based on the Sematic Web vision of Berners-Lee et al. [7] that resulted in subsequent Semantic Web technologies. Its purpose was to address the problem of "too much data and not enough knowledge," by annotating sensor data. This ties into the problem of Situation Awareness (SA) as modelled by End-

sley [16] and served as an inspiration for ontology-based SA approaches by Matheus et al. [36] and Kokar et al. [33] for situation awareness in computer programs. These problems are the subject of study in the area of *information fusion*. There are numerous approaches to SSW ontologies [12], but they can generally be categorised as coming from a sensor perspective or from an observation perspective. Shi et al. [49] propose a framework that can automatically update and extend a sensor ontology, but succeed only partially in doing so.

The SSW has been the subject of numerous research projects similar to DyKnow. Bröring et al. [10] identify challenges to achieving sensor plug & play, and state requirements and propose a method to achieve plug & play functionality [11], inspired by technologies such as USB and IEEE 1451. This is in contrast with the approach taken by Whitehouse et al. [53], who argue that fixed as opposed to ad-hoc sensor networks may be more common in practice. The implementation by Bröring et al. [11] makes use of a Sensor Bus, which is used to mediate between sensors and services, utilising semantic annotations for matchmaking and allowing for unit conversion. This functionality makes the Sensor Bus similar to information-providing services [37], which are a subset of semantic web services. Furthermore, parts of the proposed matchmaking functionality lean strongly towards well-studied [42, 15, 39] web service composition problems. Bröring et al. [11] also take into account the potential of utilising Quality of Service (QoS) in order to break away from boolean matchmaking, which is similar to techniques proposed for web service discovery such as that proposed by Ran [41] or web service composition utilising QoS as proposed by Zeng et al [54]. Sheth and Perry [48] argue that "current tools for managing Semantic Web data must be extended to better handle spatial and temporal data." The approach by Bröring et al. [11] appears to be a step into that direction for the SSW, as they incorporate these elements. This makes the concept of the Sensor Bus similar to DyKnow in its handling of information streams.

2.3 Stream reasoning

Stream reasoning is the incremental reasoning over incrementally available information, which allows for reasoning with minimal latency, making it possible to react to a rapidly changing environment. One technique used for stream reasoning is progression of metric temporal logic formulas. Metric temporal logic is an extension of first order logic with temporal operators *always*, *eventually* and *until* [20]. Informally, $\Box_{[\tau_1,\tau_2]}\phi$ holds at time τ iff it is the case that ϕ holds for all $\tau' \in [\tau+\tau_1, \tau+\tau_2]$, $\Diamond_{[\tau_1,\tau_2]}\phi$ holds at τ iff ϕ holds at some $\tau' \in [\tau+\tau_1, \tau+\tau_2]$, and $\phi \cup_{[\tau_1,\tau_2]} \psi$ holds at τ iff ψ holds at some $\tau' \in [\tau+\tau_1, \tau+\tau_2]$ and ϕ holds in all states in domain $[\tau, \tau']$ respectively. Using this logic, we can describe rules such as "it must always be the case

that uav1 has a distance of at least 10 in the xy-plane from all other UAVs" as such;

$$\forall x \in \mathsf{UAV} : x \neq \mathsf{uav1} \rightarrow \Box_{[0,\infty]}\mathsf{XYDist}[x, \mathsf{uav1}] > 10.$$

In the remainder of the text, $\Box_{[0,\infty]}$ and $\Diamond_{[0,\infty]}$ are abbreviated to $\Box$ and $\Diamond$ respectively.

Progression is a technique that – upon arrival of a new sample – partially evaluates a temporal logic formula and generates a new temporal logic formula based on the information contained in the sample. As an example, take the above formula and assume we receive one sample every 100ms. Upon receiving the first sample, progression yields

$$\begin{aligned}\forall x \in \mathsf{UAV} : x \neq \mathsf{uav1} \rightarrow (&\Box_{[0,100]}\ \mathsf{XYDist}[x, \mathsf{uav1}] > 10 \\ &\wedge\ \Box_{[100,\infty]}\ \mathsf{XYDist}[x, \mathsf{uav1}] > 10),\end{aligned}$$

where the first part of the conjunction can be evaluated using the received sample. In this example, if the subformula for $\tau \in [0, 100]$ holds, the formula is evaluated further for $\tau \in [100, \infty]$. However, if it does not hold, the reasoner can evaluate the entire formula to false without considering $\tau \in [100, \infty]$. The progression technique allows for partial evaluation of temporal logic formulas by utilising information as it arrives, especially when the truth value can be determined when only part of the information associated with a formula is actually available. This results in a low latency, which makes it possible for a system to reason about and react to rapid changes in an environment. This is especially important for programs running on time-sensitive robotic systems such as UAVs, as a slow reaction speed on such platforms may have disastrous consequences. As an example, consider a UAV encountering a bird crossing its flight path.

In order to apply progression, it is necessary to attach meaning to the symbols used in temporal logic formulas. One approach to attach such meaning to symbols is by using streams containing incrementally available values associated with symbols. Because low-level sensory information can be seen as sequences of values incrementally created at run-time, it is natural to model these sequences as streams of time-stamped samples. The source of such streams need not necessarily be physical sensors, as streams can be produced through e.g. the usage of *refinement processes* leveraging fusion techniques.

While stream reasoning using temporal logic assumes exact and unambiguous high-level information on the state of the world, sensors often provide low-level and fuzzy quantitative information. The *sense-reasoning gap* can be partially defined by this discrepancy [25]. In order to bridge this gap, there needs to be a way to construct "suitable representations of the information that can be extracted from the environment using sensors and other available sources, processing the information to generate information

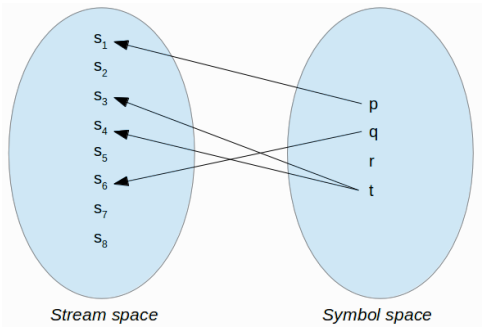

Figure 2.2: Example stream mapping

at higher levels of abstraction, and continuously maintaining a correlation between generated representations and the environment itself." [25]

One approach towards bridging this gap is to create and maintain over time some mapping from symbols in logical formulas to streams in the stream space, as illustrated in Figure 2.2. This can be considered an instance of the *symbol grounding problem*. DyKnow can be used as a tool for bridging the gap and seeks to tackle the symbol grounding problem in order to achieve this.

2.4 Formal semantic grounding with OWL

The semantic web [7] was conceived as a way to make the web understandable by machines. Most information published on the web is meant for human consumption, and therefore uses natural language and rich media such as images and video to present the user with information. While this is suitable for humans, machines have a much more difficult time traversing the web. A naive approach would be to apply tags to the information provided on the web, but as has been pointed out [30], this moves the problem from interpreting the information to interpreting the tags.

The Resource Description Framework (RDF) seeks to describe web resources, as well as the relations between different web resources [30] using infix notation. It makes use of Internationalised Resource Identifiers (IRIs), which are generalisations of Uniform Resource Locators (URLs). The RDF data model [5] consists of statements in the form of triples containing a subject, a predicate and an object. Objects can either be resources or values. As an example, observe Figure 2.3 where an RDF data model is presented as a labeled directed graph. In this example, two triples are presented. The first describes the relationship between the two UAVs, UAV1 and UAV2:

```
<http://www.ida.liu.se/dyknow-example/UAV1>
<http://www.ida.liu.se/dyknow-example/hasFriend>
<http://www.ida.liu.se/dyknow-example/UAV2> .
```

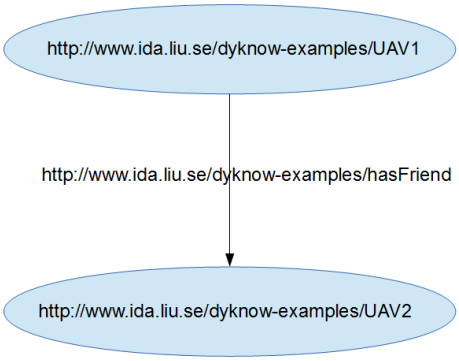

Figure 2.3: Example labeled directed graph

Here, both UAVs are resources and `hasFriend` is a predicate, i.e. UAV2 is deemed trustworthy to UAV1: `hasFriend(UAV1, UAV2)`. As an alternative, observe the following relationship:

```
<http://www.ida.liu.se/dyknow-example/UAV1>
<http://www.ida.liu.se/dyknow-example/hasColour>
"FF0000" .
```

In this example, the relationship states that UAV1 has the colour red.

In the preceding example, it is clear that the triple notation in full results in a lot of text due to the long URIs involved. Thankfully, RDF allows for namespace URIs assigned to prefixes, called a qualified name (QName) simplification [35], which simplifies the last example to:

```
dyknow:UAV1 dyknow:hasColour "FF0000" .
```

RDF can be extended with the RDF Vocabulary Description Language (RDF Schema) [9] which allows the user to further describe resources in terms of classes. These additional features can be used for defining ontologies [30]. With the introduction of RDF Schema, we can use the following special terms:

- `rdfs:subClassOf` declares that the subject `rdfs:Class` is a subclass of the object `rdfs:Class`,
- `rdfs:subPropertyOf` declares that the subject `rdfs:Property` is a subclass of the object `rdfs:Property`,
- `rdfs:domain` declares a restriction on the subjects of the subject `rdfs:Property`, where the restriction states that the objects of the subject `rdfs:Property` must be of the object `rdfs:Class`,
- `rdfs:range` is analogous to `rdfs:domain` but puts a restriction on the subjects of the subject `rdfs:Property`,

- `rdf:type` declares that the subject is an individual of the object `rdfs:Class`.

As an example, take the following statements in Qname notation:

```
dyknow:UAV rdfs:subClassOf dyknow:AutonomousVehicle .
dyknow:UAV1 rdf:type dyknow:UAV .
dyknow:UAV2 rdf:type dyknow:UAV .
dyknow:isFriend rdfs:domain dyknow:AutonomousVehicle .
dyknow:isFriend rdfs:range dyknow:AutonomousVehicle .
dyknow:isWingman rdfs:subPropertyOf dyknow:isFriend .
dyknow:UAV1 dyknow:isFriend dyknow:UAV2 .
```

In this example we have two instances of type `UAV`, such that `UAV1` is the friend of `UAV2`. The `isWingman` property inherits its domain and range requirements from its superproperty `isFriend`, which state that both the domain and range must be of type `AutonomousVehicle`. Because `UAV` is a subclass of `AutonomousVehicle`, the statements are consistent.

However, the expressivity of RDF remains limited. As an example, consider the `isFriend` property. Perhaps we want to enforce this property to be bidirectional, i.e. that `UAV2 isFriend UAV1`, but this is not possible in RDF.

The Web Ontology Language (OWL) [26] is the result of a World Wide Web Consortium (W3C) working group, which was established as a result of the need for a more expressive ontology language [30] that would tackle some of the shortcomings of RDF. OWL builds on RDF and RDF Schema, and allows for an RDF based syntax called RDF/XML that makes it easily readable for web applications [30]. For example, consider the class `UAV` from earlier:

```
<owl:Class rdf:ID="UAV">
   <rdfs:subClassOf rdf:resource="#AutonomousVehicle"/>
</owl:Class>
```

Alternative, more human-readable syntaxes exist, such as the Terse RDF Triple Language (Turtle) syntax [6] for RDF in general, or Manchester syntax [29] for OWL. The latter transforms the RDF/XML syntax into the following:

```
Class: UAV
   SubClassOf: AutonomousVehicle
```

This is a much easier to read syntax, and is therefore used from this point on for this thesis where applicable.

The aforementioned W3C working group was interested in extending RDF Schema, but this desire gave rise to a conflict between expressive power

and efficient reasoning [5]. Because the total set of requirements was incompatible, the decision was made to introduce three sublanguages for OWL, where each sublanguage was geared towards a specific subset of the incompatible requirements.

- *OWL Full* uses all of the available language primitives, and is a true extension to RDF. However, this comes at the cost of losing decidability.
- *OWL DL* is a sublanguage of OWL Full that is geared towards computational efficiency by limiting the allowed expressivity. In contrast to OWL Full, OWL DL retains decidability.
- *OWL Lite* is a sublanguage of OWL DL that even further limits the expressivity, with the advantage of making it simpler to comprehend and implement.

OWL is based in Description Logics (DLs), which can be seen as decidable fragments of first-order logics [30]. A domain consists of *concepts*, *individuals* and *roles*, where concepts and roles are equivalent to classes and properties in RDF, respectively. Individuals can be grouped into concepts, and roles describe relationships between individuals. Currently, two versions of OWL exist. OWL1 DL, simply referred to as OWL DL is based on the DL $\mathcal{SHOIN}(\mathbf{D})$. More recently, OWL2 was developed by the W3C and is based on the DL $\mathcal{SROIQ}(\mathbf{D})$, which is an extension of $\mathcal{SHOIN}(\mathbf{D})$ [31]. These DLs are semantically equivalent to their corresponding versions of OWL.

The DL $\mathcal{SHOIN}(\mathbf{D})$ for OWL is an acronym describing the features of the language [30]. The $\mathcal{S}$ denotes the Booleans and ($\wedge$), or ($\vee$) and not ($\neg$), as well as existential ($\exists$) and universal ($\forall$) quantification, available to class constructors. In OWL, these are described with *intersectionOf*, *unionOf*, *complementOf*, *someValuesFrom* and *allValuesFrom* respectively. Additionally, one can declare properties to adhere to transitivity. The $\mathcal{H}$ denotes property hierarchies, the $\mathcal{O}$ denotes the ability to define classes as collections of individuals, the $\mathcal{I}$ denotes the ability to specify inverse properties, and the $\mathcal{N}$ denotes the ability to define cardinality restrictions. Lastly, the added ($\mathbf{D}$) is a reference to XML Schema datatypes and values support discussed earlier.

The DL $\mathcal{SHOIQ}$ is an extension to the DL $\mathcal{SHOIN}$ with qualified number restrictions [31], which can be used to define a class of individuals for which every individual adheres to a restricted number of a given role, e.g. the class of UAVs for which every individual has at least one friend can be written as $(\geq 1\ hasFriend.UAV)$. The $\mathcal{Q}$ therefore denotes this addition to the $\mathcal{N}$ features discussed earlier.

The DL $\mathcal{SROIQ}$ further extends the DL $\mathcal{SHOIQ}$, and can itself be extended to $\mathcal{SROIQ}(\mathbf{D})$ [31]. It adds an RBox to the already existing TBox

and ABox. TBox statements roughly describe concepts, whereas ABox statements describe individuals. The newly added RBox statements concern roles, and the combination of the TBox, ABox and RBox makes up the knowledge base (KB). The DL $\mathcal{SROIQ}$ therefore not surprisingly adds support for disjoint roles, reflexive and irreflexive roles, negated role assertions, complex role inclusion axioms, universal role U and local reflexivity on roles to $\mathcal{SHOIQ}$. This can be considered an extension $\mathcal{R}$ to the $\mathcal{H}$ features that were restricted to property hierarchies by including role hierarchy support.

Horrocks et al. [31], besides introducing $\mathcal{SROIQ}$, further describe an algorithm for $\mathcal{SROIQ}$ that has what they define as "pay as you go" characteristics. This means that the algorithm's behaviour is determined by the ontology's expressiveness. Concretely, they use the example of expressivity limited to $\mathcal{SHIQ}$ resulting in the algorithms behaviour being the same to that of an algorithm for $\mathcal{SHIQ}$.

2.5 Semantic technologies

Consider a stream as incrementally available time-stamped samples describing the values of features. Recall that a *sort* can be seen as a collection of *objects*, the properties of which are described by *features*. Additionally, properties can describe relations between objects. For example, we may have an object uav1 of sort UAV and a feature Altitude that maps UAV to a numeric value, e.g. Altitude[uav1] = 100. Stream reasoning can be regarded as incremental reasoning over metric temporal formulas [24]. While it is possible to manually specify which field in which stream to use to represent some term in a logical formula, this is cumbersome and error-prone, resulting in a fragile system. Earlier efforts [13, 24] sought to mitigate this issue by semantically annotating streams with the features they contain, by grounding these features in an OWL-DL ontology. OWL-DL was chosen over OWL-Full in order to retain decidability. This semantic annotation was done through an introduced language called the Semantic Specification Language for Topics (SSL_T), which made it possible to refer to ontological concepts in logical formulas as opposed to specific fields in specific streams. Objects in sorts were modeled as individuals of corresponding concepts in the ontology, and support for combining ontologies through bridge rules in C-OWL.

