Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Evaluating the use of DyKnow in multi-UAV traffic monitoring applications

by

Tommy Persson

LIU-IDA/LiTH-Ex-A--09/019--SE

2009-03-24

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköping University
Department of Computer and Information Science

Final Thesis

Evaluating the use of DyKnow in multi-UAV traffic monitoring applications

by

Tommy Persson

LIU-IDA/LiTH-Ex-A--09/019--SE

2009-03-24

Supervisor: Fredrik Heintz
IDA, Linköpings universitet

Anders Holmberg
SAAB AB

Examiner: Patrick Doherty
IDA, Linköpings universitet

Abstract

This Master's thesis describes an evaluation of the stream-based knowledge processing middleware framework DyKnow in multi-UAV traffic monitoring applications performed at Saab Aerosystems. The purpose of DyKnow is "to provide generic and well-structured software support for the processes involved in generating state, object, and event abstractions about the environments of complex systems."[10] It does this by providing the concepts of streams, sources, computational units (CUs), entity frames and chronicles.

This evaluation is divided into three parts: A general quality evaluation of DyKnow using the ISO 9126-1 quality model, a discussion of a series of questions regarding the specific use and functionality of DyKnow and last, a performance evaluation. To perform parts of this evaluation, a test application implementing a traffic monitoring scenario was developed using DyKnow and the Java Agent DEvelopment Framework (JADE).

The quality evaluation shows that while DyKnow suffers on the usability side, the suitability, accuracy and interoperability were all given high marks.

The results of the performance evaluation high-lights the factors that affect the memory and CPU requirements of DyKnow. It is shown that the most significant factor in the demand placed on the CPU is the number of CUs and streams. It also shows that DyKnow may suffer dataloss and severe slowdown if the CPU is too heavily utilized. However, a reasonably sized DyKnow application, such as the scenario implemented in this report, should run without problems on systems at least half as fast as the one used in the tests.

Sammanfattning

Den här exjobbsrapporten beskriver en utvärdering av DyKnow i multi-UAV trafikövervakningsapplikationer. DyKnow är ett “knowledge processing middleware framework” vars syfte är “to provide generic and well-structured software support for the process involved in generating state, object, and event abstractions about the environments of complex systems.”[10] DyKnow gör detta genom att tillhandahålla koncept som “streams, “sources,” “computational units (CUs),” “entity frames” och “chronicles.”

Denna utvärdering är delad i tre delar: en generell kvalitetsutvärdering av DyKnow med hjälp av kvalitetsmodellen i ISO 9126-1, en diskussion av en serie frågor rörande användningen av DyKnow och dess funktionalitet och sist, en prestandautvärdering. För att utföra delar av den här utväderingen utvecklades en testapplikation som implementerar ett trafikövervakningsscenario. Till detta användes DyKnow och Java Agent DEvelopment Framework (JADE). Ett kapitel är tillägnat implementeringen av denna testapplikation.

Resultaten av kvalitetsutvärderingen visar att DyKnow lider lite vad gäller användarvänlighet. Däremot fick lämpligheten, noggranhet och interoperabiliteten bra betyg.

Prestandautvärderingen visar vilka faktorer som påverkar DyKnow vad gäller minnes- och processorkrav. Utvärderingen visar att antalet strömmar och CUs är vad som påverkar CPU-belastningen mest. Den visar även att DyKnow riskerar att förlora data och att saktas ned när CPU-belastningen blir för hög. En rimligt dimensionerad applikation, såsom det scenario som implementeras i den här rapporten, bör kunna köras utan problem på maskiner åtminstone hälften så snabba som den som användes i testerna.

Acknowledgments

First, I would like to thank the Decision Support & Autonomous Systems group at Saab Aerosystems for giving me the opportunity to perform this Master's thesis. I would also like to thank my two advisors Anders Holmberg, SAAB and Fredrik Heintz, Linköping University for their help and advice in writing this report. I also thank Patrick Doherty for being willing to be my examiner.

Contents

Chapter 1

Introduction

This report presents the work done in a Master's thesis during the fall of 2008. This introductory chapter describes the goals of the thesis and introduces some of the concepts and terms used throughout the report.

1.1 Background

The Decision Support & Autonomous Systems group at Saab Aerosystems is running a research program in the area of autonomous systems and multi-agent systems. The aim of the research is to develop technologies for nEUROn[1] and Skeldar[2], as well as future platforms. One important part of the research program is to establish collaboration with researchers at Linköping University and to transfer technologies developed there to SAAB.

One such technology is the stream-based knowledge processing middleware framework DyKnow. DyKnow is currently developed by researchers at the Artificial Intelligence & Integrated Computer Systems Division (AIICS) at the Department of Computer and Information Sciences (IDA) at Linköping University, LIU. The main purpose of DyKnow is "to provide generic and well-structured software support for the processes involved in generating state, object, and event abstractions about the environments of complex systems." [10]

DyKnow has been successfully used in several UASTech (Unmanned Aircraft Systems Technology) projects at AIICS on their UAV (Unmanned Aerial Vehicle) platforms. There DyKnow is mainly used to create high level representations of the environment the UAVs inhabits. These are used by for example planning software when generating routes and actions.[9]

The research group at SAAB wishes to determine if DyKnow is suitable for the purpose of achieving joint situational awareness among a group of agents. To evaluate whether it is suitable is the goal of this thesis.

[1]Dassault nEUROn, an experimental Unmanned Combat Air Vehicle (UCAV) developed jointly by several European companies, including SAAB.

[2]A multi-purpose prototype helicopter UAV.

1.2 Goal

The goal of this project is to evaluate the use of the DyKnow middleware to achieve joint situational awareness in applications requiring multiple agents to work together and share information. This evaluation is presented in three parts.

- A performance evaluation. The purpose of this evaluation is to document the behavior of DyKnow when stressed in various ways and to identify potential bottlenecks that might constrain the kinds of systems on which DyKnow may be used.
- A quality evaluation of DyKnow, not focusing on specific features but instead on the general characteristics of DyKnow. The evaluation tries to find the strengths and weaknesses of the current implementation of DyKnow which can be used to highlight specific areas where it can be improved. This evaluation is performed according to the ISO 9126-1 quality model[6].
- By answering the following questions regarding the use and functionality of DyKnow.

  1. How easy is it to ...
     (a) ... implement a chosen information fusion strategy?
     (b) ... use different types of information sources in DyKnow?
     (c) ... integrate DyKnow into an existing environment?
     (d) ... implement the scenario-specific chronicles in DyKnow?
     (e) ... include some kind of operator control in the system?
  2. How can temporary loss of input or communication channels be handled?
  3. How much work is needed to extend a single-UAV scenario to a multi-UAV scenario?

In order to perform this evaluation, a scenario was defined and a DyKnow application implementing it was developed. The details of the scenario are described in the next chapter, but it can be summarized as follows:

Two stationary UAVs are positioned with a mostly disjoint view of a road. It is the job of the UAVs to detect events such as an overtake between two cars traveling on the road. They do this by continually estimating the positions of the two cars. Because the overtake happens so that neither UAV witness the whole event, it is required that the two UAVs share and merge their information about the event they each possess.

The details of the DyKnow application implementing this scenario is given in Chapter 4.

1.3 Multi-agent systems

Before moving on it is useful to understand what a multi-agent system is. But let's first define what an agent is. The following definition is provided in *Introduction to MultiAgent Systems* [14].

> An *agent* is a computer system that is *situated* in some *environment*, and that is capable of *autonomous action* in this environment in order to meet its design objectives.

In the scenario described each UAV can be viewed as an agent. The UAV's environment is the road and the cars that drive on it. The actions it may perform are to monitor the objects on the road for events and to share information with the other UAV.

A *multi-agent system* is a system that contains a number of agents who interact with each other. Each agent has its own area of the environment in which it may act and observe.

1.4 Information and data fusion

To be able to work together agents need to share and combine information from multiple sources. The act of combining information and sensor data from multiple sources, such as systems and agents is typically referred to as "data fusion" or "information fusion."[5] This area has been the subject of much research and in 1985 the Joint Directors of Laboratories (JDL) Data Fusion Working Group developed a functional model for data fusion [13].

A revised version of the model [13] consists of five levels of processing, outlined below.

Level 0 - Sub-Object Data Assessment is the estimation and prediction of object states on the pixel- or signal-level.

Level 1 - Object Assessment refers to the process of estimating the state of an individual object from observations.

Level 2 - Situation Assessment is aimed at developing a description of the different entities and their relationships in their environment.

Level 3 - Impact Assessment tries to use the current situation to predict future threats and opportunities.

Level 4 - Process Refinement refers to the process of monitoring and improving the overall fusion process.

Another level has since been proposed. This new level 5 is called "User Refinement." Its purpose is to "delineate the human from the machine in the process refinement"[4].

1.5 Limitations

There are some important related areas that will not be discussed in this report. One such area is the design of information fusion algorithms. Instead, only the possibility of implementing a chosen fusion algorithm in DyKnow is discussed in the evaluation.

Also, the DyKnow test application is developed and used only on a single computer. Any effects stemming from the use of a network, such as latency or bandwidth is not taken in account in the evaluation. Instead, the theoretical result of latency and bandwidth is discussed briefly.

Another limitation is the decision to use only one pair of UAVs in the test scenario. This was done to simplify the scenario. Despite this limitation, the application and the evaluation are as general as possible.

1.6 Influences

Report on extending DyKnow for Multi-Node Distributed Networks - A Prototype Implementation [7] describes a method of extending DyKnow to allow for multi-agent functionality. This is in fact the approach the was used in this report as well.

A quality framework for developing and evaluating original software components [3] illustrates how the ISO 9126-1 quality model can be used to evaluate original software components. This article served as the main influence on the choice of evaluation method.

1.7 Outline

The disposition of the rest of the report is as follows:

Chapter 2 describes the details of the test scenario.

Chapter 3 gives an overview of DyKnow, its architecture, its functionality and its use.

Chapter 4 discusses the implementation details of the DyKnow application implementing the test scenario.

Chapter 5 presents the results of the general evaluation and discusses the questions regarding the functionality and use of DyKnow.

Chapter 6 provides the results of the performance evaluation and discusses conclusions made from them.

Lastly, Chapter 7 summarizes the report.

Chapter 2

Scenario

To evaluate a middleware framework such as DyKnow, it is suitable to develop a scenario that properly addresses the concerns that the reviewing party wishes to focus on. The requirements of the scenario as well as the chosen specification used in this evaluation is described below.

2.1 Requirements

The requirements placed on the scenario is that it must describe a series of events that needs to be recognized. Recognition of these events must require the cooperation of multiple agents.

2.2 Specification

The chosen scenario consists of two stationary UAVs. The field of view of the UAVs are disjoint except for a small common patch between them (2.1). Their purpose is to track objects (cars) moving on the road and to monitor events such as overtakes between cars. An overtake in this scenario is divided into the three events, a car being at first behind, then beside and finally ahead of the other car.

These events will be distributed over the two UAV:s respective field of view. The UAV:s will need to share and fuse information to be able to monitor events happening on the border between their fields of view. The scenario is illustrated in Figure 2.1 on the next page.