The work by Dragisic [13] resulted in an implementation that takes logical formulas and applies semantic matching to yield the 'addresses', i.e. which fields in which streams under which criteria, of the information described by the referenced ontological concepts. Unfortunately, it is not clear why specific semantic technologies (Jena and Pellet) were used. Therefore, an overview is given.

The semantic technologies used can be divided into two general areas,

being the OWL application programming interface (API) and the reasoner. The former provides an interface to manipulate and query OWL ontologies, and usually provides mechanisms to save or load ontologies in a number of syntaxes, where RDF/XML seems to be most prevalent. Individual OWL APIs may vary in their capabilities, which warrants careful consideration when choosing to use one. Reasoners, too, exist in many varieties and are tasked with operations on ontology models, like ontology inference, rule support and consistency checking. Reasoners may use different reasoning algorithms which affects their abilities and performance. For this reason, the choice of reasoner depends on a combination of the aforementioned elements. There have been studies [50] to compare the performance of reasoners using OWL benchmark ontologies, but these may not properly represent the current state of affairs with the introduction of OWL2.

The seemingly most popular OWL APIs at the time of writing are the *Apache Jena Framework*[5] and *OWL API*[6], both of which are written in the Java programming language. These APIs have varying reasoner support. Jena supports its own incomplete reasoner and Pellet, whereas OWL API supports Pellet[7], HermiT[8], FaCT++[9] and commercially available RacerPro[10]. On the flipside, Jena supports the SPARQL Protocol and RDF Query Language (SPARQL), whereas OWL API does not. SPARQL is a querying language similar to SQL that allows the user to query the ontology, and can be very useful depending on the application.

The aforementioned reasoners are written in Java, with the exceptions of FaCT++ and RacerPro. Both Pellet [50] and FaCT++ [52] support $\mathcal{SROIQ}(\mathbf{D})$, with RacerPro [18] and HermiT [46] supporting subsets of $\mathcal{SROIQ}(\mathbf{D})$. This limits the allowed expressivity when using the latter two reasoners. Lastly, of these four reasoners, FaCT++ is the only one that does not support the Semantic Web Rule Language (SWRL), which can be useful to add logical rules to ontologies.

[5] `http://jena.apache.org/`
[6] `http://owlapi.sourceforge.net/`
[7] `http://clarkparsia.com/pellet`
[8] `http://hermit-reasoner.com/`
[9] `http://code.google.com/p/factplusplus/`
[10] `http://www.racer-systems.com/`

3 DyKnow

In this chapter, DyKnow is discussed both from the perspective of a model and an implementation. In the case of the model, DyKnow is discussed at a conceptual level, introducing a mathematical description. For the implementation, the architecture as a whole is considered. The reader, upon completion of this chapter will have a general understanding of DyKnow at a conceptual level as well as at an architectural level for DyKnow-ROS. The concepts introduced in this chapter serve as an introduction to the next chapter, where we focus mainly on the contributions to the semantic integration module.

3.1 DyKnow as a model

Recall that DyKnow is a stream-based knowledge processing middleware framework. In this section, we cover DyKnow on a conceptual level by considering it as a model that can be described mathematically. Previous work [25] described DyKnow in a similar manner, and the approach taken here utilises some of the definitions and notations used.

3.1.1 Streams and knowledge processes

Streams are arguably the most important part of DyKnow, and are used to represent sequences of time-stamped samples that may for example be generated by sensors. Streams have *policies* attached to them, which are declarative specifications of the desired properties of a stream.

Definition 3.1.1 (Stream, *Heintz et al. (2010)*)**.** A *stream* is a set of *stream elements*, where each stream element is a tuple $\langle t_a, \dots \rangle$ whose first value, t_a, is a time-point representing the time when the element is *available* in the stream. This time-point is called the *available time* of a stream element

and has to be unique within a stream. A total order $<$ on time-points is assumed to exist.

Note that this general definition does not define a stream to contain samples, but more general stream elements. Because agents are often unable to access the exact value of a feature over time, they are forced to rely on approximations, which are represented by streams. In this thesis, these streams consist of samples representing approximated values of features at specific time-points. The time for which these values are approximated is called the *valid time*. A samples is represented by $\langle t_a, t_v, \vec{v} \rangle$, where $t_a \in T$ and $t_v \in T$ represent the available and valid time respectively, and $\vec{v}$ represents the value vector. We call a vector element $v_i \in \vec{v}$ a *field*, and use the shorthand notation $t_a(sa)$, $t_v(sa)$ and $\vec{v}(sa)$ for a sample sa.

Definition 3.1.2 (Field)**.** A *field* in a sample $sa = \langle t_a, t_v, \vec{v} \rangle$ from a stream s is a named element from a value vector

$$\vec{v}(sa) = \begin{pmatrix} a : v_1 \\ b : v_2 \\ \vdots \end{pmatrix},$$

where $\mathsf{FieldNames}(s) = \{a, b, \dots\}$ is a function taking a stream identifier s and returning the field names of $\vec{v}$ for its samples.

We can now introduce a stream space as the set of all possible streams within a system. The stream universe is considered to be the set of all possible streams, so the stream space is a subset of this.

Definition 3.1.3 (Stream space)**.** The *stream space* is the set $S \subseteq S^*$ of all active streams, where S^* is the set of all possible streams, called the *stream universe*.

Knowledge processes manipulate the stream space by generating new streams, and are instantiations of functions that are parameterised by streams, here called *transformations*. Given some DyKnow instance, the set of knowledge processes is denoted by F and the set of transformations is denoted by $\mathcal{F}$.

Definition 3.1.4 (Knowledge process, after *Heintz et al. (2010)*)**.** A *knowledge process* is a process whose inputs and outputs are in the form of streams.

We consider two specific types of knowledge processes: *computational units* and *sources*. Computational units take at least one input stream and produce a single output stream, whereas sources take no input streams and produce a single output stream. These are instances of refinement processes and primitive processes mentioned in earlier work [25], with the additional constraint that they may only have a single output stream each.

Definition 3.1.5 (Computational unit)**.** A *computational unit* is a knowledge process that takes one or more streams as input and produces a single output stream.

Definition 3.1.6 (Source)**.** A *source* is a knowledge process that takes no streams as input and produces a single output stream.

Sources may for example produce streams based on information gathered from sensor readings outside of the scope of DyKnow. Conceptually, a source can also be regarded as a special case of a computational unit.

3.1.2 Ontology

An ontology consists of concepts and relations between concepts. Additionally, individuals are instances of concepts that exist within an ontology. In the DyKnow model, we describe an ontology O as a tuple $\langle C, \Delta, \sqsubseteq \rangle$, where C denotes the concepts in the ontology, Δ denotes the individuals, and $\sqsubseteq$ denotes the is-a relation between concepts C. For the sake of clarity, we use O_C to denote the concepts, O_Δ to denote the individuals, and $O_\sqsubseteq \subseteq O_C \times O_C$ to denote the is-a relation. Following the notation used in $\mathcal{SROIQ}$, we denote 'o is an individual of concept C' by $o \in C$, and 'concept D is more general than concept C' by $C \sqsubseteq D$.

The ontology used in DyKnow consists of three main components.

- The *feature ontology* contains the domain features and their properties such as arity.
- The *sort ontology* contains the domain sorts and individuals described by Δ.
- The *unit ontology* contains information regarding units of measurement and conversion factors.

3.1.3 Stream and transformation specifications

Stream and transformation specifications are used to semantically annotate streams and transformations respectively.

Definition 3.1.7 (Stream specification)**.** A *stream specification* is a tuple $\left\langle s, (\phi : \alpha), \gamma(\vec{\delta}) \right\rangle$ consisting of a stream identifier $s \in S$, a feature concept $\phi \in O_C$ parameterised by a vector $\vec{\delta}$ of sorts or objects $\delta_i \in O_\Delta$, a field name α, and a partial mapping $\gamma(\vec{\delta})$ from $\vec{\delta}$ to $\mathsf{FieldNames}(s)$. We represent $\gamma(\vec{\delta})$ as a set of unordered pairs.

The intended meaning is that a stream s contains values of feature ϕ parameterised with sorts and/or objects $\vec{\delta}$, using mappings between these

ontological elements in O and fields in s. The set of stream specifications is denoted by $\hat{S}$. As an example stream specification, consider

$$\langle s_1, (\mathsf{Altitude} : \text{alt}), \{(\mathsf{UAV} : \text{id})\}\rangle ,$$

where for a stream s there exists a mapping between element in the ontology O and fields in $\mathsf{Fields}(s)$. Alternatively, when there is no mapping from e.g. object uav1, we get

$$\langle s_2, (\mathsf{Altitude} : \text{alt}), \{(\mathsf{uav1} : \text{no\_field})\}\rangle ,$$

where 'no_field' is used to represent that there is no mapping.

Definition 3.1.8 (Transformation specification)**.** A *transformation specification* is a tuple $\langle f, \phi, \{\psi_1, \ldots, \psi_n\}\rangle$ such that for a transformation identifier $f \in \mathcal{F}$, $\phi \in O_C$ and $\forall \psi_i : \psi_i \in O_C$.

The intended meaning is that the transformation f converts a vector of values for features ψ_i to a corresponding value of feature ϕ. The set of transformation specifications is denoted by $\hat{\mathcal{F}}$. As an example transformation specification, consider

$$\langle f, \mathsf{Speed}, \{\mathsf{Position}\}\rangle ,$$

which describes transformation f as transforming from feature Position to feature Speed.

3.1.4 DyKnow instance

By combining the previous elements, a DyKnow instance can be described by a tuple of the following components;

- a stream space $S \subseteq S^*$,
- a set of knowledge process instances F,
- a set of transformations $\mathcal{F}$,
- an ontology $O = \langle C, \Delta, \sqsubseteq\rangle$,
- a set of stream specifications $\hat{S}$, and
- a set of transformation specifications $\hat{\mathcal{F}}$,

where the DyKnow instance may change over time. We can then distinguish between two components, taking parts of this DyKnow instance tuple. The stream processing component of the model is described by $\langle S, F, \mathcal{F}\rangle$, and consists of streams, knowledge processes and transformations. This submodel contains all the elements necessary for manipulating the stream space. The

semantic integration component of the model is described by $\left\langle \hat{S}, \hat{\mathcal{F}} \right\rangle$, and contains semantic elements used for example for annotation of elements in the stream processing component. The semantic integration component is used to bridge the ontology O and the stream processing component. These two submodels combined make up the DyKnow model.

3.2 DyKnow and the Robot Operating System

The DyKnow model can be implemented in various ways. An earlier version [19] of DyKnow was implemented using the Common Object Request Broker Architecture (CORBA) [38]. CORBA was useful because it provided support for a distributed system where objects can be written in any language and hosted anywhere in a distributed system. The resulting implementation is referred to as DyKnow-CORBA in this thesis.

3.2.1 The Robot Operating System

More recently a ROS-based implementation of DyKnow has been developed [13, 34, 27], here called DyKnow-ROS. The Robot Operating System (ROS) [40] is a robotic software framework that enjoys increasing popularity in the robotics community. Similar to CORBA, it allows for a distributed system consisting of a number of processes called *nodes* in a peer-to-peer topology. A node can be written in C++, Python, Octave or LISP and acts as an independent program that is registered with the *master*, which serves as a central component for registry purposes similar to yellow pages. Nodes may host *services* which can be used by other nodes. Of special interest are *topics* to which nodes may subscribe, after which they receive *messages* sent over the topic.

By providing a modular framework with messaging and service support, ROS serves as a good candidate for a DyKnow implementation. The distributed nodes correspond well to the decentralised DyKnow framework, and allows for relatively simple customisation of DyKnow components. The support for multiple languages makes for a lot of flexibility, where for example C++ can be used for performance purposes or Python can be used for rapid prototyping or string handling. Topics in ROS provide good support for streams in DyKnow. Knowledge processes can subscribe to streams represented by topics in ROS and produce a resulting stream over a ROS topic.

ROS topics make use of a language-neutral *interface definition language* (IDL) to describe the fields contained by individual messages. These fields are strongly typed with primitive data types or other (custom) messages, and may be designated as arrays. An example is shown in Listing 3.1, which contains an array of Field messages. Header fields may be used to represent

time information associated with a message.

Listing 3.1: Sample message

```
Header header
time valid_time
Field[] fields
```

As any topic can only contain messages of a specific type, topics are strongly typed. This presents a problem for DyKnow as ROS messages are compiled, and therefore the collection of available message types is inherently fixed during runtime. Because DyKnow streams are not assumed to be fixed, a solution was developed [27] that considered type-flattening. When using type-flattening, every field in a Sample message shown in Listing 3.1 is represented using a triple of strings, which consists of a name, a type and a value, as shown in Listing 3.2. The resulting Sample topics represent streams in DyKnow, whose type may change during runtime.

Listing 3.2: Field message

```
string type
string name
string value
```

3.2.2 Architecture

We can now consider DyKnow-ROS instances from an architectural perspective. Given a temporal logic formula for evaluation, we want DyKnow to be able to synchronise the streams containing the necessary information, as part of its stream space manipulation support. This gives rise to three main components, consisting of engines, managers and coordinators. An *engine* takes a task specification and performs tasks according to the provided specification. A *manager* is responsible for storing specifications and contacting engines to perform tasks. A *coordinator* uses the services provided by engines and managers by calling these services in order to perform a high-level task. Figure 3.1 shows a high-level overview of DyKnow-ROS.

The *stream processing module* is responsible for hosting and manipulating streams. Its *stream processing engines* maintain instances of computational units and sources. Depending on the specifications it receives, it can apply *filtering* to individual streams and combine any number of streams through *merging* or *synchronisation*. The inner workings of the stream processing engines are beyond the scope of this thesis, but are discussed in the work by Hongslo [27] and uses a synchronisation algorithm designed by Heintz [19]. Managing the stream processing engines is the *stream manager*, which distributes specifications between the different engines and stores specifications for streams and transformations.

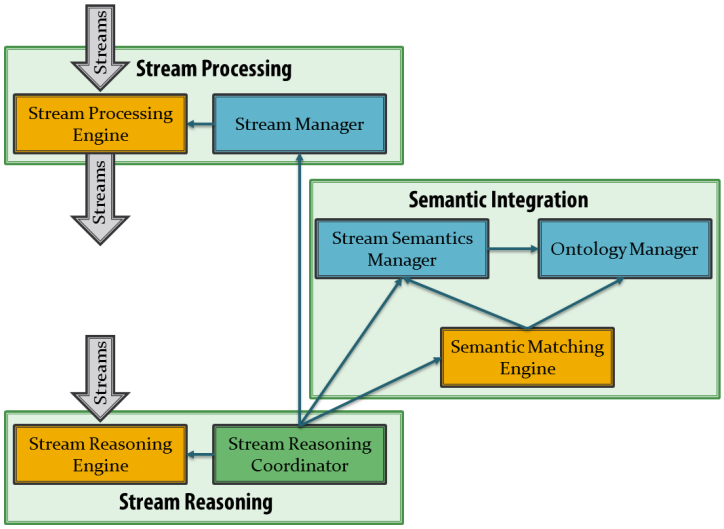

Figure 3.1: DyKnow high-level overview

The *semantic integration* module is responsible for storing semantic information and performs semantic matching to find streams containing desired information. It employs an ontology to create a common language and to specify relations between different concepts in the domain in which the DyKnow-ROS instance operates. The work by Dragisic [13] provided an early version of this module implemented in ROS. Most semantic technology tools are implemented in Java, which is not one of the languages supported by ROS. It was therefore decided to use rosjava-jni, which provides third-party support for Java implementations of ROS nodes. Unfortunately, rosjava-jni was recently replaced with rosjava[11], which lacks backward compatibility. Because the support of transformations required a lot of changes to the semantic integration module, it was decided to rewrite the module entirely. For support and performance reasons, C++ was used where possible. The connections with Java tools are maintained through the Java Native Interface (JNI), which allows C++ programs to access the Java Virtual Machine (JVM).

The semantic integration module consists of a *semantic matching engine*, a *stream semantics manager* and an *ontology manager*. The ontology manager is responsible for providing access to the ontology for reading and writing. Because DyKnow as a framework does not enforce a specific API to interact with ontologies, the goal is to support all major ontology APIs using a common interface. Since the experiments for this thesis project are partially performance-related, however, the choice was made for OWL API [28] since it supports FaCT++ [52] as reasoner. The semantic manager is

[11] `http://code.google.com/p/rosjava/`

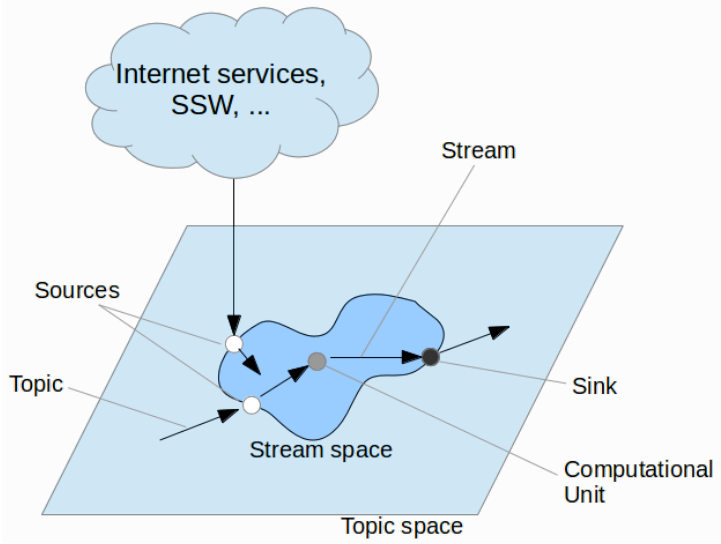

Figure 3.2: Stream space diagram

used to store semantic specifications of streams and transformations, and provides access to the semantic matching engine, which performs semantic matching utilising the semantic specifications and the ontology.

The *stream reasoning* module is responsible for the evaluation of temporal logic formulas. It consists of a *stream reasoning engine* and a *stream reasoning coordinator*. The stream reasoning engine was initially developed by Heintz [19] and later extended by Lazarovski [34] to support spatial relations in the metric temporal logic. It takes a temporal logic formula for evaluation along with a stream containing synchronised values grounded in the symbols used in the formula. Upon completion, it publishes the evaluation result over a ROS topic so that it can be reacted upon if desired. The stream reasoning coordinator provides high-level services by utilising the services provided by the individual engines and managers. This is where the initial request is made for the evaluation of a temporal logic formula.

A high-level overview of the resulting architecture is shown in Figure 3.1. When the stream reasoning coordinator receives a temporal logic formula, it first sends it to the semantic integration module for semantic matching. This results in an automatically constructed declarative statement, described later, which can be used with the stream manager to generate the desired stream specifications. The stream processing engine then takes these stream specifications to synchronise the desired information into a single stream, for which the stream name is returned. The stream reasoning coordinator then provides the stream reasoning engine with the formula and the obtained stream name for evaluation, and receives the stream name on which the evaluation result is published.

With these components, we can visualise the stream space in DyKnow-ROS and how it relates to the ROS framework. An abstract diagram is provided in Figure 3.2. It shows the *topic space* as a two-dimensional plane provided by the ROS framework, within which the DyKnow-ROS stream space resides. This is because streams in DyKnow-ROS are implemented as Sample topics. The sources act as gateways into the stream space. For this purpose topics may be used, but the flow of information used to generate a stream may also originate from elsewhere. Conceptually, sinks can be used to do the inverse of sources by generating messages on non-Sample topics or external to the topic space. Because DyKnow does not control the ROS environment, there are no constraints on the existence of ROS nodes outside of DyKnow that produce streams. Finally, within the stream space we can see computational units that take streams in order to produce refined streams. Note that the stream space is exclusively maintained and manipulated by the stream processing module, as all elements are related to either the stream processing engine, which is responsible for processing, or the stream manager, which is responsibe for names and configurations.

3.3 Stream handling in DyKnow-ROS

To allow both users and the DyKnow instance to manipulate the stream space, we can use a declarative language for constructing specifications to this end. This makes it possible for an algorithm to automatically filter and synchronise relevant streams necessary for the evaluation of some temporal logic formula. Additionally, it exposes the ability to manipulate the stream space to programs running in the DyKnow framework.

In this section, we describe two languages that are used to interact with DyKnow-ROS. To do so, we cover the syntax and semantics of the two languages and provide examples of their usage.

3.3.1 The Stream Processing Language (SPL)

The Stream Processing Language (SPL) was initially developed with topics in mind and was inspired by the Structured Query Language (SQL) [27] often used in relational database management systems (RDBMS). It is typically used to filter existing streams through selection and to combine streams through merging or synchronisation, and contains the option to set policy constraints. Additionally, aliases could be used to resolve conflicting field names and to improve readability.

However, SPL had a number of issues, the most critical being the lack of support for flattened messages in its implementation and the lack of support for transformations stemming from its design. To address these shortcomings, SPL was modified and extended. The resulting grammar is shown in Listing 3.3.

Listing 3.3: Formal grammar for SPL

```
decl               : source_decl | sink_decl | compunit_decl |
                     stream_decl ;
decls              : decl | decl SEMICOLON decls ;
source_decl        : 'source' type_decl STRING ;
sink_decl          : 'sink' stream ;
compunit_decl      : 'compunit' type_decl STRING LP
                     type_decl (COMMA type_decl)* RP ;
stream_decl        : 'stream' NAME EQ stream ;
type_decl          : basic_type | complex_type ;
basic_type         : NAME COLON type ;
complex_type       : LP basic_type (COMMA basic_type)* RP ;
type               : 'int' | 'float' | 'string' | 'boolean' ;

stream             : stream_term 'with' stream_constraints ;
streams            : stream | stream COMMA streams ;
stream_term        : STRING
                     | STRING LP streams RP
                     | 'sync' LP streams RP
                     | 'merge' LP streams RP
                     | LP 'select' select_exprs 'from' stream
                       ('where' where_exprs)? RP ;
select_exprs       : select_expr | select_expr COMMA select_exprs ;
select_expr        : field_id ('as' pstring)? ;
where_exprs        : where_expr | where_expr 'and' where_exprs ;
where_expr         : field_id EQ value ;
field_id           : STRING | STRING DOT field_id ;
pstring            : STRING | STRING? PERCENT field_id PERCENT
                     pstring? ;
value              : STRING | NUMBER
stream_constraints : stream_constraint | stream_constraint COMMA
                     stream_constraints ;
stream_constraint  : 'start_time' EQ NUMBER | 'end_time' EQ
                     NUMBER | 'max_delay' EQ NUMBER |
                     'sample_period' EQ NUMBER |
                     'sample_period_deviation' EQ NUMBER ;
```

Listing 3.4: Example SPL statements

```
s1 = select output as value from somestream where id = uav1
s2 = select output as value from cu(somestream, anotherstream) where id = uav1
s3 = merge(s1, s2)
s4 = sync(s1, s2) with sample_period = 100
s5 = select * from (select * from s1) where value = 0
s6 = sync(select value from s1 where value = 100, select * from cu(select *
     from s4, const double v 18.0))
```

SPL provides two key features: stream space manipulation support and knowledge process declaration support. The stream space manipulation support allows for the selection, synchronisation and merging of streams. Knowledge process declaration support allows for the declarative specification of a DyKnow instance by describing sources, sinks and computational units. As

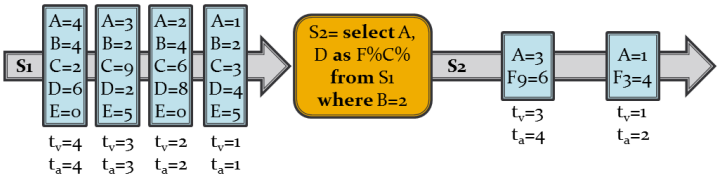

Figure 3.3: SPL select example, *Heintz (2013)*

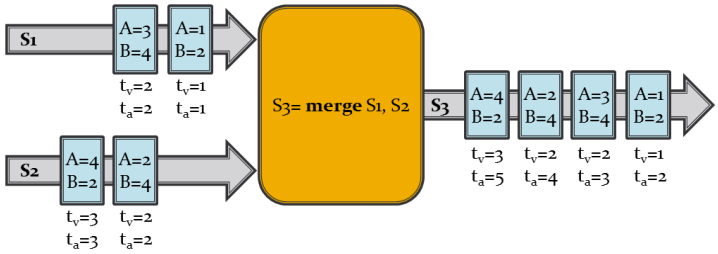

Figure 3.4: SPL merge example, *Heintz (2013)*

an example of stream space manipulation, consider the statements shown in Listing 3.4. If the parentheses around a select statement are omitted, a 'where' part is assumed to attach to the select statement to its immediate left in order to prevent ambiguity.

The first two statements in Listing 3.4 are select statements, where the first is called a *simple select* and the second a *complex select*. For the first statement, DyKnow is asked to select the field 'output' from stream somestream for all samples in which the field 'id' has a value equal to 'uav1'. The resulting stream is then called stream s1. For the second statement, DyKnow is requested to use a computational unit cu, which is parameterised by two streams. A complex select differentiates itself from simple select by the invocation of computational units in this way. DyKnow in this situation first creates an internal stream produced by cu, and then uses this stream to apply a simple select in order to apply the filtering. This resulting stream is then called stream s2. Figure 3.3 shows a diagram to illustrate a simple select statement. The arrows represent an incoming and outgoing stream. The blue boxes on the arrows are samples, within which the name-value pairs are written. Below the blue boxes, the valid time t_v and available time t_a are shown. Note that the 'D as F%C%' part in the figure renames field D to F with the value of field C as a suffix.

The third statement in Listing 3.4 shows a merge statement, with the intended meaning that stream s3 is constructed by combining all of the samples arriving from streams s1 and s2. This is illustrated in Figure 3.4.

The fourth statement in Listing 3.4 shows a synchronisation statement. During synchronisation, DyKnow is requested to generate a new stream at a certain frequency so that the values are all valid at the same time. Earlier

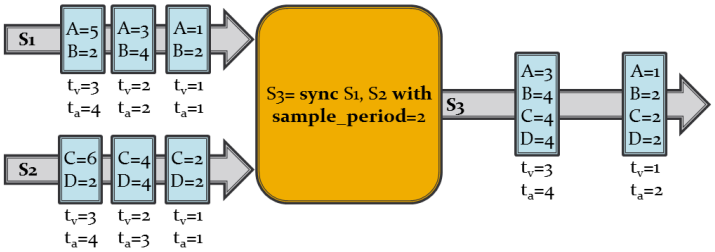

Figure 3.5: SPL synchronise example, *Heintz (2013)*

work [19] covered an algorithm to achieve synchronisation, which is beyond the scope of this thesis. The example statement tells DyKnow to synchronise streams `s1` and `s2` at every 100ms, producing a new stream `s4`. Figure 3.5 illustrates synchronisation.

The fifth statement shows the importance of the parentheses around select statement when 'where' parts are involved. In this example, the filtering is done over the outermost select statement. However, had the parentheses been absent, the filtering would have been done over the inner-most select statement instead. The resulting stream is called `s5`.

A more complex example is given in the final statement in Listing 3.4. In this example statement, it is requested to synchronise a simple select and a complex select, where the complex select applies computational unit `cu` to stream `s4` and a constant stream. A constant stream is a stream that sends a fixed sample specified in the declaration, so in this case a variable 'v' of type 'double' with value 18.0. In these nested declarations, there exists a tree structure that is leveraged by DyKnow. Using the example statement, first the stream parameterising the computational unit `cu` are created as internal streams. Then, these streams are used to instantiate the computational unit, creating another internal stream. The two remaining select statements inside the synchronisation declaration have now been reduced to simple select statements, so DyKnow handles both simple selects to create two more internal streams and uses these for synchronisation. The resulting synchronised stream is then called `s6`.

In order for DyKnow to be able to automatically use sources and computational units, these need to be declared in SPL. Listing 3.5 shows two example declarations. A declaration in SPL registers a label for a source or computational unit and details its input and output data types. These data types may be complex; they may consist of other data types.

Listing 3.5: Example SPL declarations

```
source output:(string, float) src
compunit output:(string, float) cu(input1:int, input2:int)
```

In the example declarations, a source `src` and a computational unit `cu`

are declared. The source has as output a complex data type consisting of a string and a float. The output field carries the name 'output'. The same holds for the computational unit declaration. In addition to the output, two input fields are declared, named 'output1' and 'output2', each of which assumes an integer.

3.3.2 Formal semantics of SPL

Now that an intuitive desciption of SPL has been given, we consider the formal semantics using the DyKnow submodels described earlier in this chapter. It is possible to describe the semantics of SPL using these models in combination with transition rules of the form

$$\frac{\text{declarative statement}, \phi_1, \ldots, \phi_n}{M^{\tau_0} \rightarrow N^{\tau_1}},$$

meaning that given an SPL statement and an optional list of models $\phi_1, \ldots \phi_n$, the DyKnow submodel M at time-point τ_0 transitions into N by time-point τ_1. Note that this is not standard operational semantics as the input is not part of the model. Instead, the declarative statement serves as the input that results in the model transition.

In the following paragraphs, we cover the transition rules for the simple select, complex select, merge and synchronisation statements in SPL. In these transitions rules, the function

$$\mathsf{ValidFields}(s, \vec{\alpha}) = \vec{\alpha} \subseteq \mathsf{FieldNames}(s)$$

may be used to test whether the fields $\vec{\alpha}$ indeed exist in some stream s.

Simple select Assuming that $\mathsf{ValidFields}(s, \vec{\alpha})$, meaning all selected fields indeed exist, the SPL transition rule for the simple select statement is described by

$$\frac{s = \text{select } \vec{\alpha} \text{ from } \beta \text{ where } \gamma(\vec{v})}{\langle S, F, \mathcal{F} \rangle^{\tau_0} \rightarrow \langle S \cup \{s\}, F, \mathcal{F} \rangle^{\tau_1}},$$

where $\beta \in S$. Additionally, $\exists sa' \in s$ iff $\forall sa \in \beta$ it is the case that $t_a(sa) \geq \tau_1$ and $\gamma(\vec{v})$ hold, s.t. $sa' = \langle t_a(sa), t_v(sa), \vec{\alpha} \rangle$. This means that the fields $\vec{\alpha}$ from stream β are contained by stream s iff the sample arrives at time-point τ_1 or later, and fulfills the constraints set by γ.

Complex select Assuming that $\mathsf{ValidFields}(s, \vec{\alpha})$, meaning all selected fields indeed exist, the SPL transition rule for the complex select statement is described by

$$\frac{s = \text{select } \vec{\alpha} \text{ from } f(\beta_1, \ldots, \beta_n) \text{ where } \gamma(\vec{v})}{\langle S, F, \mathcal{F} \rangle^{\tau_0} \rightarrow \langle S \cup \{s, \beta'\}, F \cup \{f(\beta_1, \ldots, \beta_n)\}, \mathcal{F} \rangle^{\tau_1}},$$

where $f \in \mathcal{F}$, $\{\beta_1, \ldots, \beta_n\} \subseteq S$, and $\beta' = f(\beta_1, \ldots, \beta_n)$. Additionally, $\exists sa' \in s$ iff $\forall sa \in \beta'$ it is the case that $t_a(sa) \geq \tau_1$ and $\gamma(\vec{v})$ hold, s.t. $sa' = \langle t_a(sa), t_v(sa), \vec{\alpha} \rangle$. This means that the fields $\vec{\alpha}$ from stream β' are contained by stream s iff the sample arrives at time-point τ_1 or later, and fulfills the constraints set by γ.

Merge The SPL transition rule for the merge statement combines the samples of the declared streams into the resulting stream and is described by

$$\frac{s = \text{merge}(\beta_1, \ldots, \beta_n)}{\langle S, F, \mathcal{F} \rangle^{\tau_0} \rightarrow \langle S \cup \{s\}, F, \mathcal{F} \rangle^{\tau_1}},$$

where $\{\beta_1, \ldots, \beta_n\} \subseteq S$. Further, it must be the case that $\forall sa \in \beta_i$ s.t. $t_a(sa) \geq \tau_1$ we have that $\exists sa' \in s$ s.t. $sa = sa'$, meaning that every sample in every stream β_i is contained in stream s.