This scenario provides the foundation for the evaluation of DyKnow presented in Chapter 5.

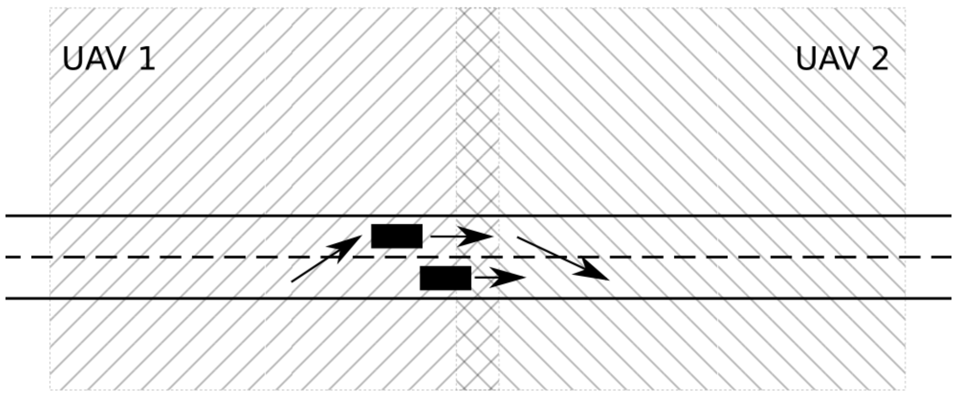

Figure 2.1. An illustration of the scenario as outlined in Section 2.2

Chapter 3

DyKnow

Before moving on to the implementation of the test software it is important to understand how DyKnow works and what features exist. This chapter will describe the relevant concepts in DyKnow and how they may be used to enable a developer to write a high-level knowledge processing application.

For the interested party, a complete description of DyKnow is given in *DyKnow: A Stream-Based Knowledge Processing Middleware Framework* [8].

3.1 Concepts

The main concepts that an application developer using DyKnow will need to understand are *streams*, *sources*, *computational units* (CUs), *entity frames* and *chronicles*. Used together these elements form a complete knowledge processing application.

- Streams, or as they are also called, fluent streams are streams of *samples*. Each sample in a stream contains a value and three different time points: the creation time, the time point at which the sample is valid and the time of the query which created the sample. In the latest version of DyKnow a sample has two time points: the valid time and the available time, where the available time is the time when the sample is available to the receiver. An example stream would be a series of values representing measurements from a sensor.
- A source is a process capable of generating a stream of samples.
- Computational units or CUs for short, are processes that take an arbitrary number of streams as input and then outputs new samples into another stream. The computational unit is automatically called each time a new sample is available in one of the streams it subscribes to. A common use case is to use computational units as filters.
- Entity frames are user-defined types that can be used as values in samples. These entity frames are similar to C-type structs and may contain any type

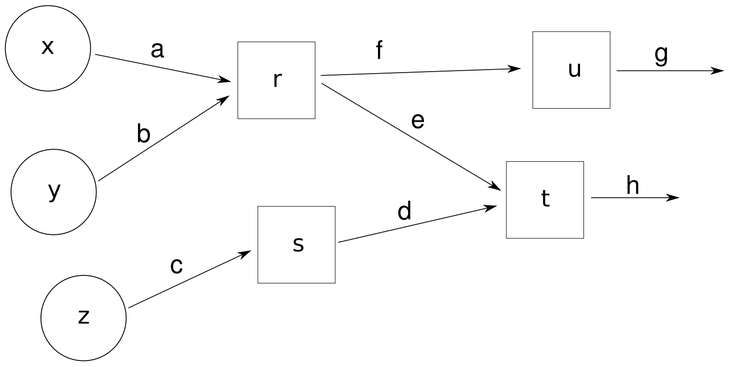

Figure 3.1. An example DyKnow application showing how sources (circles), streams (arrows) and computational units (boxes) may be connected.

of values allowed by CORBA[1], including user-defined IDL-types[2] and other entity frames.

- Chronicles describe a specific series of events with metric temporal constraints on the event occurrences that should be recognized. An event in this case might be the changing of a value in the samples of a stream from "true" to "false" (or any other value for that matter). For example, a `left_of` attribute signifying if a car is to the left of another car might shift from "false" to "true" from one sample to the next.

Figure 3.1 shows how multiple sources, streams and computational units can be connected to form a DyKnow application.

3.2 Architecture

This section describes the major components in DyKnow and their roles. The different components are called the Dynamic Object Repository, the Symbol Manager, the Chronicle Recognition Engine, the Time Server and the Alarm Server. These components are all implemented as CORBA servers that an application programmer can interact with.

3.2.1 CORBA

CORBA is a standard defined by the Object Management Group (OMG) that provides an architectural framework for developing distributed systems.

The major features of CORBA are [11]:

[1]Common Object Request Broker Architecture. described in Section 3.2.1.
[2]Interface Definition Language. A language used to define CORBA objects.

The OMG Interface Definition Language, IDL, is used for defining object interfaces independent of any particular programming language or platform.

Language mappings specify how the different IDL constructs are implemented in a specific programming language. In C++, interfaces are mapped to classes and operations are mapped to class member functions.

Operation invocation and dispatch facilities allow the invocation of requests on remote CORBA objects.

Object adapters allow a user to interact with an object without knowing its exact type.

Inter-ORB Protocols provide a standardized way for ORBs[3] developed by different vendors to communicate. Specified protocols include GIOP (General Inter-ORB Protocol) and IIOP (Internet Inter-ORB Protocol).

The CORBA specification also defines a number of services [11] that an ORB-vendor must implement.

One of the standardized CORBA services available is the OMG Naming Service. DyKnow makes extensive use of this service. The Naming Service is basically a directory where one can register CORBA servers. This directory can then be browsed easily and references to the registered CORBA servers may be retrieved by name instead of long and complicated URI[4] strings.

Also provided is the OMG Event Service which provides a decoupled communications model that can be used instead of relying on strict client-server synchronous request invocations. The model defines *suppliers* that produce events that *consumers* receive. The suppliers and consumers communicates through an *event channel* that they they must register to. Using the event channel many-to-many relationships between consumers and suppliers are possible. This event system is used much inside DyKnow to notify different subsystems that there is processing to be done. It is possible for a DyKnow application programmer to listen in on all these internal events but it is usually not necessary since defining streams and computational units is often enough.

3.2.2 The Dynamic Object Repository

The Dynamic Object Repository or DOR is a soft real-time database that is used to store all samples processed by DyKnow. The DOR is available as a CORBA service (`dor`). A programmer uses the DOR to add computational units and primitive streams to the system. It is then possible to reference the CUs when adding new streams that connect them.

[3]Object Request Broker. The central component of any CORBA implementation.
[4]Uniform Resource Identifier.

3.2.3 The Symbol Manager

The symbol manager acts as a look-up table in DyKnow. It is through the symbol manager that one can create, replace, retrieve or remove descriptors used to refer to streams and computational units. The symbol manager is exposed to the programmer as the CORBA service `symbol_manager`.

3.2.4 The Chronicle Recognition Engine

Chronicle recognition in DyKnow is provided by a piece of software called the chronicle recognition system (CRS) developed by France Telecom. DyKnow wraps this system with its own CORBA server (`cre`).

Using this wrapper it is simply a matter of calling a method to register a pre-compiled chronicle in the system. The CRE will then output successfully recognized chronicle instances in the stream `chronicles`. It is possible to set up a separate event channel to listen to recognized chronicles.

3.2.5 The Time and Alarm servers

The time server provides a simple interface to get the global time in a DyKnow application. This can be useful when the application is distributed on multiple computers.

Using the alarm server it is possible to schedule any kind of task by defining a TimerCallback-object. This object will need a start time, a stop time, a delay time, and a time value for how much time should pass between each call to the callback.

3.3 Using DyKnow

3.3.1 System requirements

DyKnow is written in C++ on top of ACE[5] and TAO[6]. The version of DyKnow used in this report relied on version 1.4a of the OCI[7] distribution of TAO although it is possible other versions might work as well. Other dependent libraries include Boost.

No specific requirements on CPU speed or memory have been determined, but some insight might be learned from Chapter 6 where the performance of DyKnow is evaluated.

[5]ADAPTIVE Communication Environment. “A freely available, open-source object-oriented (OO) framework that implements many core patterns for concurrent communication software.”[12]

[6]The ACE ORB. A real-time CORBA implementation built upon the ACE framework.

[7]Available at http://www.theaceorb.com/downloads/1.4a/index.html

3.3.2 Defining sources

A source in DyKnow is simply a function that returns a sample when asked. The following piece of C++ code shows the type definition of the function.

```
typedef boost::function<Idl::Dyknow::Sample*
                        (const Idl::Time&, const std::string&)>
        GetSample
```

This source function is then wrapped in a structure called "SensorInterfaceWrapper" before it can be added to the DOR and made available for creation of named streams.

3.3.3 Defining streams

There are a number of ways to add a stream to a DyKnow application, depending mostly on which type of stream one wants to add.

Firstly, computed streams are streams that receive their samples from a computational unit. Secondly, a named stream is connected to a source. It pulls samples from the source it references with a given sample rate. Lastly, there is the primitive streams. These streams are not connected to anything else. Samples are explicitly added by any component in the system.

The different types of streams are all created in a similar way. First a *policy* is defined, determining the type of the stream and what values may be placed in it as well as various *constraints* that can be placed on the stream. Using this policy object a descriptor is created by the symbol manager.

The constraints that can be placed on a stream include:

Duration Constraints Specifies a time interval within which samples should be valid.

Sample period The difference in valid time between two consecutive samples.

Cache Constraint There are two fundamental cache constraints, the Time Cache Constraint which specifies how long a sample may be stored and the Size Cache Constraint which determines how many samples may be stored simultaneously.

Temporal Constraint (Also know as Order Constraint) Specifies how the samples should be ordered with regard to their valid times, while still being ordered by the time they were made available. Options include any order, monotone (increasing), and strict monotone (increasing, with no samples with the same time point).

Delay Constraint Specifies the maximum difference between the available time and the valid time of a sample.

These constraints are all declarative, they specify conditions that a stream should satisfy. It is up to DyKnow to satisfy them.

3.3.4 Defining computational units

Computational units are any function objects implementing either of the following `operator()`-methods:

```
Idl::Dyknow::Sample*
  operator()(const Idl::Dyknow::SampleSeq& input,
             const Idl::Time query_time)

Idl::Dyknow::SampleSeq*
  operator()(const Idl::Dyknow::SampleSeq& input,
             const Idl::Time query_time)
```

The second version is needed when a computational unit needs to return multiple samples in the same function call.

A computational unit is added to the DOR in much the same way that a source is, using a wrapper object. In this case the computational unit is wrapped by a `CompUnitWrapper` object before it is passed on to the DOR. To actually use the CU it is also necessary to set it up with a policy and create a descriptor in the symbol manager.