Synchronise The SPL transition rule for the synchronisation statement is described by

$$\frac{s = \text{sync}(\beta_1, \ldots, \beta_n) \text{ with } \phi_1, \ldots, \phi_m}{\langle S, F, \mathcal{F} \rangle^{\tau_0} \rightarrow \langle S \cup \{s\}, F, \mathcal{F} \rangle^{\tau_1}},$$

where if the current time $t = \tau_{\text{start}} + n \cdot t_{\text{sample}}$ for some $n \in \mathbb{N}$ then $\exists sa \in s$ s.t.

- $t_a(sa) = t$,
- $t_v(sa) = \max_{\beta_i}(t_v(\mathsf{MostRecent}(\beta_i, t)))$, and
- $\vec{v}(sa) = (\vec{v}(\mathsf{MostRecent}(\beta_1, t)), \ldots, \vec{v}(\mathsf{MostRecent}(\beta_n, t)))$.

Here $\mathsf{MostRecent}(\beta, \tau)$ returns the sample $sa \in \beta$ s.t. $t_v(sa) \leq \tau$ holds and $\forall sa' \in \beta$ it is the case that $t_v(sa') \leq \tau \rightarrow t_v(sa) \geq t_v(sa')$. Additionally, $\phi_1, \ldots, \phi_m$ denote the stream properties, which determine τ_{start} and t_{sample}. Concretely, the most recent samples from streams $\beta_1, \ldots, \beta_n$ are synchronised at the frequency specified by t_{sample}.

Finally, recall that SPL aside from stream space manipulation support also provides knowledge process declaration support. A knowledge process can be either a source or a computational unit. Sinks are not included, as they conceptually act as an interface between DyKnow and something external to DyKnow. The following semantics make use of the term 'complex data type', defined below.

Definition 3.3.1 (Complex data type)**.** A *complex data type* is an n-ary tuple $(\phi_1, \ldots, \phi_n)$ s.t. $\phi_i \in \mathbf{T}$ or ϕ_i is a complex data type, where

$$\mathbf{T} = \{\text{int}, \text{float}, \text{bool}, \text{string}\}.$$

This implies that complex data types are nested lists of primitive data types described by **T**. As an example, (string, (float, float)) is a valid complex data type.

Source declaration A source declaration specifies a named source and its output value name and value type. It is described by

$$\frac{\text{source } \alpha : \phi \ src}{\langle S, F, \mathcal{F} \rangle^{\tau_0} \rightarrow \langle S, F, \mathcal{F} \cup \{src\} \rangle^{\tau_1}},$$

where

- ϕ is a complex data type,
- $\mathsf{Type}(src) = \phi$ describes the type for src, and
- $\mathsf{Output}(src) = \alpha$ described the output value name.

Computational unit declaration A computational unit declaration specifies a named computational unit and its output value name and value type, doing the same for its inputs. It is described by

$$\frac{\text{compunit } \alpha : \phi \ cu(\beta_1 : \psi_1, \ldots, \beta_n : \psi_n)}{\langle S, F, \mathcal{F} \rangle^{\tau_0} \rightarrow \langle S, F, \mathcal{F} \cup \{cu\} \rangle^{\tau_1}},$$

where

- ϕ is a complex data type, and
- $\forall \psi_i : \psi_i$ is a complex data type,
- $\mathsf{Type}(cu) = (\phi, \{\psi_1, \ldots, \psi_n\})$ describes the type for cu,
- $\mathsf{Inputs}(cu) = \{\beta_1, \ldots, \beta_n\}$ describes the input value names, and
- $\mathsf{Output}(cu) = \alpha$ describes the output value name.

3.3.3 The Factory Specification Language (FSL)

Recall that the stream processing submodel for DyKnow is represented by $\langle S, F, \mathcal{F} \rangle$. The stream space S is supported by the ROS architecture, but for the knowledge processes no implementation details were given. This is due to the fact that DyKnow serves as a framework that supports various implementations of knowledge processes. The Factory Specification Language (FSL) was developed to connect computational units and sources to external implementations. The FSL grammar is shown in Listing 3.6, and can easily be extended to include more path types.

Listing 3.6: Formal grammar for FSL

```
decl           : source_decl | sink_decl | compunit_decl ;
decls          : decl | decl SEMICOLON decls ;
source_decl    : 'source' NAME EQ path arguments? ;
sink_decl      : 'sink' NAME EQ path arguments? ;
compunit_decl  : 'compunit' NAME EQ path arguments? ;

path           : system_path | ros_path ;
system_path    : ('a'..'z'|'A'..'Z'|'0'..'9'| SLASH | DASH |
                 UNDERSCORE | DOT )+ ;
ros_path       : ('a'..'z'|'A'..'Z'|'0'..'9'| SLASH | DASH |
                 UNDERSCORE )+ ;
arguments      : NAME (COMMA NAME)* ;
NAME           : ('a'..'z'|'A'..'Z'|'0'..'9')+ ;
```

Listing 3.7: Example FSL statements

```
compunit dyknow_multiply = libdyknow_multiply.so
source sensor = /sensor_service 300
```

An FSL statement is converted into a factory specification, which is stored in the Stream Manager. Conceptually, the factory specification serves as a transformation in the DyKnow model. It serves as a blueprint for the generation of a knowledge process, which is a parameterised instance of a transformation. Given a specification and a collection of streams for parameterisation, DyKnow has sufficient information to create a knowledge process instance as was shown earlier. Two examples are shown in Listing 3.7, the first declaring a factory specification for a computational unit dyknow_multiply, and the second declaring a factory specification for a source sensor with some numeric argument. Note that the two specifications both refer to different implementations, with the computational unit using a Shared Object whereas the source uses a ROS service. The architecture of DyKnow allows for relatively easy extension of these implementation details.

3.4 Summary

In this chapter, DyKnow was discussed at the level of a mathematical model and at an architectural level. The Stream Processing Language syntax and semantics were covered, where the semantics were described in terms of transition rules between DyKnow models. Finally, the Factory Specification Language was introduced as a declarative interface between DyKnow and its implementation environment, where examples were given of knowledge processes implemented as Shared Objects and as ROS services.

This concludes an overview of the stream processing module. In the next chapter, the semantic integration module is covered, and the DyKnow model described here is used.

4 Semantic information integration

In this chapter, we first consider basic semantic information integration for stream reasoning. This is where an example ontology is presented, which is subsequently used to explain the workings of semantic annotations. The main contributions towards tackling the research question of this thesis are subsequently presented, as well as an algorithm for the resulting semantic matching. Upon completion of this chapter, the reader should understand the concepts behind semantic matching and the extensions developed and presented, and how these extensions may help answer the research question.

4.1 Semantic information integration for stream reasoning

DyKnow utilises semantic web technologies to partially tackle the problem of symbol grounding. Given a temporal logic formula, the goal is to synchronise the streams containing the necessary information for evaluation. The process of matching semantic specifications of streams or transformations to ontological concepts and individuals in temporal logic formulas is referred to as semantic matching. This process is performed by the semantic integration module. From a matching result, an SPL statement can be constructed. This statement can be used to generate the desired stream specification for the stream processing module. The resulting synchronised stream can then be used for formula evaluation. In this section, the relationship between semantic annotations and the ontology in DyKnow is discussed, introducing a declarative language used for such annotations.

4.1.1 Ontology

An ontology provides the semantic integration module with the information it needs in order to perform its intended function. This ontology contains

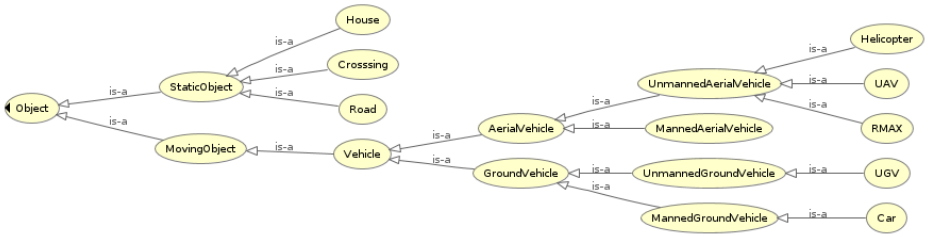

Figure 4.1: Example object ontology

information on the relationships between concepts and provides a common language with which temporal logic formulas and semantic annotations can be formulated. The ontology used in DyKnow-ROS consists of three subontologies.

- The *object ontology* contains the different object types and the generalisation relationship between them. Individuals in this ontology represent objects of a certain type.
- The *feature ontology* contains the different features that may be referred to. Features describe properties of objects, and therefore the ontology contains information on the arity of features as well as the relevant sorts for a given feature.
- Finally, the *unit ontology* contains information on units of measurement and is discussed in further detail later.

As an example, consider the object ontology shown in Figure 4.1. This ontology differentiates at the highest level between moving objects and static objects. Further down we can see a split between aerial vehicles and ground vehicles, leading up to the base sorts Car, UGV, RMAX, UAV and Helicopter. In this example, we will assume to have five objects, where uav1, uav2 and uav3 are individuals of UAV, and car1 and car2 are individuals of Car. Due to the generalisation relations between the various concepts, we obtain a generalisation tree such that car2 is a Car, MannedGroundVehicle, GroundVehicle, Vehicle, MovingObject, Object, and finally as is common for the root of an ontology but is not shown here, a Thing. This property makes it possible to consider the objects in this example when considering a formula for example of the form

$$\forall x \in \text{Vehicle} : \ldots,$$

as both Car and UAV objects are also Vehicle objects. Note that the formula makes use of the semantic concepts defined in the ontology as part of the common language.

With the object ontology as support, it is now possible to describe a feature ontology. An example is shown in Figure 4.2. It describes relations

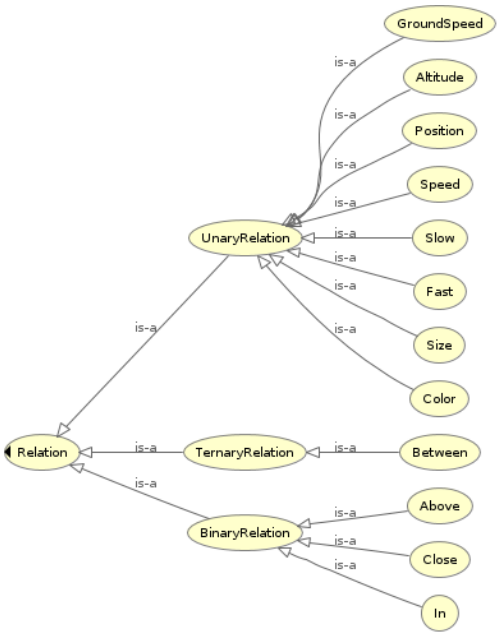

Figure 4.2: Example feature ontology

between sorts by their arity, where this example limits itself to unary, binary and ternary relations. Whereas unary relations such as Altitude describe a property of some object, higher arities may be used to describe relations between multiple objects. This is particularly useful for describing safety constraints, such as forcing UAVs to keep some distance between each other or marking geographic areas as off-limits, utilising the spatial reasoning support developed by Lazarovski [34].

Aside from creating a common language and providing a generalisation relationship between the feature concepts, the ontology also encodes constraints associated with the described features. For example, the feature Altitude only makes sense for objects that can have a varying altitude, such as aeroplanes and helicopters. Recall that it is possible to formally define these constraints using OWL object properties, here limited to $arg1$, $arg2$ and $arg3$ due to the arity limitation. In the case of Altitude, we can specify these restriction in Manchester syntax [29] as follows.

```
Class: Altitude
    SubClassOf: UnaryRelation and (arg1 only AerialVehicle)
```

DyKnow can use this to check whether provided temporal logic formulas are semantically valid.

Definition 4.1.1 (Semantically valid)**.** A temporal logic formula is called *semantically valid* iff all features used in the formula exist in the ontology,

every feature has the arity specified in the ontology, and every feature argument fulfills the type constraints specified in the ontology.

A formula has to be semantically valid in order for DyKnow to be able to evaluate it. As an example, consider the following temporal logic formulas.

$$\mathsf{Altitude}[\mathsf{uav2}, \mathsf{uav1}] > 10$$
$$\mathsf{Altitude}[\mathsf{car1}] > 10$$
$$\mathsf{Height}[\mathsf{uav3}] > 10$$

In this first example, the feature Altitude has an incorrect arity as it is supposed to be a unary feature. In the second example, the term used as a feature argument is of an incorrect type, as car1 is of sort Car, which has no generalisation relation to AerialVehicle. The last example uses a feature Height, which does not exist in the ontology. This latter example is of particular interest in the merging or combining of ontologies, for example on heterogeneous platforms, and has been covered in the context of DyKnow [13] using Context OWL (C-OWL) [8].

4.1.2 Semantic annotation with the Semantic Specification Language (SSL)

The next step is to consider methods to semantically annotate streams in DyKnow. This is done using semantic specifications. We differentiate between semantic stream specifications and semantic transformation specifications, where semantic transformation specifications are discussed later.

In order to generate semantic specifications, the Semantic Specification Language (SSL) was developed. The initial version of SSL was the Semantic Specification Language for Topics (SSL_T), and was used to semantically annotate ROS topics by the ontological concepts they contained [13]. Recent intermediate work [21] towards this thesis extended SSL_T with units of measurement and introduced the Semantic Specification Language for Transformations (SSL_{TF}). The focus on topics, however, presented a problem when considering streams in ROS. Since SSL_T focused on the annotation of the strictly typed fields in ROS, it was not possible to annotate the type-flattened data used in Sample topics. SSL was developed as a combination of the progress made and the need for stream support. Its full grammar is shown in Listing 4.1 and will be referred to throughout this chapter.

For now, we consider the base functionality of SSL to be the semantic annotation of streams. This is done by specifying for a given stream in DyKnow-ROS which fields contain information represented by which feature in the ontology. As an example, consider the SSL statements in Listing 4.2.

Listing 4.1: Formal grammar for SSL

```
decl           : stream_decl | source_decl | compunit_decl ;
stream_decl    : 'stream' NAME 'contains' feature_list
                 for_part? ;
source_decl    : 'source' NAME 'provides' field_feature ;
compunit_decl  : 'compunit' NAME 'transforms' field_features
                 'to' field_feature ;
field_features : field_feature (COMMA field_feature)* ;
field_feature  : NAME COLON NAME unit_list? ;
feature_list   : feature (COMMA feature)* ;
feature        : NAME LP feature_args RP EQ NAME unit_list? ;
feature_args   : feature_arg (COMMA feature_arg)* ;
feature_arg    : NAME alias? ;
for_part       : 'for' entity (COMMA entity)* ;
entity         : sort | object ;
unit_list      : ( OPEN unit (COMMA unit)* CLOSE ) | 'no_unit' ;
unit           : NAME power? ;
power          : ('+' | '-')? NUMBER ;
alias          : 'as' NAME ;
object         : entity_full ;
sort           : sort_type entity_full ;
entity_full    : NAME EQ NAME ;
sort_type      : 'some' | 'every' ;
NAME           : ( 'a'..'z'|'A'..'Z'|'0'..'9')+ ;
NUMBER         : ('0'..'9')+ ;
```

Listing 4.2: Example SSL statements for streams

```
stream s1 contains Altitude(uav1) = alt
stream s2 contains Altitude(uav1) = alt for uav1 = id
stream s3 contains Speed(UAV) = spd for every UAV = id
stream s4 contains XYDist(UAV as arg1, UAV as arg2) = dist for every arg1 =
    id1, arg2 = id2
```

The first statement states that stream s1 contains information on the Altitude of object uav1 in the field named 'alt'. This is different from the second statement, which states that stream s2 contains the same information as stream s1 with the difference that this is only the case when the field named 'id' has the value 'uav1'. The last two statements make use of sorts that are specified in the object ontology. Stream s3 contains information on the Speed for all objects in sort UAV, where the speed information is presented in the field named 'spd' for the UAV object referred to in the field named 'id'. We can see a similar construct in the semantic stream specification for stream s4. However, here we encounter some ambiguity as the sort UAV occurs twice. This is resolved by using an alias, in this case 'arg1' and 'arg2'.

4.2 Supporting units of measurement

Previously we considered temporal logic formulas that compared features to other features or values, for example

$$\forall x \in \mathsf{Car} : \Box \mathsf{Speed}[x] < 60.$$

While its meaning may be intuitive to a human reader, this intuitiveness is grounded in an assumption on the semantic meaning of the value 60 presented in the formula. To a reader using the metric system, 60 km/h may come to mind, whereas to a reader using the imperial system it may be 60 mi/h. Neglecting to solve this ambiguity may lead to problems after matching streams for the feature Speed.