3.3.5 Defining chronicles

Chronicles are defined in a special language specifically created for that purpose [8]. To define a chronicle it is first required to define the *attributes* which the chronicle will have observations about. The different values of an attribute must be defined by a *domain*.

To illustrate, a very simple chronicle may look like the one below:

```
domain Boolean = {true, false, unknown}
domain Objects = ~{}

attribute some_stream.is_large[?object]
{
  ?object in Objects
  ?value in Boolean
}

chronicle is_large_chronicle[?object]
{
  event(some_stream.is_large[?object]:(false,true), t1)
  noevent(some_stream.is_large[?object]:(true,false), (t1+1, t2))

  t1 < t2
  t2 - t1 in [100, 100]
}
```

This chronicle will check if the attribute `is_large` in the stream `some_stream` is ever set to true and then stays true for 100 time units. The `stream.attribute` naming convention used above is not something natively supported by the chronicle recognition system. It is up to DyKnow to serve CRS with the appropriate information.

Any recognized chronicle is added to the `chronicles` stream which CUs or others can then read.

3.3.6 Defining entity frames

As noted before, it is possible to use any CORBA-compatible value type as a value in a sample. That includes strings, integers, booleans and other simple types. It is also possible to define sequences and aggregations of the available types. User-defined CORBA IDL-types can also be used.

DyKnow provides an object type called EntityFrame that may contain a sequence of attribute-value pairs of many different types. To simplify their usage it is possible to write definition files that are then compiled into full C++ CORBA-compatible classes ready for use in streams and computational units. This allows a user to simply call "obj.GetMyAttribute()" instead of manually traversing the list of all attributes.

A small definition file showing how an EntityFrame might be constructed is displayed below:

```
MyEntityFrameType MY_ENTITY_FRAME_TYPE
long attrib1  default_value1
double attrib2  default_value2
bool attrib3  default_value3
EntityFrame MyOtherEntityFrameType Idl::Dyknow::EntityFrame()
```

3.4 Summary

This chapter has given an introduction to DyKnow, its concepts, architecture and feature set. It has described streams and how they relate to samples, sources and computational units. An introduction to the basic syntax and functionality of chronicles has also been given. With this information it is time to move on to describing a test application implementing the scenario outlined in the previous chapter. This is done in the next chapter.

Chapter 4

Implementation

This chapter describes the implementation details of a DyKnow application that successfully implements the scenario outlined in Chapter 2. The work that went into creating this implementation serves as the basis for the evaluation described in Chapter 5.

The chapter begins with a section dedicated to a description of how DyKnow can be extended to enable information sharing between multiple instances. In Section 4.2 the actual scenario implementation is discussed. Finally in Section 4.3 the implementation of a multi-agent environment using JADE[1] is discussed.

4.1 Extending DyKnow for information sharing

As the version of DyKnow available to the author doesn't offer any support for multi-agent environments it is necessary to extend it with this functionality. One such approach is to use import and export proxies registered in each DyKnow-instance[7]. This is also the approach that was chosen in this implementation and it is described further below.

4.1.1 Import and export proxies

The job of the two proxies is to provide an interface for exporting streams from one DyKnow-instance to another. The proxies are called ImportProxy and ExportProxy and are provided as CORBA-services in each DyKnow-instance. One may then simply request the export proxy from another DyKnow-instance and call the appropriate methods to begin exporting a stream. To differentiate between the two active DyKnow-instances, they are each given a unique unit number on start-up.

The methods provided by the ExportProxy service are the following:

`start_exporting(from_semantic_label, to_semantic_label, unit)` Calling this method will begin exporting the stream referred to by `from_semantic_label`

[1]Java Agent DEvelopment Framework. A middleware for developing multi-agent systems.

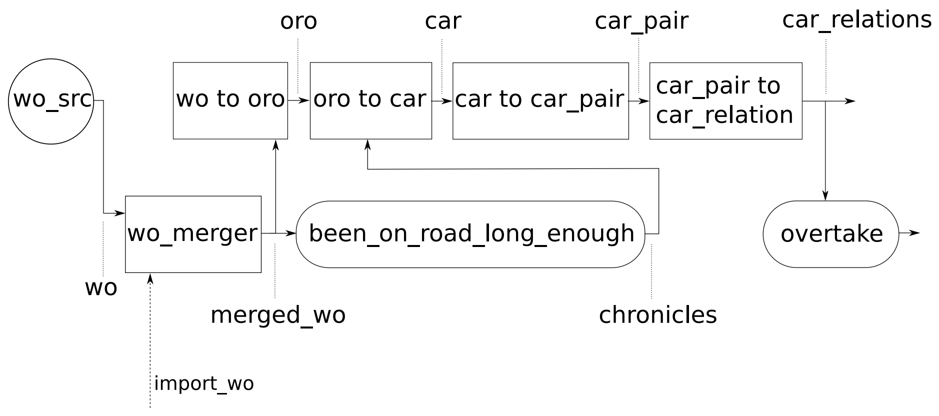

Figure 4.1. An overview of the different parts in the scenario implementation and how they are connected. Rectangles represent computational units while rounded rectangles are chronicles. The only source in the system is shown as a circle.

on the DyKnow-instance providing the export proxy to the stream referred to by `to_semantic_label` on the DyKnow-instance with the unit number of `unit`. A semantic label is the name given to each stream when they are created.

To actually export the stream, the export proxy must create a subscription to the stream it will export. The export proxy then calls a callback function every time a new sample enters the stream. In this case the purpose of the callback is to call the `push` method (described below) of the import proxy on the receiving unit.

`stop_exporting(semantic_label, unit)` This method stops the exportation of a stream. This is as simple as canceling the subscription made to the stream that was being exported.

The ImportProxy service provides the following method:

`push(unit, semantic_label, sample)` This method is called when a DyKnow-instance with the unit number of `unit` wants to export a `sample` to a stream labeled `semantic_label`.

4.2 Implementing the scenario

To implement the scenario described in Section 2.2 it is necessary to use all the main features of DyKnow: Streams, computational units and chronicles.

The source of all the samples is a pair of logfiles. One for each simulated UAV.

```
mSymbol car("car");
mSymbol car_pair("car_pair");
mSymbolSeq car_pair_args(car);
mComputedFstreamPolicy car_pair_pol(dor_service_name,
                                    car_to_car_pair_cu,
                                    car_pair_args,
                                    fstream_constraints);
create_descriptor(symbol_manager.in(),
                  car_pair,
                  mPolicy(car_pair_pol),
                  CarPairWrapper::type(),
                  symbol_already_exist_policy,
                  exception_policy,
                  verbose);
```

Listing 4.1. The code used to setup the `car_pair` stream.

4.2.1 Fluent streams

These are the different fluent streams used in the traffic monitoring application:

`wo` This stream contains all the samples polled from the source.

`import_wo` The stream where all samples imported from the other agent are made available.

`merged_wo` This stream contains all the samples merged by the `wo_merger CU`.

`oro` This stream contains all the world object samples determined to be on a road.

`car` This stream contains all the on road objects determined to be cars.

`car_pair` This stream contains all the different combinations of car pairs.

`car_relations` This stream contains all car pairs that have been given attributes (by the preceding CU) such as `left_of`, if the first car in the pair is left of the other, and vice versa for `right_of`.

To illustrate how one may add a new stream in DyKnow, the code responsible for setting up the `car_pair` stream is given in Listing 4.1.

4.2.2 Sources

The only source of samples in this application is `wo_src`. This source reads samples from a prepared logfile. It is polled every 100 milliseconds by the `wo` stream.

4.2.3 Computational Units

This section describes all the different computational units developed to implement the scenario. In Figure 4.1 we can see how they are connected.

`wo_merger` The purpose of this computational unit is to merge any two streams of WorldObjects. Presently, it is connected to the two streams `wo` and `import_wo`. This is where one would implement a potential fusion algorithm in the system. For now, the CU simply interleaves the two streams to combine them, or if the CU is told that two WorldObjects are equivalent, renames one of them.

`wo_to_oro` Originally intended to reason about whether a world object is on a road and if so, convert them, this CU now trusts the `on_road_system` flag of the world object samples. The assumed image-processing system which generates the samples is thereby trusted to see if a world object is on a road. If so, the sample is passed onto the `oro` stream as an `oro` object.

`oro_to_car` This computational unit reasons whether a given road object is in fact a car. It does this by listening to the `oro` and `chronicles` streams. It searches for a specific recognized chronicle showing that a road object has been on a road for a given period of time. When this happens, the CU wraps the sample in a `car` structure and adds it to the `car` stream.

`car_to_car_pair` One of the more complicated computational units. It is responsible for preparing pairs of car samples which other CUs may reason more about later.

The CU does this by keeping a list of all the latest samples from each car received in its internal memory. Every time a new sample arrives the previous one from the same car is removed (this assumes that samples of the same car object are received in the correct order, which can be specified with an order constraint as described in Section 3.3.3), after which a list of all pair combinations of this new car sample and all others stored are sent as outputs.

If a stored sample is stored for too long, it is automatically thrown away. This is to prevent meaningless comparisons by other CUs.

`car_pair_to_car_relations` This computational unit calculates the relation between the two cars in each sample. It calculates if the first car is `behind`, `in_front`, `left_of` or `right_of` the other. This information is then used by the `overtake` chronicle (described below) to recognize if one car has overtaken another.

4.2.4 Chronicles

The `been_on_road_long_enough` chronicle is intended to detect if an `on_road_object` has been seen moving on a road long enough to be classified as a car. If so the `oro_to_car` CU should convert the `oro` object into a `car` object.

```
chronicle overtake[?car1, ?car2]
{
  event(car_relations.behind[?car1, ?car2]:(false,true), t1)
  event(car_relations.left_of[?car1, ?car2]:(false,true), t2)
  event(car_relations.in_front[?car1, ?car2]:(false,true), t3)

  t1 < t2 < t3
}
```

Listing 4.2. A simplified chronicle responsible for recognizing overtakes.

The `overtake` chronicle can be implemented as shown in Listing 4.2, while the full code necessary to compile this chronicle with all the necessary attributes and variable domains is given in Listing B.2.

4.3 Enabling multi-agent behavior with JADE

In order to test the DyKnow application as well as evaluating the possibilities of integrating support for operator-control in DyKnow, an agent and a operator has been written using JADE.

JADE (Java Agent DEvelopment Framework) is a Java middleware aiming to simplify the development of multi-agent systems [1]. Agents developed with JADE all run on the JADE Agent Platform which complies with the FIPA[2] specification and agent-to-agent interaction are done through the FIPA-ACL language.

A running JADE platform contains at least the Main Container in which the AMS (Agent Management System), DF (Directory Facilitator) and (optionally) RMA (Remote Monitoring Agent) agents reside.

- The AMS is the authoritative agent in its platform. It is the only agent capable of creating or killing agents. It may also remove agent containers and shutdown the agent platform.
- The DF has a yellow pages service which it uses to advertise the services of the agents in the platform to other agents in the same or other platforms.
- The RMA GUI allows an administrator to remotely or locally view all the platforms and their containers and agents. It is also possible to launch, stop and send messages to agents. Agents may also be moved to other platforms and containers seamlessly.

In order for an agent to be able to perform several tasks concurrently JADE agents may have a set of user-defined behaviors. These behaviors are then per-

[2]Foundation for Intelligent Physical Agents. An IEEE Computer Society standards organization that promotes agent-based technology and the interoperability of its standards with other technologies[2].

formed by a scheduler contained in each agent. An example behavior may be to listen to incoming messages from other agents and then react to them.

4.3.1 DyKnow agents

In this agent environment, each DyKnow-instance is wrapped by a JADE-agent called DyKnowAgent.

When the agent starts up, its first task is to launch all the required DyKnow executables, such as the DOR, the CRE and the symbol manager. During the start-up phase all these processes are monitored until they are successfully launched. At that time an agent behavior (described below) changes the state of the agent and notifies the operator.

JADE/DyKnow communication

Communication with the DyKnow-instance is done through a network socket which is connected during the entire run-time of the agent. It is through this socket that the DyKnow-agent may communicate with and control its ImportProxy and ExportProxy. The interface between the DyKnow-instance and the DyKnow-agent is named the JADE/DyKnow interface in Figure 4.2.

Another solution to enable communication between JADE and DyKnow is to use a Java CORBA implementation such as JacORB[3] to reference the two objects directly, but using network sockets was deemed as simpler and faster to implement.

Behaviors

The DyKnow-agent has two defined behaviors: One to listen to incoming requests, the MessageRequestReceiverBehavior and another to monitor and change its state, the StateChangeBehavior.

The MessageRequestReceiverBehavior listens to requests from the operator to either shut down the agent, to run its setup routine or to start or stop exporting streams to other agents. The request messages are composed in a custom FIPA-ACL ontology.

The StateChangeBehavior monitors the agent's state and notifies the operator-agent when it changes. The operator is notified when the DyKnow-agent has finished running its initial loading routines and later also when its setup task has been completed.

GUI

Each agent has a simple GUI attached to them. From it, it is possible to send raw string commands through the network socket (for debugging purposes) and also to run the agents setup routine.

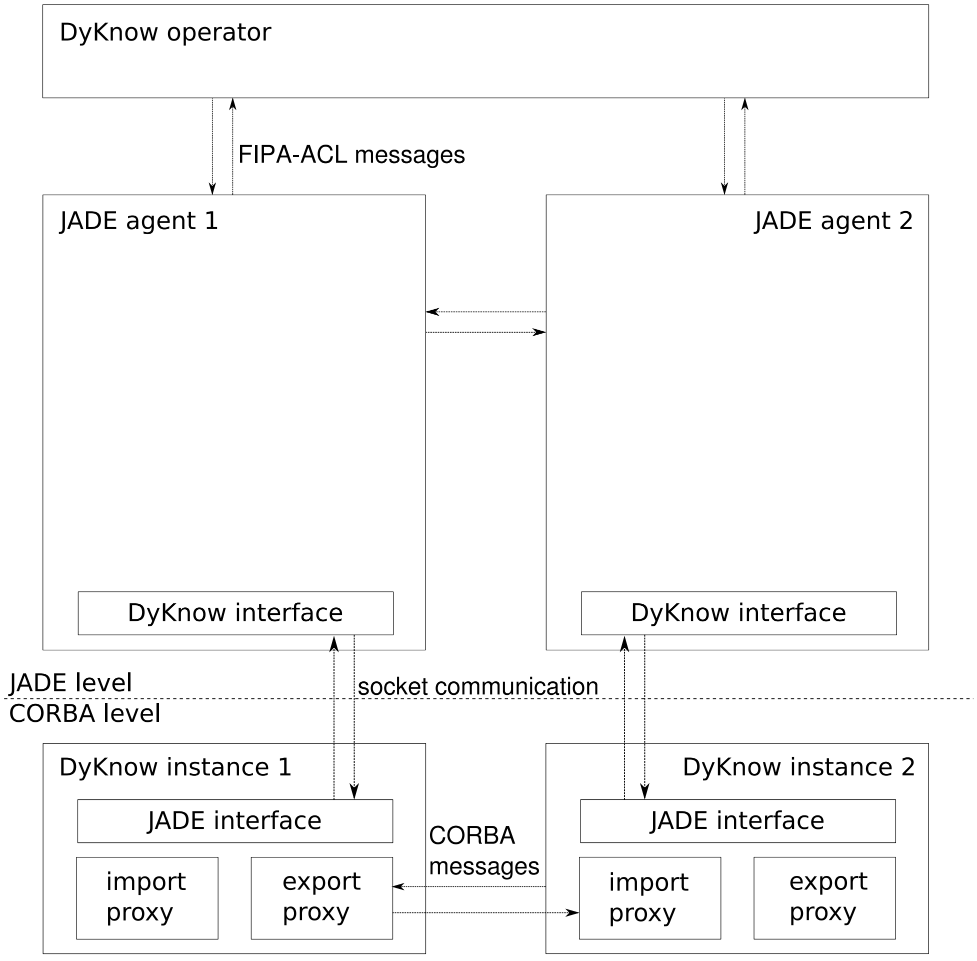

Figure 4.2. An architectural overview of the implementation described in this chapter. Shown are the various interfaces and communication pathways.

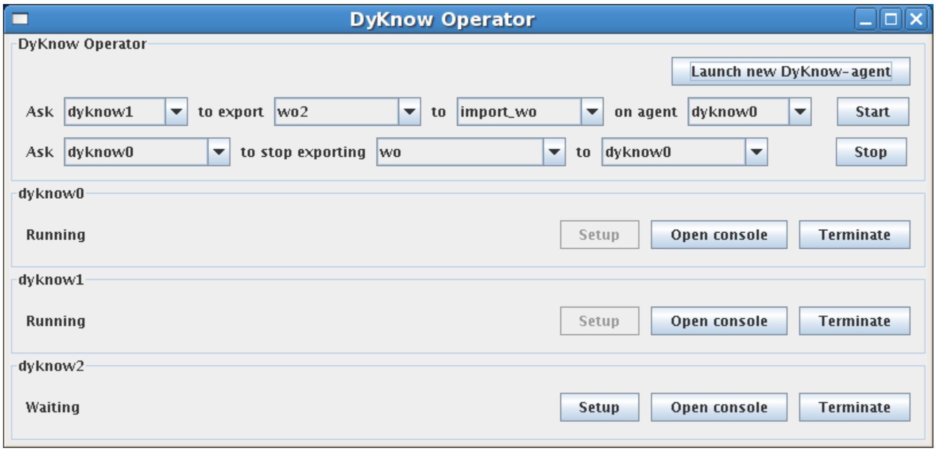

Figure 4.3. A screen shot of a running session of the DyKnow operator. In this particular run three DyKnow-agents have been launched. Two of them have started running while the third just finished its initialization. One of the running agents (`dyknow1`) has also been told to export all samples in its `wo2` stream to the `import_wo` stream of `dyknow0`.

4.3.2 Operator agent

The operator has the ability to start and stop agents, as well as the ability to call their import and export proxies directly to begin a stream export.

Apart from its own initialization the operator is also responsible for setting up the crucial CORBA Naming Service process. It is through the naming service that DyKnow-instances can browse and subscribe to the CORBA services of other instances.

Like the agents it cares for, the operator agent has a behavior defined for listening to incoming messages, MessageReceiveBehavior. This behavior listens for status updates from the running agents.

Agents may be launched by the user by clicking the “Launch new DyKnow-agent” button in the operator GUI (see Figure 4.3). From this GUI it is also possible to terminate the execution of an agent and to run their setup routines (from the similarly labeled buttons).

There is currently no support for viewing the contents of the streams in the individual agent.

If a user were to exit the operator it will automatically send messages to any active agents telling them to shut down, ensuring that no processes are left running.

[3] Availabe from http://www.jacorb.org

4.4 Summary

This chapter has described the implementation of a DyKnow application implementing the scenario described in Chapter 2. The application consisted of a number of streams, computational units, sources and chronicles which brought the abstraction level of the world object samples up to cars and the relationship between pairs of cars. A chronicle was finally able to determine if one car had overtaken another.

To allow information sharing between the two DyKnow instances representing the UAVs of the scenario, a pair of export and import proxies were created. Using these it was possible for one instance to subscribe to the observations of the other.

JADE was used to write a simple wrapper agent around DyKnow and also a separate operator agent. Using the operator it was easy to start and stop DyKnow-agents and to tell one to export a specific stream of information to the other. Communication between the JADE agents and the DyKnow-instance was facilitated using a simple network socket. That said, a CORBA connection would probably have been safer and more preferable to use as it enables tighter integration without the extra layer of command parsing forced by the network socket.

Chapter 5

Framework evaluation

This chapter discusses the results of the evaluation performed. The evaluation was performed by implementing the scenario as described in Chapter 4. Many of the results and answers below will therefore refer back to the implementation.

5.1 Metrics

The chosen metrics include judging the quality and functionality of the software in general terms as well as with regards to the specific scenario.

The quality model used in the first part of the evaluation is from the international standard ISO 9126, described in Section 5.1.1. This model was selected because it defines a number of formal and standardized evaluation characteristics.

The results of the quality evaluation is described in Section 5.2 and mainly concerns the external quality (as defined by the model) of DyKnow. The viewpoint in the evaluation is that of an application developer using DyKnow.

Later in this section the questions that were raised regarding the specific use of DyKnow are listed. Unlike the general software metrics, these will not be graded. Instead they will be subject to more in-depth discussion.

5.1.1 The ISO 9126-1 quality model

ISO 9126 is an international standard for the evaluation of software [6]. The standard is divided into four parts which addresses, respectively, the following subjects: Quality model, external metrics, internal metrics and quality in use metrics. This evaluation will only use the quality model.

ISO 9126-1 defines a quality model in terms of internal quality, external quality and quality in use. Internal quality is evaluated using internal attributes of the software, for example design modularity and compliance with coding standards. External quality is evaluated when the software is executed, typically during formal testing activities. This evaluation will focus on the external quality since DyKnow is distributed as a set of libraries and executables, not as source code.

The rest of this section is a direct quote from ISO 9126-1 [6] describing all the quality characeristics.

Functionality – The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions.

Suitability The capability of the software product to provide an appropriate set of functions for specified tasks and user objectives.

Accuracy The capability of the software product to provide the right or agreed results or effects with the needed degree of precision.

Interoperability The capability of the software product to interact with one or more specified systems.

Security The capability of the software product to protect information and data so that unauthorized persons or systems cannot read or modify them and authorized persons or systems are not denied access to them.

Functionality Compliance The capability of the software product to adhere to standards, conventions or regulations in laws and similar prescriptions relating to functionality.

Reliability – The capability of the software product to maintain a specified level of performance when used under specified conditions.

Maturity The capability of the software product to avoid failure as a result of faults in the software.

Fault Tolerance The capability of the software product to maintain a specified level of performance in cases of software faults or of infringement of its specified interface.

Recoverability The capability of the software product to re-establish a specified level of performance and recover the data directly affected in the case of a failure.

Reliability Compliance The capability of the software product to adhere to standards, conventions or regulations relating to reliability.