When considering a stream space, knowledge processes may introduce dynamicity when they are instantiated and destroyed over time. These knowledge processes may have varying provenance and can act as sources or computational units. While this increases the amount of implicitly available information, special care is necessary with regards to the interpretation of the actual numeric values produced by these processes. When two streams containing the same features but assuming different units of measurement are regarded to be equal, this could lead to potentially disasterous situations. One example of this is the 1999 NASA 'Mars Climate Orbiter mishap' [51], where two teams provided software that assumed different units of measurement. The interaction between the software provided by these teams resulted in the loss of the Mars Climate Orbiter when it incorrectly calculated its altitude to be much higher than was actually the case. Such a scenario is not unlikely to repeat itself in distributed frameworks such as DyKnow, especially as the development of the Semantic Sensor Web progresses, making (live) sensor data available online.

The detection of inconsistencies in units of measurement is but one part of the problem; they need to be resolved as well. Since units of measurement represent scales for *physical quantities* such as length or mass, it is possible to convert between different units of measurement. The ability to do so automatically ensures that streams containing desired information may be used regardless of the assumed unit of measurement. This section explores the automatic transformation between units of measurement for this purpose, and extends work previously presented in Heintz and de Leng [21].

4.2.1 Ontology support for units of measurement

In order to use units of measurement in formulas, we need some kind of representation of these units. There have be numerous approaches to this representation problem. One example is the Unified Code for Units of Measurement (UCUM), which was proposed by Schadow et al. [45]. UCUM utilises a table format, where labels are mapped to units of measurement and

unit conversions are provided explicitly. The amount of unit conversions is kept low by making unit prefixes such as 'kilo' implicit. However, there are a few downsides to this approach. The UCUM tables assume a custom system of units described by length, time, mass, charge temperature, luminous intensity, and angle. Depending on the application area, this may be an undesirable system. Furthermore, due to its generality, the UCUM table can be overly complex for a given application that may depend on only a small subset of the encoded unit conversions.

An alternative to using a table representation of units of measurement and their conversions is the usage of an ontology. There are a number of ontologies seeking to represent units of measurement and conversion units. One example is the Measurement Units Ontology (MUO) [1], which seeks to model conversion tables such as UCUM. Rijgersberg et al. [43] analyse existing ontologies that incorporate units of measurement, including the Suggested Upper Merged Ontology (SUMO) [3], a subset of the Semantic Web for Earth and Environmental Terminology (SWEET) ontology [4], the EngMath ontology [17], ScadaOnWeb [14], and the OpenMath units and dimension CD groups [2]. The analysis for example looked at concepts such as unit prefixes and resulted in the Ontology of Units of Measure and Related Concepts (OM) [44].

Instead of selecting a specific approach, DyKnow supports all ontologies that satisfy a number of set criteria. This makes it possible to (partially) reuse existing or new ontologies, which is important as ontology representation of units of measurement is an active area of research. The criteria were chosen to be minimal while still providing the desired support for encoding units of measurement in an ontology, and are outlined below.

Units of measurement are related to their physical quantities It is possible to describe many physical quantities by a signature $\sigma = (\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \eta)$ corresponding to the unit $L^\alpha M^\beta T^\gamma I^\delta \Theta^\epsilon N^\zeta J^\eta$, where L represents length, M represents mass, T represents time, I represents electric current, Θ represents thermodynamic temperature, N represents amount of substance, and J represents luminous intensity [43]. For example, $\sigma = (1, 0, -1, 0, 0, 0, 0)$ describes speed as length (distance) divided by time. By associating (with an `unit-of` generalisation relation in OWL) for example {'metre', 'foot'} to length L and {'second'} to time T, speed L/T can be described in two different ways, either as metres per second or feet per second. The signature above assumes a seven-dimensional system, although any set of physical quantities may be used, including those not mentioned in the example. UCUM, for example, opted to use angle in its signature. The choice of base physical quantities, described by a signature of equal dimension, determines which physical quantities may be derived. The ability to choose a desired set of physical quantities makes it possible to keep a unit ontology minimal for its

intended domain. We could therefore describe the signature as $\sigma' = (\alpha, \beta, \gamma)$ if we only consider length, mass, and time. However, for completeness, the seven-dimensional system is used.

Every physical quantity must have a default representative unit of measurement The International System of Units (SI) uses metre, kilogram, second, ampere, kelvin, mole, and candela. However it co-exists alongside various other systems of units such as the imperial system. Additionally, minor variations such as using the gram as opposed to the kilogram for M may sometimes be desired. However, for every physical quantity there must be one default representative unit of measurement. These units of measurement make up the *system of units*, and are modeled through a relation in the ontology called `physquant` from a physical quantity to a unit of measurement.

Every physical quantity is closed under unit conversion This means that every unit of measurement is guaranteed to have a conversion function f and its inverse f^{-1} to the representative unit of a physical quantity, which makes it possible to convert any unit within a physical quantity to any other unit within the same physical quantity (see Theorem 1 in Section 4.2.3). For the representative unit of measurement, f and f^{-1} are identity functions. In the case of units on the ratio scale, the conversion function f will be of the form $f(x) = v \times x$, where v represents a conversion factor. For example, the conversion function for 'feet' to 'metres' is defined as $f(x) = 0.3048 \times x$. While it is possible to support non-ratio scale units such as degrees Celsius and degrees Fahrenheit using conversion functions of the form $f(x) = v \times x + b$, these are left out of this thesis as they are considered to be straightforward extensions of the proposed approach.

When the above criteria are satisifed, we end up with an ontology that presents a common language for units of measurement and contains information necessary to do unit conversions between different units of the same physical quantity, i.e. commensurable units of measurement. It is now possible to use these units in temporal logic formulas. This can be made easier by using equality relations for the different labels associated with units, for example 'm', 'metre' and 'meter'. The ontology we created fulfills the criteria by utilising OWL Annotations for `physquant` and conversion factor `v`. It implicitly specifies the default unit of measurement for a physical quantity by assigning a conversion factor of 1. The result is shown here using Manchester syntax and local names, and restricts itself to length and time. Note that the 'f' behind a number indicates that the value is assumed to be a float.

```
Class: Length
    SubClassOf: PhysicalQuantity

Class: Time
    SubClassOf: PhysicalQuantity


Class: h
    Annotations:
        v 3600.0f,
        physquant Time
    SubClassOf: Unit

Class: min
    Annotations:
        v 60.0f,
        physquant Time
    SubClassOf: Unit

Class: s
    Annotations:
        v 1.0f,
        physquant Time
    SubClassOf: Unit

Class: mi
    Annotations:
        v 1609.34f,
        physquant Length
    SubClassOf: Unit

Class: ft
    Annotations:
        v 0.3048f,
        physquant Length
    SubClassOf: Unit

Class: km
    Annotations:
        v 1000.0f,
        physquant Length
    SubClassOf: Unit
```

```
Class: m
    Annotations:
        v 1.0f,
        physquant Length
    SubClassOf: Unit
```

In order to convert from miles to kilometres in this example, we first convert from miles to the default unit of metres, followed by a conversion from metres to kilometres;

$$v^* = v_{mi} \times v_{km}^{-1} = 1.6093$$

More complicated conversions may occur when considering combinations of units of measurement. For example, consider the case of speed, where $\sigma = (1, 0, -1, 0, 0, 0, 0)$. We can use this signature in combination with the conversion factors encoded in the ontology to determine the resulting conversion factor. As an example, we can use the conversion from (mi/h) to (m/s), where we can alternatively write [mi.h-1] and [m.s-1] respectively. By plugging in the conversion factors, we obtain

$$v^* = (v_{mi} \times v_m^{-1}) \times (v_h \times v_s^{-1})^{-1} = 0.4470,$$

where v^* is the desired resulting conversion factor from (mi/h) to (m/s). One interesting problem resulting from unit conversion is how to retain precision. While DyKnow seeks to retain precision by performing the multiplications for the numerator and denominator separately prior to performing the division, precision cannot be guaranteed. This problem, however, remains outside of the scope of this thesis.

While it is possible to combine base units of measurement to describe derived physical quantities such as speed, it can at times be desirable to assign a label to these combinations. One example of this is the physical quantity pressure, with $\sigma = (-1, 1, -2, 0, 0, 0, 0)$. Here we have the units of measurement Pascal and bar, which are defined as $1\text{Pa} = 1\text{kg}/(\text{m} \cdot \text{s}^2)$ and $1\text{bar} = 10^5\text{Pa}$. We call these units *derived units of measurement*, and they can be used with physical quantities that are composed of base physical quantities. In order to represent these units of measurement, we can use a `derived` relation in the ontology. We want to use this to relate a derived unit of measurement to base units and their powers. This means `derived` will have to be a ternary relation, but this is not supported in OWL. Therefore, we choose to specify six binary derivation relations `derived`; one for every power in the set $\{-3, -2, -1, +1, +2, +3\}$. The choice of these six relations is based in the assumption that most commonly used derivation relations can be captured by these six relations, and if necessary the set can be expanded to include higher and lower powers. Note that the zero-power relation is always omitted as it is indirectly represented by the absense of a `derived`

relation. As an example of derived units, take Pascal, which can be written as [kg.m-1.s-2]. In order to represent Pascal, we need three `derived` relations; `(Pa derived+1 kg)`, `(Pa derived-1 m)`, and `(Pa derived-2 s)`.

While the `derived` relation allows us to specify which units of measurement a derived unit is derived from, it does not allow for units such as bar to be described directly in a similar way. The problem arises when derived units of measurement require a conversion factor on top of the `derived` relationship. In the case of bar, we need to represent 10^5[kg.m-1.s-2] in the ontology, but we are missing a way to specify this conversion factor of 10^5. In order to solve this discrepancy, we introduce the *derived conversion factor*, which is associated with bar in this case.

The presented requirements allow us to model units of measurement and conversion factors in an ontology. Existing ontologies such as SWEET and SUMO already contain concepts for physical quantities, and could therefore be extended with these criteria. Alternatively, new ontologies may be created from scratch, such as the unit ontology presented in the example earlier. Currently the possible conversion functions are quite limited due to the need to encode the variables into the ontology. One possibility to model these conversion functions as actual functions as opposed to their variables is by using the built in math functions in the Semantic Web Rule Language (SWRL) [32], but this is left for future work.

4.2.2 Semantic annotation of streams with units of measurement

Given a unit ontology containing conversion information for units of measurement, we also have a common vocabulary. In the previous section, we introduced a notation for describing combinations of units of measurement. This allows us to use units of measurement in temporal logic formulas. As an example, we can specify that

$$\forall x \in \mathsf{Car} : \Box \mathsf{Speed}[x] < 60 \text{ [mi.h-1]},$$

which has the intended meaning that all cars in the sort Car must always have a speed lower than 60 mi/h. This removes the ambiguity we saw earlier by making concrete the previously assumed unit of measurement. When no unit of measurement is given, a default unit `no_unit` is assumed.

In order to allow for semantic matching, it is important to incorporate this new element into the existing approach. Earlier, we could annotate a stream without specifying the unit of measurement. While this is still possible, and even necessary in the case of features such as e.g. NumberOfUAVs, this is now interpreted as having the unit `no_unit`. Listing 4.3 shows a few semantially sound examples of stream annotations in SSL that make use of units of measurement.

Listing 4.3: Example SSL statements with units of measurement

```
stream s5 contains Altitude(uav1) = alt [ft]
stream s6 contains Speed(UAV) = spd [km.h-1] for every UAV = id
stream s7 contains XYDist(UAV as arg1, UAV as arg2) = dist [m] for every arg1
    = id1, arg2 = id2
```

From the annotations in this example it is possible to obtain corresponding signatures σ. In the case of stream `s5`, for example, we can obtain $\sigma = (1, 0, 0, 0, 0, 0, 0)$, as ft is associated with the physical quantity of length. In the special case of `no_unit`, such a signature is undefined. The ability to infer signatures from units of measurement is important for semantic matching, as it may be used to check whether two units of measurement are commensurable, which is the case when their signatures match.

4.2.3 Semantic matching with units of measurement

The semantic matching problem is as follows. Given an ontology, a stream specification, and a parameterised feature find a set of streams which allows the estimation of the value of a feature over time. With the introduction of units of measurement, the previous semantic matching algorithm [24] must be extended to take into account these units of measurement during matching. An algorithm to this effect is presented in Procedure 1.

Procedure 1 Semantic matching with units of measurement

Input: A well-formed formula Φ and a set of stream specifications S
Output: Set of matching stream specifications S'

```
Set S' ← ∅
List F ← ExtractGroundFeatures(Φ)
for all f ∈ F do                                   ▷ Match and convert
    for all s ∈ MatchSpecs(S,f) do
        if Unit(s) = Unit(f) then
            S' ← S' ∪ {s}
        else if Convertable(s, Unit(f)) then
            S' ← S' ∪ {Convert(s, Unit(f))}
        end if
    end for
end for
return S'
```

In order to illustrate *unit alignment* in semantic matching, consider a formula: Altitude[uav1] $\geq$ 10ft. In this formula, we have object uav1, feature Altitude, and unit of measurement 'ft' for feet. Running the algorithm on the formula results in Altitude[uav1] being extracted and inserted into a list. Every element of this list is then checked to see if it is grounded. If the extracted feature had for example been Altitude[UAV], then it would have

been grounded for every object of the sort UAV yielding a new (expanded) list {Altitude[uav1], Altitude[uav2], Altitude[uav3]}. For Altitude[uav1], no expansion is necessary (or even possible).

Next, the stream specifications are matched. In our example, we use the three stream specifications from Listing 4.3. As per the non-extended semantic matching approach, streams containing information for feature Altitude[uav1] are selected. Only stream s5 contains the Altitude feature for the object uav1. Therefore, s5 is selected.

Finally, the unit of measurement for feature Altitude is checked. If the default unit of measurement for Altitude is 'ft', the matching procedure is done and returns stream s5. However, if the default unit is different from 'ft', this qualifies as a *misalignment problem*, and a unit conversion mechanism is used to fix the alignment if possible. When this happens, the resulting realigned topic is returned as a match. In the case where a feature is compared against another feature (e.g. Altitude[uav1] > Altitude[uav2]) instead of a constant such as in this example formula, the desired unit of measurement is set to be equal to the unit of measurement selected for the feature on the right-hand side.

The process of *unit alignment* is defined here as the process of converting the unit of measurement of a stream containing a feature in order to match the desired unit of measurement, and may be used to solve misalignment problems. Recall that every feature described by a physical unit can be described by a signature σ. For example, 'area' is described by $\sigma = (2, 0, 0, 0, 0, 0, 0)$, without specifying the unit of measurement used, which could be e.g. metres, feet, or even furlong.

Theorem 1 (Commensurability)**.** Given two physical quantities on the ratio scale, described by signatures σ_1 and σ_2, then if $\sigma_1 = \sigma_2$, there exists a transformation function f that converts the unit of measurement described by σ_1 to the unit of measurement described by σ_2.

Proof. Assume the criteria specified in Section 4.2.1 hold. Given two physical quantities, described by signatures σ_1 and σ_2, such that $\sigma_1 = \sigma_2$, we denote u_1 and u_2 to be the units of measurement for σ_1 and σ_2 respectively. From the closure criterion, there exists a function $f_1(u_1) = u_d$ and a function $f_2(u_2) = u_d$, where u_d is the default unit. Because of the requirement, a conversion function exists as well as its inverse, i.e. $f_2^{-1}(u_d) = u_2$. This means that $f = f_1 \circ f_2^{-1}$, and therefore $\exists f : f = f_1 \circ \cdots \circ f_n$. It is thus shown that there exists a transformation function f that converts the unit of measurement described by σ_1 to the unit of measurement described by σ_2. □

This shows that the criteria presented earlier allow for unit conversion by gathering the necessary conversion factors from the ontology. However, it is not the task of the semantic integration module to perform unit conversion.

4.2.4 Generation of SPL statements

Recall that the semantic integration module takes a temporal logic formula and returns a specification referring to the streams that need to be synchronised in order to evaluate the formula. As an example, consider a scenario where we wish to evaluate a temporal logic formula

$$\forall x \in \mathsf{UAV} : \Diamond_{[0,10000]}(\mathsf{Speed}[x] < 60\ \text{[mi.h-1]} \wedge \mathsf{Altitude}[x] > 10\ \text{ft}),$$

for which we have a stream uavstate with the following SSL annotation.