Usability – The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions.

Understandability The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use.

Learnability The capability of the software product to enable the user to learn its application.

Operability The capability of the software product to enable the user to operate and control it.

- **Attractiveness** The capability of the software product to be attractive to the user.
- **Usability Compliance** The capability of the software product to adhere to standards, conventions, style guides or regulations relating to usability.

Efficiency – The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions.

- **Time Behaviour** The capability of the software product to provide appropriate response and processing times and throughput rates when performing its function, under stated conditions.
- **Resource Utilisation** The capability of the software product to use appropriate amounts and types of resources when the software performs its function under stated conditions.
- **Efficiency Compliance** The capability of the software product to adhere to standards or conventions relating to efficiency.

Maintainability – The capability of the software product to be modified. Modifications may include corrections, improvements, or adaptation of the software to changes in environment, and in requirements and functional specifications.

- **Analysability** The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.
- **Changeability** The capability of the software product to enable a specified modification to be implemented.
- **Stability** The capability of the software product to avoid unexpected effects from modifications of the software.
- **Testability** The capability of the software product to enable modified software to be validated.
- **Maintainability Compliance** The capability of the software product to adhere to standards or conventions relating to maintainability.

Portability – The capability of the software product to be transferred from one environment to another.

- **Adaptability** The capability of the software product to be adapted for different specified environments without applying actions or means other than those provided for this purpose for the software considered.
- **Installability** The capability of the software product to be installed in a specified environment.
- **Co-existence** The capability of the software product to co-exist with other independent software in a common environment sharing common resources.

Replaceability The capability of the software product to be used in place of another specified software product for the same purpose in the same environment.

Portability Compliance The capability of the software product to adhere to standards or conventions relating to portability.

5.1.2 DyKnow-specific evaluation metrics

Listed below are the questions that were raised with regards to DyKnow in the beginning of this thesis project.

1. How easy is it to ...
   - (a) ... implement a chosen information fusion strategy?
   - (b) ... use different types of information sources in DyKnow?
   - (c) ... integrate DyKnow into an existing environment?
   - (d) ... implement the scenario-specific chronicles in DyKnow?
   - (e) ... include some kind of operator control in the system?
2. How can temporary loss of input or communication channels be handled?
3. How much work was needed to extend the single-UAV scenario to the multi-UAV scenario?

5.2 ISO 9126-1 evaluation results

5.2.1 Functionality

Suitability High

The scenario requires the continuous reading of input samples and detection of different events. DyKnow, with its concept of sources, streams and chronicles fits these requirements nicely. A programmer is able to focus on writing code for the scenario-specific data structures and how to process them. While DyKnow manages the actual distribution of data around the system.

Accuracy High

DyKnow performed well in all tests.

Interoperability High

DyKnow is implemented using CORBA, which provides an excellent platform for integrating components distributed in a network and implemented using different programming languages. Any application may interact with DyKnow as long as they have access to the definitions of the appropriate interfaces. The publish and subscribe communication used by DyKnow is implemented using a notification channel which can also be accessed by any application.

Security Low
The interfaces and the notification channel that DyKnow uses are available to anyone, without any kind of authentication. Although good for interoperability, it makes the system inherently insecure.

Functionality Compliance Not Applicable
No requirements has been set regarding any business rules or other standards.

5.2.2 Reliability

Maturity Medium
Under normal system load the failure rate of DyKnow is very low. No sudden crashes or errors have been noted.

Fault tolerance Medium
In the event that either the DOR, symbol manager, or the alarm server fail, the system will no longer be able to continue processing. No recovery is possible without performing a restart. If the time server fails the system will only temporarily stop processing until the time server is restarted. Should the CREshutdown, it is possible to restart it, but previously registered chronicles must be re-registered. Finally, the failure of the CORBA naming service does not affect the running of DyKnow after the different servers have already retrieved references to each other. Recovery of the naming service requires that each server re-registers itself with it.

Recoverability Medium
As discussed in the previous item.

Reliability Compliance Not applicable
DyKnow is not subject to any standards or regulations regarding reliability.

5.2.3 Usability

Understandability Medium
The internals of DyKnow are hidden from application developers. While the high level functionality of the framework are fairly straight forward and described in a number of articles, detailed documentation regarding actual usage is missing.

Learnability Low
Due to an almost complete lack of documentation, it is hard to get started with DyKnow as a programmer. This also makes it harder to learn how to use the full functionality of the framework. However, a couple of examples were provided which made getting started much easier.

Operability Medium
A DyKnow application is divided into a number of different executables including the DOR, the symbol manager and the chronicle recognition engine.

All of these programs must be started prior to the DyKnow client application. For this purpose, a shell script is used to simplify the process.

Attractiveness Not applicable
DyKnow is a software framework, not an end user application.

Usability Compliance Not applicable
DyKnow is not subject to any standards or regulations regarding usability.

5.2.4 Efficiency

The space and time efficiency is discussed in detail in Chapter 6.

5.2.5 Maintainability

This evaluation is performed in the view of an application developer using a binary DyKnow distribution. As such it is not applicable to comment on the maintainability of DyKnow itself.

5.2.6 Portability

Adaptability Low
DyKnow is not distributed with the source code, this severely limits the adaptability without the help of the original programmers.

Installability Medium
DyKnow relies on a number of different third party libraries and programs, all of which are freely available. Under the assumption that these prerequisites are installed, the installation of DyKnow itself only requires the extraction of an archive.

Co-existence High
DyKnow does not require exclusive access to any specific system resources.

Replacability Low
DyKnow is a complex and specialized system, replacing it in an application would be non-trivial.

Portability compliance Not applicable
DyKnow is not subject to any requirements regarding conformance to portability standards. At the same time, it is built upon the TAO+ACE CORBA middleware and is therefore portable to any platform that is supported by that middleware.

5.3 DyKnow-specific evaluation metrics discussion

5.3.1 How easy is it to implement a chosen fusion strategy?

A fusion algorithm is best implemented in a CU because of the fact that they can be implemented as normal C++ classes, with a defined interface. A very simple algorithm may be to interleave two input streams.

The strategy used in the implementation discussed in Chapter 4 is also very simple. When one agent detects that is no longer receiving observations of a car, it begins to translate the car IDs of the samples imported from the other agent into the same ones used by its own samples. This is only possible because the two agents use the same coordinate system and because the receiving agent somehow knows which imported car IDs matches its own. A real attempt at information fusion is obviously much more complex.

An implementation must not, however, be too taxing on the computer. As shown in Chapter 6, when the processor nears 100% utilization the different threads in the application may be starved and performance and correctness will suffer greatly as a result.

Another thing to take into account is the speed of the fusion algorithm. If each sample is read from the different streams every 100 milliseconds the CU should not take any longer, on average, to process the sample or the rest of the system might suffer a delay.

5.3.2 How easy is it to use different types of information sources in DyKnow?

As described in Section 3.3.2 DyKnow makes no assumptions regarding what kind of information sources are used. All DyKnow needs is a function object capable of returning a pointer to a sample.

```
typedef boost::function<Idl::Dyknow::Sample*
                    (const Idl::Time&, const std::string&)>
        GetSample
```

This function object is then wrapped by a `SensorInterfaceWrapper` which is then registered in the DOR.

Later, when setting up the policy for the stream which will read from this source, it is possible to set a number of constraints to adapt the stream to the particular nature of the source. These constraints are described in Section 3.3.3.

A value in a sample may be of a number of different types, including integers, strings, booleans and entity frames. Sequences and aggregations of all types are also possible. When creating a sample in DyKnow it is necessary to set a parameter called `vtime`, the time point at which the sample is deemed valid. A CU can then filter away old samples if necessary.

If an application needs more complex types then objects called "entity frames" can be used. Entity frames are basically aggregations of different values. Defining them is easy. An example object is given below:

```
WorldObject WORLD_OBJECT
long id −1
Vec pos Idl::Vec()
Vec vel Idl::Vec()
double length 0
double width 0
bool on_roadsystem true
```

It is also possible for an entity frame to contain another, like in the following example:

```
Car CAR
long id −1
long on_road_object −1
EntityFrame OnRoadObject Idl::Dyknow::EntityFrame()
```

Linking objects in this way lowers the amount of attributes that needs to be duplicated. In the example shown above it is possible to get the Car object's position by getting the reference to the OnRoadObject, which in turn offers a reference to the WorldObject that contains the position value.

The universality of streams, sources and entity frames should allow for almost any kind of input to be used in a DyKnow application.

5.3.3 How easy is it to integrate DyKnow into an existing environment?

Integrating any software into an existing environment may be summarized in three (not so small) steps: Define inputs, define outputs and resolve synchronization issues.

Defining application specific inputs was discussed in the previous section. Let us instead move quickly on to outputs. Remember that there are no specific outputs in DyKnow, although you can easily subscribe to and read from all the different streams from any DyKnow and CORBA-aware application.

The export proxy described in Chapter 4 is an example of an object subscribing to a stream. Instead of talking to the import proxy the export proxy might as well have stored all samples in a text file, or sent them to another program or subsystem entirely.

As mentioned about the implementation of the export proxy in Chapter 4, subscribing to a stream only requires that you write a suitable callback function and pass it to a subscription proxy that monitors the stream you wish to get samples from.

Defining inputs and outputs is however not always enough. To integrate well with the rest of the system the DyKnow application needs to be properly synchronized with the other parts. For example, the software responsible for generating data sent to DyKnow may need to know when DyKnow is ready to receive that data. Unfortunately, there is no global state exposed by DyKnow, no "I am ready" signal is broadcast. DyKnow is by design an asynchronous framework. It is however simple enough for a DyKnow application programmer to add this kind of signal when all CUs and streams are properly configured.

5.3.4 How easy is it to implement the scenario-specific chronicles in DyKnow?

As mentioned in Section 3.3.5 a chronicle requires the definition of attributes and domains as well as the actual chronicle.

The example used below to illustrate what is required to define a chronicle is the overtake chronicle used in the scenario implementation.

The domains specified in the `overtake` chronicle is the set of possible `CarRelation` objects, the set of possible `Car` objects and the set of boolean values (`unknown`, `true` and `false`). The definitions look like the following:

```
domain CarRelations = ~{}
domain Car = ~{}
domain Boolean = {unknown, true, false}
```

The `CarRelations` and `Car` domains are both set to the universal set (or rather, the complement of the empty set) to allow any possible value to be recognized. The peculiar `unknown` value in the Boolean domain is used as an initial value, until an attribute can be said to be either false or true.

The chronicle attributes `car_relations.{left_of, right_of, in_front, behind}` all map straight onto the values of the same name in the `car_relations` stream. They all share the same basic structure, shown below:

```
attribute car_relations.right_of[?car1, ?car2]
{
  ?car1 in Car
  ?car2 in Car
  ?value in Boolean
}
```

The full chronicle is shown in Listing B.1.

As seen, the chronicles themselves are quite simple, the real issue is to prepare the streams that the chronicles monitor. In the case of the `overtake` chronicle, the stream of cars must be converted into a stream of all pairs of cars. These pairs in turn needs to be given the proper attributes. Only when we have this proper `car_relations` stream may instances of the chronicle be recognized.

Chronicles can however become more complex when the values of the attributes they monitor oscillate. If we were using the naïve implementation of the `overtake` chronicle:

```
chronicle overtake[?car1, ?car2]
{
  event(car_relations.behind[?car1, ?car2]:(false,true), t1)
  event(car_relations.left_of[?car1, ?car2]:(false,true), t2)
  event(car_relations.in_front[?car1, ?car2]:(false,true), t3)

  t1 < t2 < t3
}
```

What happens when a car is behind another, then left of, then behind again before completing an overtake? The answer is that the chronicle recognition system recognizes all possible sequences of events, resulting in "duplicate" chronicles. Therefore it is necessary to provide checks against these sort of events. The revised chronicle looks like this:

```
chronicle overtake[?car1, ?car2]
{
  event(car_relations.behind[?car1, ?car2]:(false,true), t1)
  noevent(car_relations.behind[?car1, ?car2]:(true,false),
                                                  (t1+1, t2-1))

  event(car_relations.left_of[?car1, ?car2]:(false,true), t2)
  noevent(car_relations.behind[?car1, ?car2]:(false,true),
                                                  (t2+1, t3-1))

  event(car_relations.in_front[?car1, ?car2]:(false,true), t3)
  noevent(car_relations.left_of[?car1, ?car2]:(false,true),
                                                 (t3+1, t3+50))
  noevent(car_relations.in_front[?car1, ?car2]:(true,false),
                                                 (t3+1, t3+50))

  t1 < t2 < t3
}
```

The added `noevent` declarations prevent most duplications. Note that this chronicle assumes that the different attributes cannot be true at the same time.

Having compiled the chronicle, all that is left to run the chronicle in a DyKnow application is to register it with the CRE. This can be done with this single line of code:

```
cre->register_chronicle("overtake");
```

The CRE will automatically look for the file "overtake.xrs" in the chronicles directory.

In the default settings, the DyKnow CRE outputs all the events it has recognized into a file called "pe_received" in the working directory. Viewing this file allows you to see which events the chronicle recognition system has been given. When a chronicle is either recognized or rejected they are placed in the files "chronicles_recognized" and "chronicles_rejected" respectively. Using these files you are able to perform basic debugging of your chronicles. It is also possible to read recognized chronicles from the `chronicles` stream inside DyKnow as well as a ACE real-time event channel.

The chronicle recognition system CRS distributed with DyKnow provides a compiler (`ccrs`), a tool to view the contents of compiled chronicles (`lcrs`) and GUI application (`xcrs`) that can be used to test chronicles. The GUI application is able to load chronicles and events, and display what happens internally in the recognition engine.

To summarize, writing a simple chronicle is easy, though you might need to put some more work into them to avoid duplication and to accommodate for some border cases. Much of the work necessary is done outside the chronicle, including tweaking the necessary streams.

5.3.5 How easy is to include some kind of operator control in the system?

The implementation of an operator to control a single or multiple DyKnow instances was discussed in Chapter 4.

To summarize, the implementation of an operator is made easier by the fact that DyKnow uses CORBA, allowing implementation in a number of different programming languages and platforms. Implementing an operator may be made even simpler by using a framework defined specifically for multi-agent communication, such as JADE.

One thing that was not implemented or discussed in Chapter 4 was the ability of a human operator to control the processing inside a CU. For example, one might want to manually tell DyKnow which WorldObjects are to be considered OnRoadObjects or which OnRoadObjects are actually just false positives. This can be done by adding a new stream as input to the CU. The stream will not contain sensor values or anything of the like. Instead it will be used as a command channel. The command samples can then be generated anywhere and added to the stream as necessary.

A typical interface for this kind of functionality is a video stream of the vision of the UAV, overlaid with the objects identified by DyKnow. The operator may then select the different objects and create commands to send to DyKnow. Actually implementing this kinds of interfaces might be tricky. Getting the values from DyKnow is as easy as subscribing to the relevant streams but depending on the implementation, getting coordinates that are compatible with those of the video stream might require a series of transformations.

5.3.6 How can temporary loss of input or communication channels be handled?

What if there simply is no new samples in a stream? If any of the CUs subscribing to the stream also listens to another stream, then the last available sample is used. If however, the CU subscribes only to the stream no longer producing samples, it will simply never be called. A computational unit never processes without receiving any new samples.

In the event that there is a CU that for some reason needs to be called often despite the fact that it has run out of samples to process, it should be possible to set a value approximation constraint on the connected stream. Using a value approximation constraint it is possible to control what happens when new samples are requested but no new samples exist.

Unfortunately, the value approximation constraint was missing from the version of DyKnow used in this report, instead it was possible to to connect a simple source constantly providing only small integers. This will cause the CU to execute every time it receives a new integer, even though it may just ignore it.

What happens if the link to a CORBA object is suddenly lost? For example, what happens if the export proxy in the scenario implementation loses its connection with the import proxy object on the other agent? In that case any method called on the import proxy object will fail, causing a CORBA exception to be thrown. Depending on the situation the exception thrown is likely to be either `COMM_FAILURE`, `TRANSIENT` or `OBJECT_NOT_EXIST`. You can then handle these exceptions in ways appropriate for the particular application. The current implementation of the export proxy simply aborts the export and remove any references to the receiving import proxy. Another solution would be to look up the import proxy of the receiver on every new push, instead of storing the reference. This should cause only a minor hit to the performance, depending on how often new samples are made available. It is also possible to call the `_non_existent()` method on a CORBA object to see if it is a dangling reference before you try to call any other methods on it.

5.3.7 How much work was needed to extend the single-UAV scenario to the multi-UAV scenario?

The streams, sources and computational units that make up the scenario didn't need to be changed at all when a new instance was introduced. In fact, each instance run the exact same binaries, with a program switch to determine which unit number was assigned to which. This setting affect mainly which file the logged samples are read from.

What was necessary to get these instances to share information was the introduction of the import and export proxies. These proxies are described in Section 4.1.1. With the successful implementation of the proxy objects what was left was to somehow merge the exported stream with the local streams. This was done by introducing the `wo_merger` computational unit described in Section 4.2.3. This CU merges the two streams into a single stream which is then used by the rest of the system exactly like if all samples were collected locally. To launch yet another instance, only minor accommodations would have to be made.

A lot of work was however done creating a flexible operator to control the different DyKnow instances. This was primarily to test how easy it was to integrate DyKnow with another program. This was discussed earlier in Section 5.3.5. If this integration was not a goal, it might have sufficed with a couple of small scripts to launch or terminate agents and send them commands.

5.4 Summary

This chapter has described both a general evaluation of DyKnow using the ISO 9126-1 evaluation model and a number of DyKnow-specific evaluation questions.

The DyKnow-specific questions are not given concrete answers. Instead, enough information as possible has been presented for the reader to make his or her own decision.

5.4.1 ISO 9126-1 evaluation summary

An evaluation of DyKnow was done using the quality model for external quality from the ISO 9126 standard. The results may be summarized as follows:

- The functionality section scored high in all attributes with the exception of security which scored low because of the lack of access control or authentication.
- The reliability section got a medium overall score.
- The usability section scored medium to low, mainly because of the lack of detailed documentation regarding the low level usage of DyKnow.
- The scores regarding portability varied greatly. Adaptability and replacability scored low while installability was judged as medium and co-existence as high.

Left out from the evaluation was the efficiency characteristic which are described thoroughly in Chapter 6. The maintainability characteristic was also disregarded because of the chosen viewpoint of the evaluation. The different sub-characteristics concerning compliance was dropped from the evaluation since there were no requirements specified regarding any sort of compliance.

The results show that one of the areas that could be improved is the usability of DyKnow. In particular, more thorough documentation regarding its usage would help. One other area that scored low was security. Security is arguably outside the scope of DyKnow and might be better implemented separately as necessary in each individual application.

It is important to note that DyKnow as it is now is a research prototype and therefore characteristics such as usability and security have not been explicit goals during development.

Overall, DyKnow performs well in the tasks that it is supposed to do.

Chapter 6

Performance evaluation

The purpose of this chapter is to evaluate the performance of DyKnow using a set of different test cases. The goal is to see how DyKnow performs under load and see what affects the run-time performance.

The chapter starts by defining the different test cases in Section 6.1 and goes on to discuss the test results in Section 6.2.

6.1 Metrics and test cases

The metrics chosen to study in this performance evaluation and its test cases are the following:

1. The CPU usage of the application.
2. The memory usage of the application.
3. The time it takes for a sample to arrive at the last CU (computational unit) from the source. (This is also called the "sample delay time" throughout this chapter)
4. The time between samples arriving at the last CU.

The test cases were divided in three where the number of CUs, samples and chronicles were varied.

1. In the first test case, the number of samples was varied, the number of CUs was set to 10 and the number of chronicles was set to 0.
2. In the second test case, the number of CUs was varied, the number of samples was set to 1000 and the number of chronicles was set to 0.
3. In the third test case, the number of chronicles was varied, the number of CUs was set to 10 and the number of samples was set to 1000.

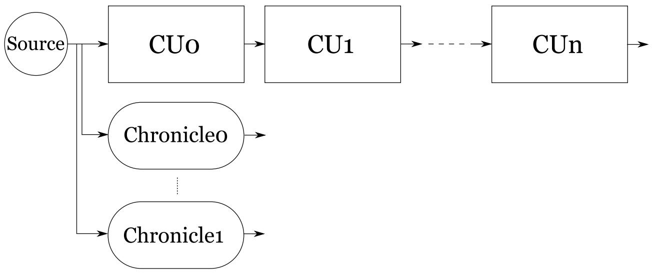

Figure 6.1. An overview of the architecture of the test application. All computational units are connected by streams in a series, while the chronicles monitor the first stream. In each test case the chain of CUs and the number of registered chronicles are varied.

6.1.1 Test setup

Figure 6.1 shows the architecture of the test application. The application was set to poll its source every one hundred milliseconds. All CUs were configured to simply pass the sample forward in the chain, no other processing was done.

The computer the test were run on was a Dell Precision M6300 equipped with a 2.20 GHz Pentium Core 2 Duo-processor and 2 gigabytes of memory running Red Hat Enterprise Linux Client 5.0. To ensure uniform test results despite the processors dual-core nature, the DOR-process was forced to use one of the CPU cores while the other processes (including the CRE) were forced to use the other using the program `taskset`.

No other demanding programs were run simultaneously as the tests. The CPU and memory were measured manually using the simple system monitor provided by the operating system. The values in the results should therefore individually be viewed as inexact.

In the test cases where the number of chronicles were varied the time between each recognized chronicle were set to 400 ms. This low value (in a typical application, chronicle instances are probably recognized much left often) was chosen to show how the system behaves when it is under heavy load.

6.2 Results

In this section the results of the test cases are described, beginning with the CPU and memory usage and continuing with the sample delay times and time difference at the last CU.

All the figures shown were generated from data available in the appendix marked “Tables.” In the different time-measuring figures, the error bars note the highest and lowest recorded time respectively.