Listing 4.4: Example scenario SSL annotation

```
stream uavstate contains Altitude(UAV) = alt [m] , Speed(UAV) = spd [m.s-1]
    for every UAV = id
```

Assuming there are no other streams, the semantic matching algorithm from Procedure 1 will return uavstate as the resulting stream, albeit with some necessary unit conversions obtained from the example unit ontology:

$$v_{alt} = v_m \times v_{ft}^{-1} = 3.28084,$$
$$v_{spd} = (v_m \times v_{mi}^{-1}) \times (v_s \times v_h^{-1})^{-1} = 2.23694.$$

The information contained in the specification sent from the semantic integration module gives the stream reasoning coordinator the information it needs to automatically construct the SPL statement shown in Listing 4.5.

Listing 4.5: SPL synchronisation statement with units of measurement

```
SYNC (
    SELECT result AS Speed FROM dyknow_multiply(
        SELECT spd AS value FROM uavstate WHERE id = uav1 ,
        const double scalar 2.23694
    ) WHERE id = uav1 ,
    SELECT result AS Speed FROM dyknow_multiply(
        SELECT spd AS value FROM uavstate WHERE id = uav2 ,
        const double scalar 2.23694
    ) WHERE id = uav2 ,
    SELECT result AS Altitude FROM dyknow_multiply(
        SELECT alt AS value FROM uavstate WHERE id = uav1 ,
        const double scalar 3.28084
    ) WHERE id = uav1 ,
    SELECT result AS Altitude FROM dyknow_multiply(
        SELECT alt AS value FROM uavstate WHERE id = uav2 ,
        const double scalar 3.28084
    ) WHERE id = uav2
)
```

In this SPL statement, the features Speed and Altitude are selected from uavstate. However, because their units of measurement are misaligned from those used in the temporal logic formula, DyKnow automatically applies unit

conversion. The first step is to calculate the necessary conversion factors, which can then be represented as constant streams. Recall that a constant stream sends a single sample specified after the 'const' declaration. This is then combined with the original feature streams by using a computational unit dyknow_multiply. This is a default computational unit in DyKnow, which means it is part of the DyKnow architecture even though it can be used as an ordinary computational unit. It is parameterised by a stream containing a double named 'value', and (another stream containing) a double named 'scalar'. The resulting stream contains a double named 'result'. These named fields are referenced in the SPL statement using aliases.

The combination of an ontology, semantic annotations and a semantic matching algorithm have been shown to be sufficient in order to generate SPL statements for the evaluation of temporal logic formulas with transformations between units of measurement. Transformations such as dyknow_multiply for multiplying values in streams can however be generalised further, as is discussed in the next section.

4.3 Supporting feature transformations

The approach taken thus far only considers features for which streams already exist, for example transforming from an Altitude feature stream in feet to one in metres. While this helps in cases where a different unit of measurement is needed, the information presented is not new; the semantics of the values change only to represent a different scale. However, recall that DyKnow consists in part of transformations that are parameterised by streams to generate new streams. Data fusion techniques often take low-level data to produce higher-level information with a semantic meaning related to but different from that of the low-level data. A natural step is to consider semantic annotations of knowledge processes that display this behaviour. This step fits well with the ongoing research into the semantic web and semantic web services.

In this section, the semantic annotation of feature transformations is covered, followed by a semantic matching procedure that takes into account feature transformations in order to perform semantic matching. It extends the work presented in [21].

4.3.1 Semantic annotation of feature transformations

By semantically annotating transformations, it is possible to take into account these transformations during semantic matching. This can be useful in cases where certain features are only indirectly available through transformations. This makes it possible to automatically select the necessary transformations for the creation of a stream containing the desired feature. If one transformation is insufficient, multiple such transformations could be

chained together. In order to support this functionality, a semantic specification of a transformation is needed.

Transformation instances of this kind can be represented with computational units. In DyKnow a computational unit is seen as a function from a set of streams (the inputs) to an output stream. For the purpose of semantic matching we consider a computational unit C as a function from a set of features to a single feature, $C : \mathcal{F}^n \rightarrow \mathcal{F}$, where $\mathcal{F}$ denotes the set of all possible features in the ontology. The reason is that the ontology represents features, not streams, and since streams are annotated with features the connection is clear. In order to also represent the unit of measurement of a feature, each feature is represented as a tuple $\langle name, unit \rangle$. The feature name must refer to a feature in the feature ontology.

For example, imagine a stream s annotated with the feature f and two computational units C_1 and C_2. The transformations applied by these computational units are represented by functions $t_1 : f_{11} \rightarrow f_{12}$ and $t_2 : f_{21} \rightarrow f_{22}$. If it is the case that $f = (\mathsf{Altitude}, \mathrm{m})$ and $f_{11} = (\mathsf{Altitude}, \mathrm{m})$, then it follows that stream s and computational unit C_1 match since the same feature is associated with stream s and the input of C_1, i.e. $f = f_{11}$. However, if it is the case that $f_{21} = (\mathsf{Altitude}, \mathrm{ft})$, then C_2 does not match the stream s unless there is a unit transformation between m and ft.

Recall that in Theorem 1 the notion of commensurability was introduced in terms of signatures σ. We now introduce a function $\sigma : \mathcal{F} \rightarrow \Sigma$, where Σ is the set of all possible signatures σ. Concretely, this means that the function $\sigma(f)$ for a feature f yields the signature of the feature. In the previous example, this means that $\sigma(f) = \sigma(f_{11}) = \sigma(f_{21}) = (1, 0, 0, 0, 0, 0, 0)$. Because the signatures are the same, it can be concluded that stream s is commensurable with the inputs of computational units C_1 and C_2. In cases of misalignment, it is possible to distinguish between *commensurable misalignment*, which can be realigned through unit conversion, and *strict misalignment*, which can not be realigned through unit conversion.

SSL allows for the semantic annotations of such transformations by semantically annotating the transformation with its input features and output feature. This is in line with research into semantic web services. Alongside every feature the assumed unit of measurement as described in the previous section is also included.

Listing 4.6: Example SSL statements for transformations

```
compunit cu1 transforms from Distance [m] to Speed [m.s-1]
compunit cu2 transforms from Distance [km] to Speed [mi.h-1]
compunit cu3 transforms from NumVehicles no_unit to NumUAVs no_unit
source src1 provides Distance [m]
source src2 provides NumVehicles no_unit
```

Listing 4.6 shows a number of example SSL declarations used to describe transformation in this fashion. In this example, three computational units

and two sources are described by a semantic specification. The first computational unit, called cu1, transforms from the feature Distance to the feature Speed, where the temporal information in samples is utilised. It assumes distances in metres and speeds in m/s. Computational unit cu2 performs a similar transformation, but expects kilometres and mi/h respectively for its units of measurement. Not all features have to have units of measurements, and this is clearly shown in the case of computational unit cu3. The sources are considered to be transformations that take no inputs. Conceptually they can be regarded as manipulating the stream space by changing its composition, as they introduce new streams into the stream space. This also applies to computational units. The first source, called src1, provides the feature Distance in metres. Just like computational units, sources can handle features that do not assume units of measurement, as is shown for source src2.

The semantic integration module in DyKnow converts these SSL declarations to transformation specifications as defined in Definition 3.1.8. These can then be used during the semantic matching procedure.

4.3.2 Semantic matching with feature transformations

The semantic matching procedure can now be extended to take into consideration feature transformations. This functionality provides access to indirectly available features through the creation of streams containing these features. This can be useful in cases where DyKnow has no streams directly available for a desired feature, which may occur in a dynamic stream space.

In order to take into consideration feature transformations during semantic matching, the semantic matching procedure needs to be extended. The new semantic matching problem can be stated as follows. Given a desired feature f either find a commensurable stream s or a transformation tree tf generating a commensurable stream. A *transformation tree* is a tree where the interior nodes are computational units and the leaves are streams. The output of a transformation tree is the output of the root node (which can be a stream if the tree consists of only a leaf node). A computational unit with n inputs must have exactly n children. A transformation tree describes a *valid transformation* if every leaf node is a stream, each interior node is a computational unit with the right number of children, and the signature of each child is commensurable with the output of its sub-tree.

Procedure 2 describes an algorithm computing a valid transformation tree given a feature using the set of stream specifications $\hat{S}$ and transformation specifications $\hat{\mathcal{F}}$. The algorithm recursively extends the transformation tree one node at the time. To find a matching stream Procedure 1 is used. If no commensurable stream is found, then try to match each input of each computational unit which produces a commensurable stream. As soon as a valid transformation tree is found the algorithm may terminate. To in-

Procedure 2 Tree $match$(Feature f)

Assume: A set of stream specifications $\hat{S}$, a set of transformation specifications $\hat{\mathcal{F}}$, and a cache of results M.
if $f \notin M$ **then**
$M(f) \leftarrow$ Tree()
if $\exists s : f' \in \hat{S}$ such that $\sigma(f) = \sigma(f')$ **then**
$M(f) \leftarrow$ Tree(s)
else if $\exists C : f_1, \ldots, f_n \rightarrow f_{n+1} \in \hat{\mathcal{F}}$ such that $\sigma(f) = \sigma(f_{n+1}) \wedge t_1 = match(f_1) \neq$ Tree() $\wedge \ldots \wedge t_n = match(f_n) \neq$ Tree() **then**
$M(f) \leftarrow$ Tree(C, $[t_1, \ldots, t_n]$)
end if
end if
return $M(f)$

crease the efficiency and avoid cycles previous results are cached. Initially the cache is empty. By adding an empty tree as the default result before matching the feature allows cycles to be detected and handled.

The algorithm is non-deterministic in its choice of where to extend the transformation tree. Many different strategies can be employed to guide the search. If the transformation tree is extended in a depth-first manner and there is a solution it should normally be found quickly. The algorithm can also find optimal solutions by exhaustively trying all possibilities. The optimal solution could for example be the transformation tree with the least number of leaf nodes (streams) or the minimal maximal length of any path from the root to a leaf. If there are n features and m computational units then the worst case complexity is $\mathcal{O}(nm)$ since each feature is only computed at most once and each feature at most tries every computational unit once.

Upon creating a transformation tree, the semantic integration module has enough information in order for an SPL statement to be constructed in the stream reasoning coordinator, similar to the one shown in Listing 4.5. This will create instances of the transformations used in the transformation tree, thereby creating computational units parameterised to the desired streams. The resulting intermediate streams may be semantically annotated themselves, as their semantic specification is derived from the knowledge process generating the stream. Doing so will make the (intermediate) features directly available for semantic matching in DyKnow, albeit limited to the object domain they were instantiated for.

The ability to generate SPL statements based on SSL specifications and temporal logic formulas containing features described in an ontology shows that there is a clear relationship between semantic integration and stream processing in DyKnow. In the next section, this is made more clear through a formal description of the semantics of SSL.

4.4 Formal semantics of SSL

In the previous sections, SSL was described in the form of examples and the semantic matching procedure. Earlier the DyKnow model was described, and the semantics of SPL were formally defined in terms of transition rules between these models. The same can be done for SSL. Recall that transition rules are of the form

$$\frac{\text{declarative statement, } \phi_1, \ldots, \phi_n}{M^{\tau_0} \rightarrow N^{\tau_1}},$$

where $\phi_1, \ldots, \phi_n$ are optional DyKnow models that must hold at the time of the statement. In the case of SSL, a stream processing model is required since SSL describes the semantics of elements in this model.

In this section, the semantics of SSL are formally defined for the semantic annotation of streams, sources and computational units, utilising the previously described stream processing model.

Semantic annotation of streams The semantic annotation of streams adds semantic specifications for streams to the set of stream semantics specifications, and is described by

$$\frac{\text{stream } s \text{ contains } \phi(\vec{\delta}) = \alpha \text{ for } \gamma(\vec{\delta}), \langle S \cup \{s\}, F, \mathcal{F} \rangle^{\tau_0}}{\left\langle \hat{S}, \hat{\mathcal{F}} \right\rangle^{\tau_0} \rightarrow \left\langle \hat{S} \cup \left\{ \left\langle s, (\phi, \alpha), \gamma(\vec{\delta}) \right\rangle \right\}, \hat{\mathcal{F}} \right\rangle^{\tau_1}},$$

where $\langle S, F, \mathcal{F} \rangle^{\tau_0}$ is a stream processing submodel of DyKnow. Further, $\forall sa \in s$ we have $\alpha \in \mathsf{FieldNames}(sa)$ and there is some partial mapping $\gamma(\vec{\delta})$ from δ to $\mathsf{FieldNames}(sa)$ represented as a set of unordered pairs. Additionally, $\phi \in O_C$ is a feature and $\delta_i \in \vec{\delta}$ is either a sort $\delta_i \in O_C$ or an object $\delta_i \in O_\Delta$.

Semantic annotation of sources The semantic annotation of sources adds a semantic specification for that source to the set of semantic transformation specifications, and is described by

$$\frac{\text{source } src \text{ provides } \phi, \langle S, F, \mathcal{F} \cup \{src\} \rangle^{\tau_0}}{\left\langle \hat{S}, \hat{\mathcal{F}} \right\rangle^{\tau_0} \rightarrow \left\langle \hat{S}, \hat{\mathcal{F}} \cup \{\langle src, \phi, \varnothing \rangle\} \right\rangle^{\tau_1}},$$

where $\phi \in O_C$. Note that a source only provides a single feature and takes no features as arguments.

Semantic annotation of computational units The semantic annotation of computational units adds a semantic specification for that computational

unit to the set of semantic transformation specifications, and is described by

$$\frac{\text{compunit } cu \text{ transforms from } \phi \text{ to } \psi_1, \ldots, \psi_n, \langle S, F, \mathcal{F} \cup \{cu\}\rangle^{\tau_0}}{\left\langle \hat{S}, \hat{\mathcal{F}} \right\rangle^{\tau_0} \rightarrow \left\langle \hat{S}, \hat{\mathcal{F}} \cup \{\langle cu, \phi, \{\psi_1, \ldots, \psi_n\}\rangle\} \right\rangle^{\tau_1}},$$

where $\phi \in O_C$ and $\forall \psi_i : \psi_i \in O_C \wedge \psi_i \neq \phi$. It is similar to a source with the difference that a computational unit can take any number of features as inputs, as long as they are not the same as the output feature.

This concludes the semantics of SSL as well as SPL. One may have noticed that the ontology O, while referenced, is not manipulated by either SPL or SSL. This however does not mean it is considered to be static. Rather, anything may manipulate the ontology, be it by adding concepts or individuals, or by including bridge rules for the fusion of multiple ontologies. These manipulations are beyond the scope of this thesis, although the latter is explained in previous work on DyKnow [13, 24].

4.5 Summary

In this chapter, the semantic integration module of DyKnow was discussed in relation to the contributions made in this thesis. An updated version of SSL was presented and its formal semantics were discussed once the contributions were explained. Transformations between different units of measurement were described in terms of the ontological encoding of conversion factors through a predefined number of criteria. It was shown that if the criteria are upheld, the ontology contains the information necessary to convert between commensurable units of measurement. Next, feature transformations were considered using knowledge processes. In both cases, the necessary changes to the semantic matching procedure were covered.

This concludes the chapter on semantic information integration with the semantic integration module. In the next chapter, the quantitative experiments with DyKnow-ROS are covered.

5 Experiments

This chapter covers a number of experiments regarding the performance of DyKnow-ROS. The experiments are focused mainly on the semantic integration module, as the main contributions are made in this module. Additionally, the introduction of computational units is tested as part of the stream processing module, which relates strongly to the semantic matching procedure presented in this thesis. In the following experiments, the performance of individual modules in DyKnow-ROS is considered. These are the first DyKnow-ROS integration experiments since work on DyKnow-ROS began.

5.1 Experiment setup

The experiments are run on a laptop running Xubuntu 12.04 LTS, which is a light-weight derivative of Ubuntu using the Xfce Desktop. The CPU is an Intel Core i7-2670QM quad-core running at 2.2GHz with 4GB of RAM. The ROS version used for the experiments is ROS Hydro Medusa. The DyKnow-ROS engines, managers and coordinators are each run as separate ROS nodes written in C++, with the Semantic Integration module running as a single node using JNI for the OWL API, which in turn uses JNI for the FaCT++ reasoner. All parsers with the exception of the Stream Reasoning Engine formula parser are written as Python nodes using pyparsing, whereas the Stream Reasoning Engine uses ANTLR 2.7.7[12]. One instance of the Stream Processing Engine is run during the experiments. In order to obtain performance results, a performance manager node is used to subscribe to a performance topic over which the individual components send their performance data. This data is kept in memory during the experiments and written to disk in CSV format after the experiment is concluded. The result graphs are then generated using the R language.

[12] `http://antlr.org`

5.2 Experiments and results

This section presents the experiments and their results, after which an analysis is given. First, an experiment to test the underlying ROS framework is performed by varying the number of subscribers to a topic. Then the semantic integration module is tested by varying the number of stream specifications, ontological concepts and ontological individuals. Lastly, the stream processing module is tested by varying the number of necessary transformations performed by computational unit instances.

Where applicable, stacked barcharts are used to represent the time needed by the various components of DyKnow-ROS. Six components are identified;

- *symbol extraction* is the procedure of extracting symbols from a temporal logic formula,
- *expansion* denotes the expansion of features in the case of sorts,
- *matching* denotes the semantic matching procedure,
- *SPL construction* refers to the construction of an SPL statement once semantic matching has finished,
- *SPL parsing* denotes the creation of a specification tree from the SPL statement,
- *stream creation* is the execution of the specification tree resulting in a number of streams, and
- *evaluation request* denotes the process of providing the stream reasoning engine with the information necessary to start evaluation.

The barcharts provide module-specific information by separating the semantic integration module from the stream processing and stream reasoning modules in terms of performance measurements.

5.2.1 Varying the number of subscribers

Streams in DyKnow-ROS are realised using ROS topics which can be subscribed to. When there exists a number of streams, a stream can be subscribed to by many different nodes. In order to test the scalability of DyKnow-ROS, an experiment is done where for every subscriber the average delay between the sending and receiving of a sample is kept track of. This provides insight into the scalability of streams. In order to run this experiment, a publisher node is used to send messages containing the sending time. A receiver node has 1500 subscriptions to the stream published by the publisher node and determines the delays for every subscription. This

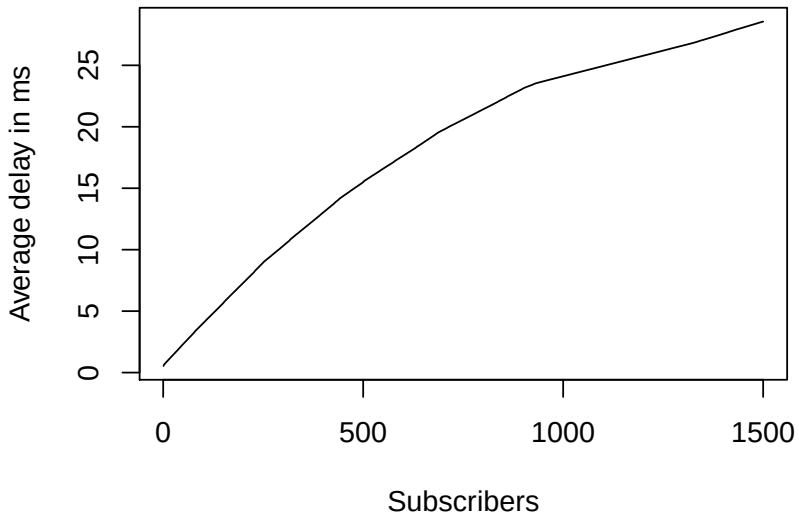

Figure 5.1: Average delay by subscriber where all subscriptions are to the same topic

experiment is run five times after which the average is taken in order to smoothen out any outliers.

The results of the experiment are shown in Figure 5.1. The first subscriber receives a sent message almost instantaneously within a single milisecond, whereas the 1500th subscriber experiences a delay of close to 30 ms. This is an extremely small delay. Furthermore, it can be observed that the graph seems to be logarithmic as the number of subscribers increases. This is a good indication that DyKnow-ROS can easily handle large numbers of subscribers.

5.2.2 Varying the number of stream specifications

The number of semantic specifications for a temporal logic formula in part determines the running time of the semantic matching procedure. When there exist many stream specifications, the semantic matching procedure needs to consider all of them, which results in a lower performance. In order to verify this, an experiment is performed where the number of stream specifications gradually increases from 0 to 4000. The temporal logic formula used here and in future experiments is

$$\forall x \in \mathsf{UAV} : \Box_{[0,1000]}(\mathsf{Altitude}[x] > 10 \text{ ft} \wedge \mathsf{Speed}[x] < 60 \text{ mi/h}).$$

A single relevant stream specification is used to ensure that no unit conversion is necessary:

Listing 5.1: Experiment stream specification

```
stream uavstate contains Altitude(UAV) = alt [ft], Speed(UAV) = spd [mi.h-1]
    for every UAV = id
```

The remaining stream specifications refer to features of differing arity that are not used in this temporal logic formula. Additionally, the ontology contains 50 concepts and 50 individuals representing UAV objects. Because all stream specifications need to be considered by the semantic matching procedure, they impact performance. By minimising the number of relevant specifications, we can determine a lower bound on the performance impact.

The results are shown in Figure 5.2, which is a stacked barchart showing the time needed by different components in DyKnow-ROS. The values presented are averages over 50 individual runs in order to smoothen out any outliers. It can be observed that the time needed by the semantic matching procedure increases linearly as the number of stream specifications increases. Additionally, the performance of the stream processing module and the evaluation request remain constant.

5.2.3 Varying the number of concepts in the ontology

By changing the number of concepts in the ontology, the API used to interact with the ontology is expected to determine the change in performance. Earlier experiments [13] used the Jena framework in order to interact with the ontology. Recall that in this experiment the OWL API is used instead, alongside the FaCT++ reasoner. During this experiment, the number of individuals representing objects of the sort UAV is fixed at 50, and the number of concepts other than those necessary for the evaluation of the formula presented in the previous experiment is gradually increased from 0 to 1000 in addition to the concepts necessary for formula evaluation. Only the semantic specification necessary to evaluate the temporal logic formula presented earlier is considered.

The results are shown in Figure 5.3 and are obtained through 50 individual runs over which the averages are taken. It is clear that the performance of all components remains constant as the number of concepts is increased. This is a different result from the earlier experiments, where there was a performance impact. This may be due to the fact different OWL APIs were used, as well as different implementations of the semantic integration module.

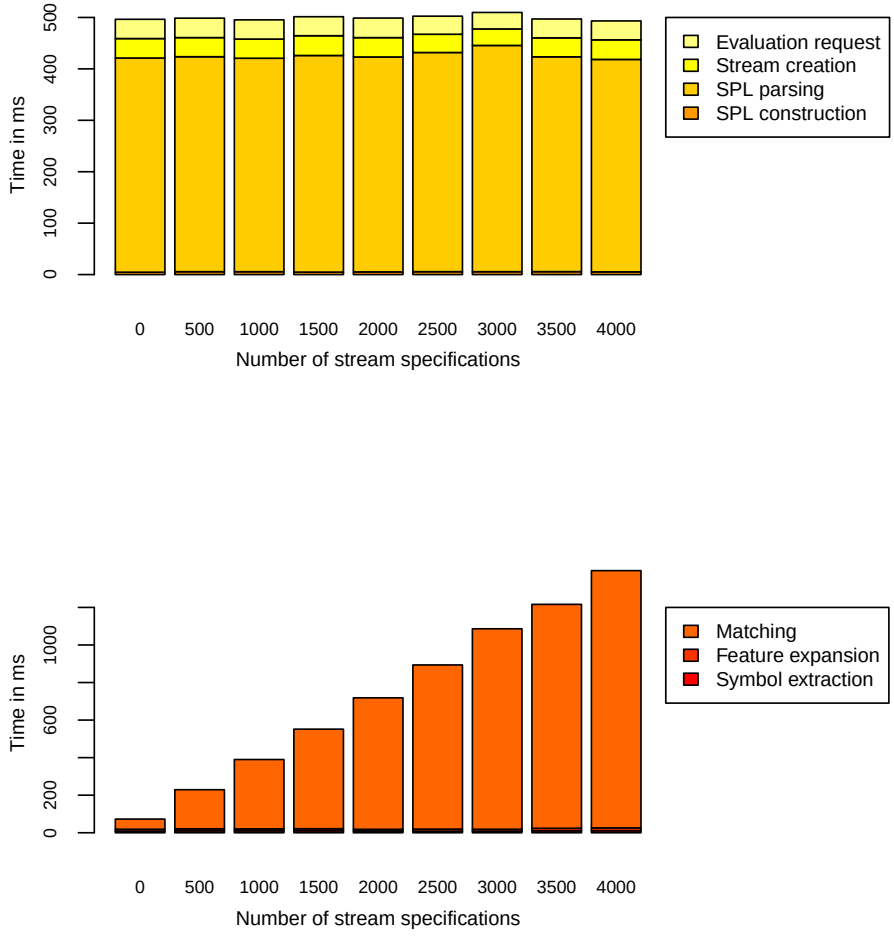

Figure 5.2: Stacked barcharts showing the time needed by individual components in DyKnow-ROS when the number of stream specifications increases

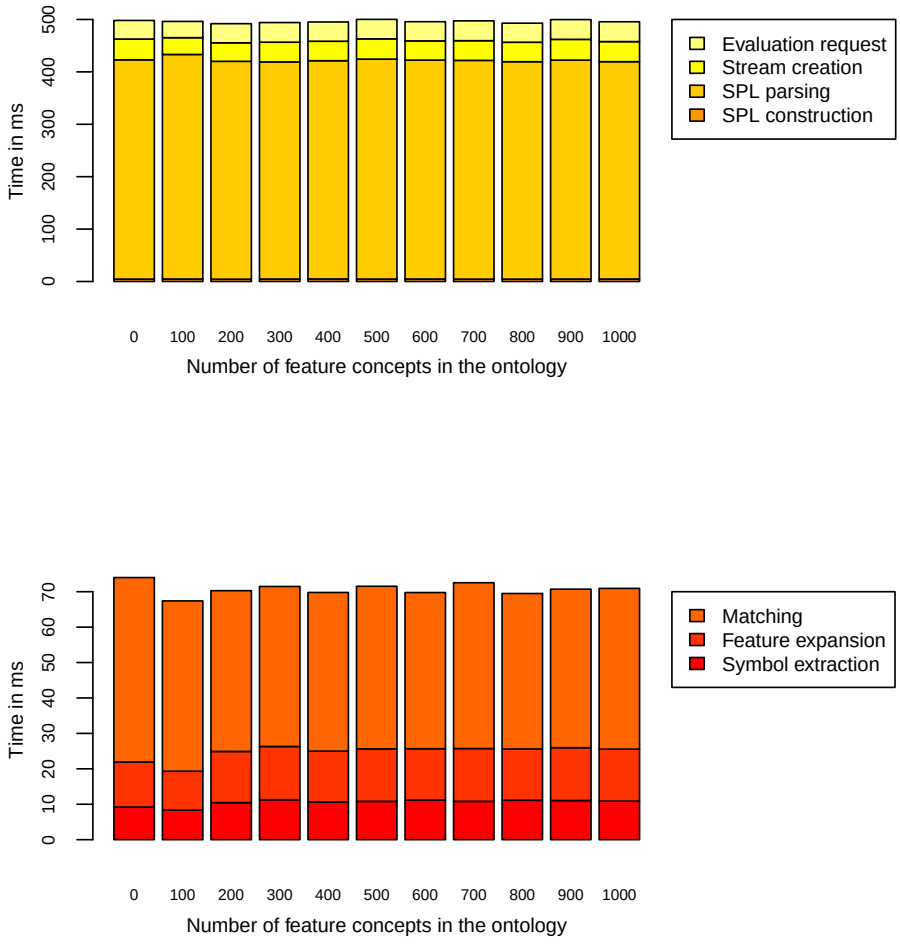

Figure 5.3: Stacked barcharts showing the time needed by individual components in DyKnow-ROS when the number of concepts in the ontology increases

5.2.4 Varing the number of individuals in the ontology

By increasing the number of individuals in the ontology, the number of represented objects is increased. This is interesting because it increases the length of the resulting SPL statement. Still using the temporal logic formula presented earlier, no semantic specifications are used in addition to the one necessary for formula evaluation. The number of objects in the ontology gradually increases from 50 to 500. Only unary features are considered.

The results are shown in Figure 5.4 and are obtained through 50 individual runs over which the averages are taken. As expected, the stream processing module shows a linear increase in time needed. However, the semantic integration module shows a quadratic increase in time needed. Closer inspection shows that this is due to the expansion of sorts in specifications during semantic matching, which takes $\mathcal{O}(n)$ time and is done for every expanded unary formula feature. This results in a performance of $\mathcal{O}(n^2)$ time for unary features. By expanding semantic specifications immediately when they are added, a $\mathcal{O}(n^a)$ performance may be achieved for a-ary features. However, this is left for future work.

5.2.5 Varying the number of computational units

The performance of DyKnow-ROS in part depends on the number of computational units running at any given time. Initial experiments have shown that ROS is unable to handle a load of around 700 dyknow_multiply computational units at any given time. Crossing this threshold results in the ROS master freezing. In this experiment, the performance of DyKnow-ROS is measured through the evaluation of a temporal logic formula that requires unit conversion, thus resulting in the instantiation of computational units to this effect. By increasing the number of objects, the number of unit conversions increases at the same rate. For this experiment, 50 concepts are used and the number of individuals representing objects is gradually increased from 10 to 100. Only a single stream specification is used, which is the same one presented earlier. The temporal logic formula, however, is changed to

$$\forall x \in \mathsf{UAV} : \Box_{[0,1000]}(\mathsf{Altitude}[x] > 10 \text{ m} \wedge \mathsf{Speed}[x] < 120 \text{ km/h}),$$

which entails a unit conversion from imperial units to metric units for formula evaluation. Because there are two unit conversions involved, every object needs two dyknow_multiply computational unit instances, which means that the number of necessary transformations gradually increases from 20 to 200.