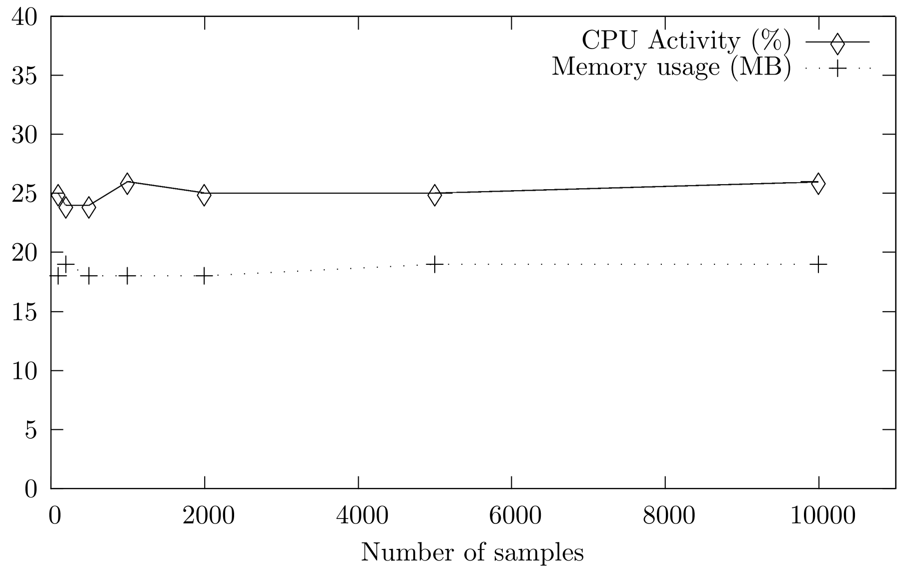

Figure 6.2. The memory and CPU requirements of DyKnow when the number of samples is varied. The number of CUs is fixed at 10. The data used to generate this figure is given in Table C.1.

6.2.1 Memory consumption and CPU usage

The effect the different test cases had on the memory consumption and CPU usage of DyKnow are described below.

Varying the number of samples

As seen in Figure 6.2 both the CPU and memory stayed largely constant despite the increase in the amount of samples processed. This is because the test application was configured never to permanently store any samples.

Varying the number of CUs

In Figure 6.3 we see that the CPU usage rises sharply with the number of CUs, or rather, the streams connecting the CUs, as each stream is contained in a separate thread. The memory requirements rose linearly with the number of computational units.

It is important to note that when the CPU usage approached 100% the threads were probably being starved and started to drop small amounts of samples randomly. The number of samples lost were at one time measured to 5 out of 1000 when using 30 CUs and the number only increased slightly when the number of CUs were increased further.

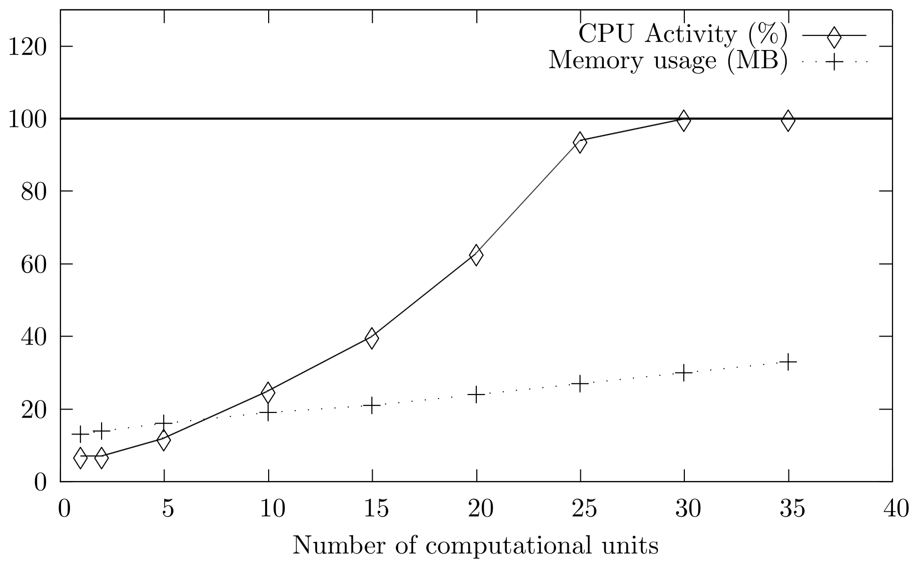

Figure 6.3. The memory and CPU requirements of DyKnow when the number of computational units is varied. The number of samples is fixed at 1000. The data used to generate this figure is given in Table C.2.

Varying the number of chronicles

We see in Figure 6.4 that while the demand of the CRE rose only slightly with the number of chronicles, the CPU usage for the DOR went up somewhat faster, despite the fact that no new streams were explicitly added. This is probably due to the amount of new samples created (but not stored) by the chronicles.

The demand for memory again rose linearly with the number of chronicles.

6.2.2 Sample delay time

During the same tests as above, data was also gathered to measure the time it took for a sample to propagate from its creation at the source to the last CU in the series as well as the time between samples at the last CU.

Note that the logging method used to measure the sample delay time is responsible for a delay of roughly 100 ms.

Varying the number of samples

Figure 6.5 shows how the total sample delay time differs when the number of samples is varied. We see that the mean time is almost constant at around 120 ms (more precise numbers are available in Table C.4). The maximum time however vary more greatly independently of the number of samples. The high times were

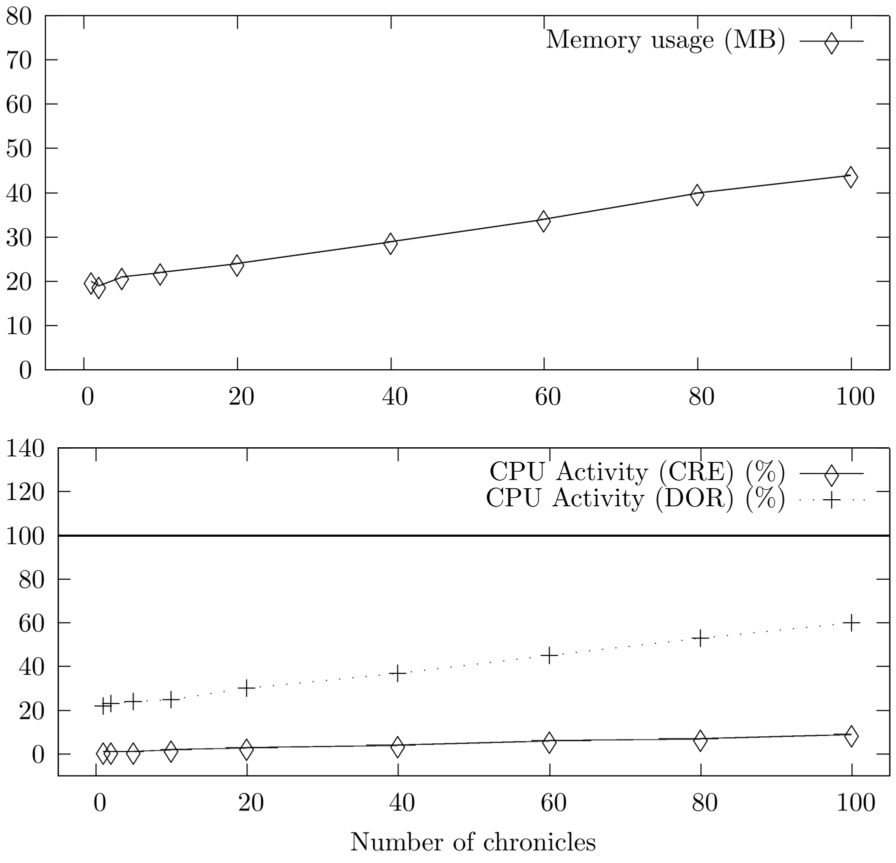

Figure 6.4. The memory and CPU requirements of DyKnow when the number of chronicles are varied. The number of CUs and samples are fixed at 10 and 1000 respectively. In the top-most plot we see the overall memory consumption and in the lower plot we see the effect on the two processes. The data that was used to generate this figure is given in Table C.3.

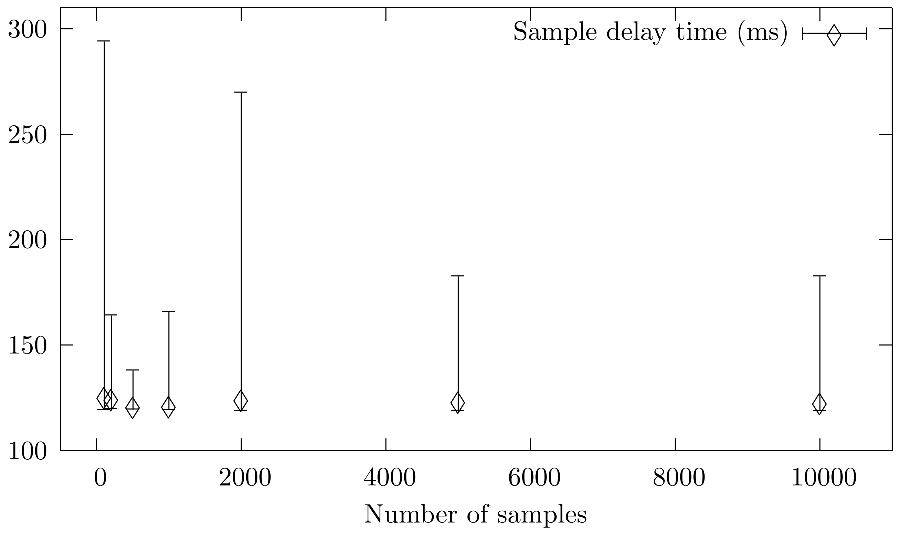

Figure 6.5. The number of milliseconds it takes for a sample to traverse 10 CUs when the number of samples is varied. The data used to generate this figure is given in Table C.4. Note: The logging method for obtaining these numbers is responsible for a delay of roughly 100 ms.

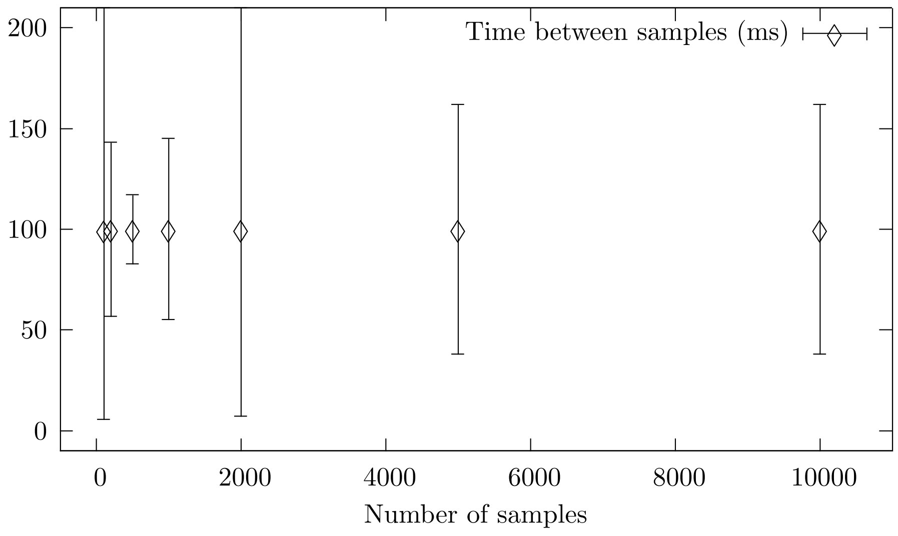

Figure 6.6. The number of milliseconds between each sample when they reach the last CU in the sequence when the number of samples is varied. The data used to generate this figure is given in Table C.5.