The results are shown in Figure 5.5 and are obtained through 50 individual runs over which the averages are taken. We can see a linear increase in time needed for the stream processing module. While the semantic integration module makes the performance $\mathcal{O}(n^2)$ for unary features in $\lim_{n\to\infty}$,

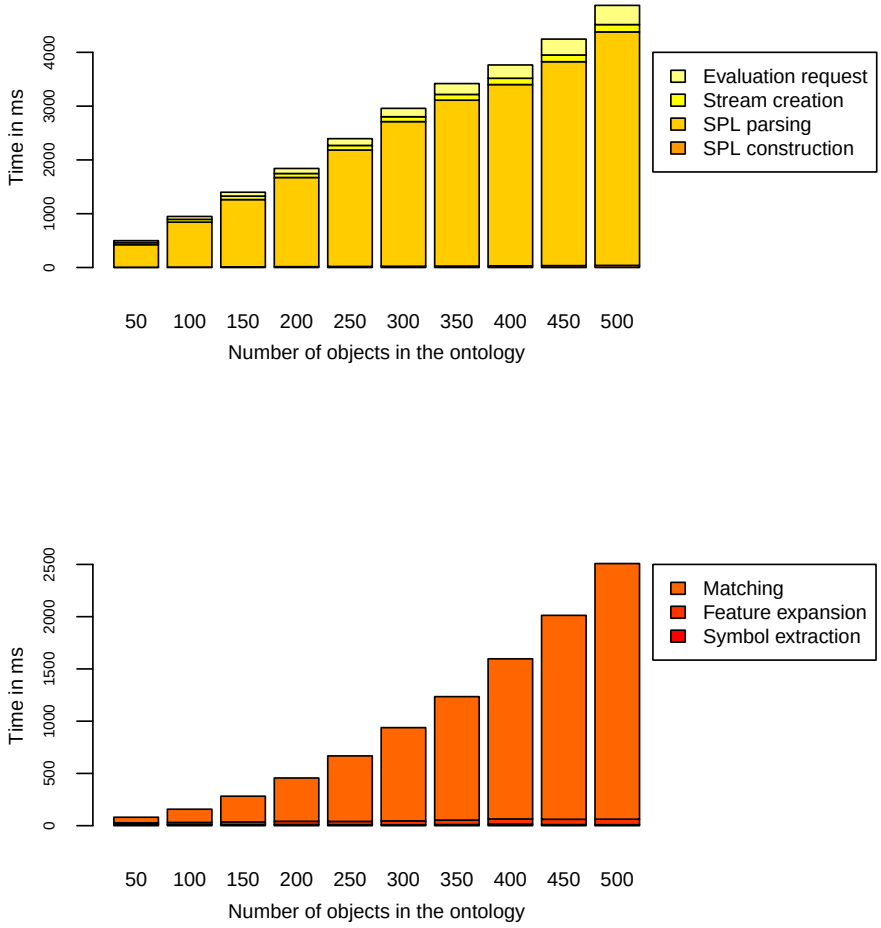

Figure 5.4: Stacked barcharts showing the time needed by individual components in DyKnow-ROS when the number of individuals in the ontology increases

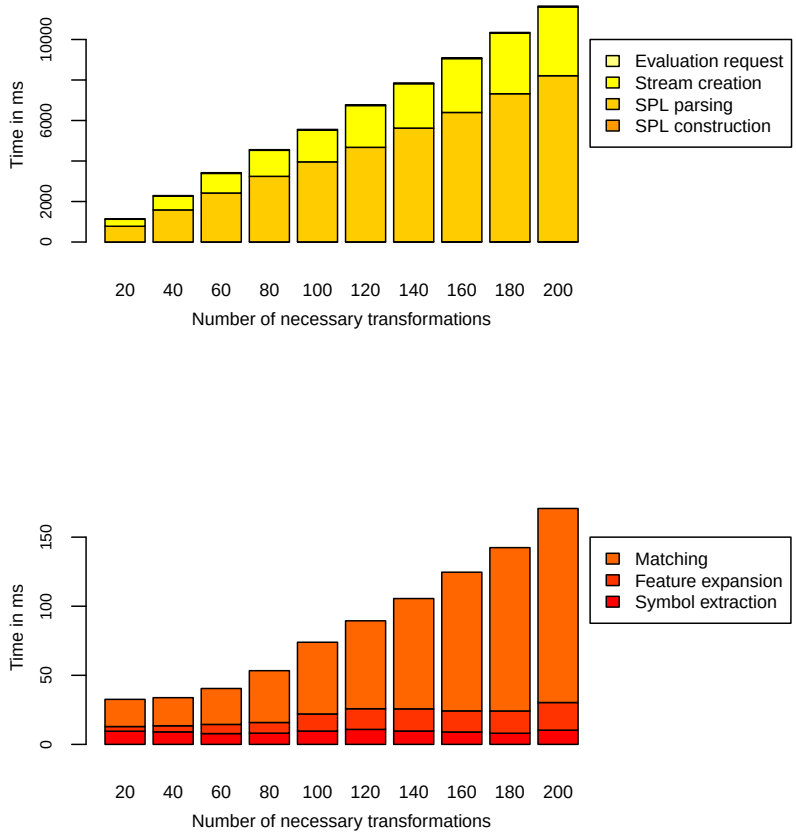

Figure 5.5: Stacked barcharts showing the time needed by individual components in DyKnow-ROS when the number of necessary transformations increases

its contribution is practically negligible when compared to the contribution of the stream processing module. Of special interest is the time required by the stream processing module, which, albeit linearly, shows a steep increase in the amount of time used. For 200 necessary transformations this time has grown to be over 5 seconds, which corresponds to an average of 50ms per computational unit instance.

5.3 Discussion

The results presented in the previous sections give an indication for the scalability of DyKnow-ROS. Throughout the experiments, the semantic integration module shows a high performance that is mostly impacted when the number of individuals in the ontology increases. However, the SPL parsing in the stream processing module is generally slow, moreso when the number of individuals in the ontology increases or computational units are involved.

5.3.1 SPL parsing

Not surprisingly, there seems to be a relation between the length of an SPL statement and the time it takes to parse this statement into a specification. The impact of the experiments in the previous section on the SPL statement can be shown for both the situation where no unit conversion is necessary and when it is necessary. In the former case, the SPL statement for 2000 objects is shown in Listing 5.2. This entails the creation of 4000 intermediate streams that are subsequently synchronised into a final stream used for evaluation. The length of the SPL statements becomes more problematic when computational units are involved, as shown in Listing 5.3.

This performance issue is caused by the need to synchronise streams such that every sample contains the desired information for every object. One interesting angle to improving this is by looking into ways for computational units to handle synchronised samples. That way the transformations performed by the computational unit can be deferred to after the synchronisation step. However, this remains an open problem.

5.3.2 Semantic integration performance

Looking back at earlier experiments [13] where the performance of the semantic integration module was tested, it is clear that there has been an increase in performance. This performance increase is quite large; in some experiments the improvement is a time reduction by ten-fold. This is most likely due to a number of factors related to the new implementation of the semantic integration module. Firstly, the semantic integration module presented in this thesis is written in C++ and uses a different OWL API

and reasoner. Secondly, the semantic integration module presented in the previous work used a lot of interaction with the disk, which can dramatically slow down an application. The difference between the hardware used to run the experiments can be considered negligible. The results presented relate well to the second subgoal of this thesis regarding performance.

Listing 5.2: SPL statement without computational units

```
SYNC (
    SELECT alt AS Altitude FROM uavstate WHERE id = uav1 ,
    ...
    SELECT alt AS Altitude FROM uavstate WHERE id = uav2000 ,
    SELECT spd AS Speed FROM uavstate WHERE id = uav1 ,
    ...
    SELECT spd AS Speed FROM uavstate WHERE id = uav2000
)
```

Listing 5.3: SPL statement with computational units

```
SYNC (
    SELECT result AS Speed FROM dyknow_multiply(
        SELECT spd AS value FROM uavstate WHERE id = uav1 ,
        const double scalar 1.60934
    ) WHERE id = uav1 ,
    ...
    SELECT result AS Speed FROM dyknow_multiply(
        SELECT spd AS value FROM uavstate WHERE id = uav2000 ,
        const double scalar 1.60934
    ) WHERE id = uav2000 ,
    SELECT result AS Altitude FROM dyknow_multiply(
        SELECT alt AS value FROM uavstate WHERE id = uav1 ,
        const double scalar 0.3048
    ) WHERE id = uav1 ,
    ...
    SELECT result AS Altitude FROM dyknow_multiply(
        SELECT alt AS value FROM uavstate WHERE id = uav2000 ,
        const double scalar 0.3048
    ) WHERE id = uav2000
)
```

6

Conclusions and future work

The research question studied in this thesis is how to efficiently extend DyKnow to handle the semantic matching of indirectly-available streams. Two subgoals towards answering this research question were defined. The first subgoal was to extend the semantic information integration in DyKnow to better handle indirectly-available streams. This was done by introducing transformations between different features and transformations between units of measurement. These transformations were implemented in DyKnow-ROS. In order to support these changes, the stream processing language SPL was extended, the semantic specification language SSL was extended and combined from SSL_T and SSL_{TF}, and the factory specification language FSL was introduced. The SPL and SSL languages were formalised by considering their semantics using transition rules. Additionally, the semantic matching algorithm was extended to support transformations. The second subgoal was described as achieving good scalability and a low response time. These properties were tested through quantitative experiments and show at most quadratic increases in the time used by individual modules relative to the number of objects and necessary transformations. Improvements have been proposed to reduce this to linear increases. Further performance increase is needed in the case of SPL parsing, which is still relatively slow in certain circumstances.

The introduction of transformations allows DyKnow to start tapping into latent streams. This thesis covers theory, algorithms and applications by extending DyKnow's formal semantics and providing a concrete extension to earlier work. The extensions described in this thesis provide a good initial solution to the problem and open up many interesting opportunities for future work. The progress made on the semantic web may provide additional angles towards improving the semantic annotation of streams, and may itself present an additional challenge by providing streams of sensor information over the semantic sensor web. Ubiquitous stream reasoning

utilising sensor information from different geographic areas from different points in time is interesting for purposes of situation awareness. Continuous semantic information integration considers streams that may start and stop at any time, as well as knowledge processes that may become available and disappear over time. This is a challenging problem and affects stream reasoning directly. Future work may also include a graphical representation of DyKnow information for a user, which may for example utilise packages such as rosbridge_suite.

The contributions made in this thesis allow DyKnow to utilise streams of information that were previously out of reach. By making available this previously unavailable information, DyKnow can better handle situations that would previously result in the failure to obtain any streams through semantic matching.

Bibliography

[1] MUO. Available at `http://idi.fundacionctic.org/muo/`.

[2] OpenMath. Available at `http://www.openmath.org/ontology/`.

[3] SUMO. Available at `http://www.ontologyportal.org/`.

[4] SWEET ontologies. Available at `http://sweet.jpl.nasa.gov/`.

[5] Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. MIT Press, Cambridge, MA, USA, 2004.

[6] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. Terse RDF triple language. 2013.

[7] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, pages 29–37, May 2001.

[8] Paolo Bouquet, Fausto Giunchiglia, Frank Van Harmelen, Luciano Serafini, and Heiner Stuckenschmidt. C-OWL: Contextualizing ontologies. In *Journal Of Web Semantics*, pages 164–179. Springer Verlag, 2003.

[9] Dan Brickley and R.V. Guha. RDF vocabulary description language 1.0: RDF schema. 2004.

[10] Arne Bröring, Krzysztof Janowicz, Christoph Stasch, and Werner Kuhn. Semantic challenges for sensor plug and play. In *Proceedings of the 9th International Symposium on Web and Wireless Geographical Information Systems (W2GIS '09)*, volume 5886, pages 72–86, 2009.

[11] Arne Bröring, Patrick Maúe, Krzysztof Janowicz, Daniel Nüst, and Christian Malewski. Semantically-enabled sensor plug and play for the sensor web. *Sensors*, 11(8):7568–7605, 2011.

[12] Michael Compton, Cory Henson, Laurent Lefort, Holger Neuhaus, and Amit Sheth. A survey of the semantic specification of sensors. In *Proceedings of the Second International Workshop on Semantic Sensor Networks (SSN09)*, volume 18, pages 17–32, 2009.

[13] Zlatan Dragisic. Semantic matching for stream reasoning. Master's thesis, Linköping University, 2011.

[14] Thomas Dreyer, David Leal, Andrea Schröder, and Michael Schwan. ScadaOnWeb - web based supervisory control and data acquisition. In *Proc. ISWC*. 2003.

[15] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.

[16] Mica R. Endsley. Theoretical underpinnings of situation awareness: A critical review. In *Situation Awareness Analysis and Measurement*. 2000.

[17] Thomas R. Gruber and Greg R. Olsen. An ontology for engineering mathematics. In *Proc. KR*, 1994.

[18] Volker Haarslev, Kay Hidde, Ralf Möller, and Michael Wessel. The RacerPro knowledge representation and reasoning system. *Semantic Web*, 2010.

[19] Fredrik Heintz. *DyKnow : A Stream-Based Knowledge Processing Middleware Framework*. PhD thesis, Linköping University, 2009.

[20] Fredrik Heintz. Semantically grounded stream reasoning integrated with ROS. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE conference proceedings, 2013.

[21] Fredrik Heintz and Daniel de Leng. Semantic information integration with transformations for stream reasoning. In *Information Fusion (FUSION 2013)*. IEEE, 2013.

[22] Fredrik Heintz and Patrick Doherty. DyKnow : An approach to middleware for knowledge processing. *Journal of Intelligent and Fuzzy Systems (JIFS 2004)*, 15(1):3–13, 2004.

[23] 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)*, pages 1063–1069, 2010.

[24] Fredrik Heintz and Zlatan Dragisic. Semantic information integration for stream reasoning. In *Proceedings of the 15th International Conference on Information Fusion (FUSION)*, 2012.

[25] Fredrik Heintz, Jonas Kvarnström, and Patrick Doherty. Bridging the sense-reasoning gap: DyKnow - stream-based middleware for knowledge processing. *Advanced Engineering Informatics*, 24(1):14–26, 2010.

[26] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. OWL 2 web ontology language primer (second edition). 2012.

[27] Anders Hongslo. Stream processing in the Robot Operating System framework. Master's thesis, Linköping University, 2012.

[28] Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for working with OWL 2 ontologies. In *OWLED*, volume 529, pages 11–21, 2009.

[29] Matthew Horridge and Peter F. Patel-Schneider. OWL 2 web ontology language Manchester syntax (second edition). 2012.

[30] Ian Horrocks. Ontologies and the semantic web. *Communications of the ACM*, 51(12):58–67, 2008.

[31] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible SROIQ. In *Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006)*, pages 57–67, 2006.

[32] Ian Horrocks, Peter Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A semantic web rule language combining OWL and RuleML, 2004. Available at `http://www.w3.org/Submission/SWRL/`.

[33] Mieczyslaw M. Kokar, Christopher J. Matheus, and Kenneth Baclawski. Ontology-based situation awareness. In *Proceedings of the Tenth International Conference on Information Fusion (FUSION)*, volume 10, pages 83–98, 2009.

[34] Daniel Lazarovski. Extending the stream reasoning in DyKnow with spatial reasoning in RCC-8. Master's thesis, Linköping University, 2012.

[35] Frank Manola and Eric Miller. RDF primer. 2004.

[36] Christopher J. Matheus, Mieczyslaw M. Kokar, and Kenneth Baclawski. A core ontology for situation awareness. In *Proceedings of the Sixth International Conference on Information Fusion (FUSION)*, volume 1, pages 545–552, 2003.

[37] Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.

[38] Object Management Group. The CORBA specification v 3.1, 2008. Available at `http://www.omg.org/spec/CORBA/3.1/`.

[39] Emad Pejman, Yousef Rastegari, Pegah Majlesi Esfahani, and Afshin Salajegheh. Web service composition methods: A survey. In *Proceedings of the International MultiConference of Engineers*, volume 1, pages 560–564, 2012.

[40] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[41] Shuping Ran. A model for web services composition with QoS. *ACM SIGecom Exchanges*, 4(1):1–10, 2003.

[42] Jinghai Rao and Xiaomeng Su. A survey of automated web service composition methods. In *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC '04)*, volume 3387, pages 43–54, 2005.

[43] H. Rijgersberg, M. Wigham, and J.L. Top. How semantics can improve engineering processes: A case of units of measure and quantities. *Advanced Engineering Informatics*, 25(2), 2011.

[44] Hajo Rijgersberg, Mark van Assem, and Jan Top. Ontology of units of measure and related concepts. *Semantic Web*, 4(1), 2013.

[45] Gunther Schadow, Clement J. McDonald, Jeffrey G. Suico, Ulrich Fähring, and Thomas Tolxdorff. Units of measure in clinical information systems. *Journal of the American Medical Informatics Association*, 6(2), 1999.

[46] Rob Shearer, Boris Motik, and Ian Horrocks. HermiT: A highly-efficient owl reasoner. In *Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2008)*, pages 26–27, 2008.

[47] Amit Sheth, Cory Henson, and Satya S. Sahoo. Semantic sensor web. *IEEE Internet Computing*, 12(4):78–83, 2008.

[48] Amit Sheth and Matthew Perry. Traveling the semantic web through space, time and theme. *IEEE Internet Computing*, 12(2):81–86, 2008.

[49] Yimin Shi, Xiaoping Zhou, and Xianzhong Zhang. Sensor ontology building in semantic sensor web. *Internet of things*, 312(1):277–284, 2012.

[50] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51–53, 2007.

[51] Arthur G. Stephenson, Lia S. LaPiana, Daniel R. Mulville, Peter J. Rutledge, Frank H. Bauer, David Folta, Greg A. Dukeman, Robert Sackheim, et al. Mars Climate Orbiter mishap investigation board phase I report, 1999.

[52] Dmitry Tsarkov and Ian Horrocks. FaCT++ description logic reasoner: system description. In *Proceedings of the Third international joint conference on Automated Reasoning*, IJCAR'06, pages 292–297, 2006.

[53] Kamin Whitehouse, Feng Zhao, and Jie Liu. Semantic streams: a framework for composable semantic interpretation of sensor data. In *Proceedings of the Third European conference on Wireless Sensor Networks (EWSN'06)*, volume 3868, pages 5–20, 2006.

[54] Liangzhou Zeng, Boualem Benatallah, Marlon Dumas, Jayant Kalagnanam, and Quan Z. Sheng. Quality-driven web services composition. In *Proceedings of the 12th international conference on World Wide Web (WWW '03)*, pages 411–421, 2003.