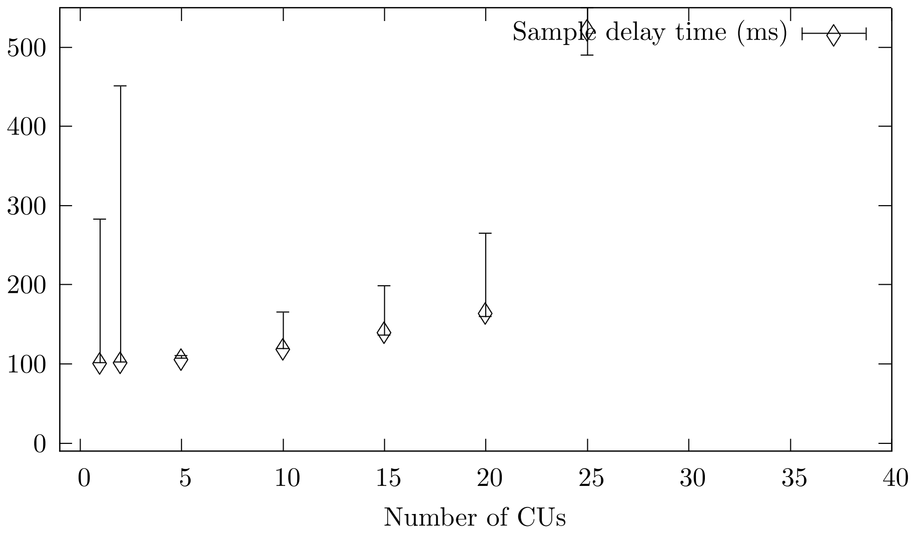

Figure 6.7. The number of milliseconds it takes for a sample to traverse all CUs when the number of samples is fixed at 1000. The data used to generate this figure is given in Table C.6. Note: The logging method for obtaining these numbers is responsible for a delay of roughly 100 ms.

probably caused by temporary load spikes in the computer.

If we instead measure the time between when samples arrive at the last CU we get the results shown in Figure 6.6 (Table C.5). We see that the mean time between the samples remain at the 100 ms they were sampled at the source. The look on the error bars indicate that whenever one sample is delayed, the following sample catches up proportionally.

Varying the number of CUs

The results gotten when we vary the number of computational units are displayed in Figure 6.7 and Figure 6.8 respectively. We see that when the number of CUs, or rather, the number of streams connecting the CUs rise, the total sample delay time increases linearly until the CPU nears 100% utilization and the delay time increases drastically. Remember that every stream is implemented in its own thread which needs processor time. When the CPU utilization nears 100% the threads are starved which may cause samples to be lost as well as considerably larger delay times (compare Figures 6.7 and 6.3). The mean time between the samples arriving at the last CU however remains at about 100 ms.

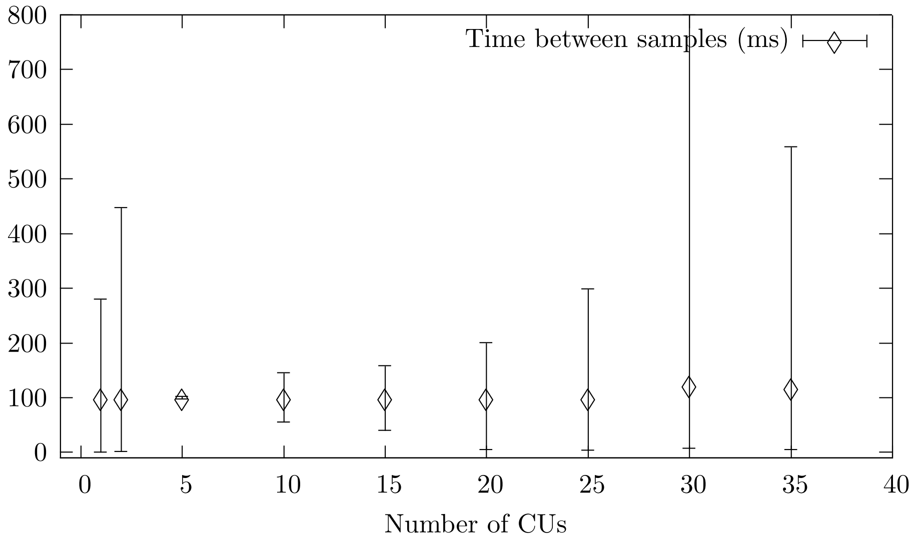

Figure 6.8. The number of milliseconds between each sample when they reach the last CU in the sequence when the number of CUs is varied. The number of samples is fixed at 1000. The data used to generate this figure is given in Table C.7.

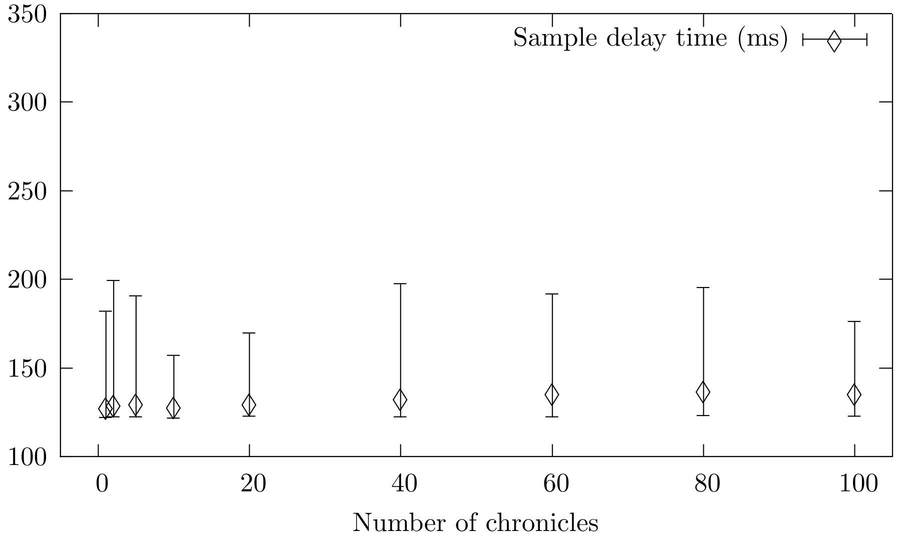

Figure 6.9. The number of milliseconds it takes for a sample to traverse 10 CUs when the number of samples is fixed at 1000 and the number of chronicles is varied. The data used to generate this figure is given in Table C.8. Note: The logging method for obtaining these numbers is responsible for a delay of roughly 100 ms.

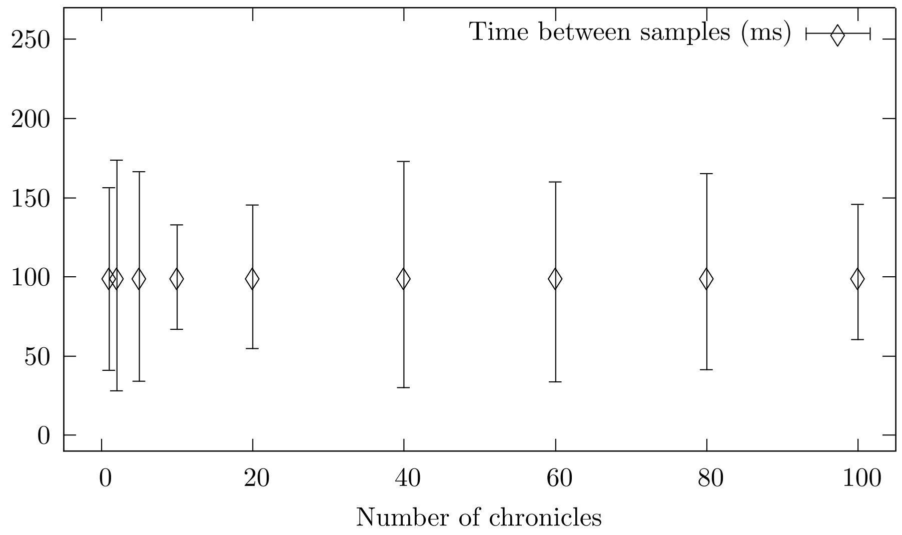

Figure 6.10. The number of milliseconds between each sample when they reach the last CU in the sequence when the number of chronicles is varied. The number of CUs and samples are fixed at 10 and 1000 respectively. The data used to generate this figure is given in Table C.9.

Varying the number of chronicles

In Figures 6.9 and 6.10 we see, confirming what one would expect, that the amount chronicles has no effect on either the delay time or the time between samples reaching the last CU.

6.3 Conclusion

In conclusion, the number of CUs and streams seem to be the single most significant factor in the demand placed on the CPU. It is important to not let the CPU utilization reach 100% as that will lead to thread starvation and loss of samples as well as sharp spike in delay times.

Neither the number of samples or chronicles affected the delay times and no test case showed an increase in the mean time between samples. The number of samples processed also had no measurable impact on either CPU or memory usage. Increasing the number of chronicles in the system does increase the load on the DOR and the amount of memory required.

The fact that DyKnow suffers from loss of data and severe slowdown when the CPU is heavily utilized may present a problem when DyKnow is not the only application running on a system.

It is difficult to judge these results without a referring to a set of requirements.

All that can be said is that the performance of DyKnow presents no problem for applications of the same size as the one discussed in this report.

Chapter 7

Conclusion

This report has presented an evaluation of the knowledge processing middleware framework DyKnow.

In order to perform the evaluation, a test application was developed using DyKnow to implement the car monitoring scenario described in Chapter 2. A detailed description of this implementation is given in Chapter 4 and describes all the different fluent streams, computational units and chronicles used. Alongside of DyKnow, a multi-agent environment was implemented using JADE, the Java Agent DEvelopment framework. The JADE-agents allowed the DyKnow instances to communicate more dynamically. Yet another agent was written, an operator, that allowed a user to give the DyKnow agents commands indirectly.

An evaluation of DyKnow using the quality model found in the international standard ISO 9126-1 is presented in Chapter 5. The quality model is divided in a number of software quality characteristics such as "functionality," "reliability" and "usability." The results of this evaluation highlights the fact that while the usability of DyKnow was judged as low, (mainly because of a lack of comprehensive documentation and programming guides) the suitability, accuracy and interoperability was judged as high.

Several of the poor results in the ISO 9126-1 evaluation can be attributed to the fact that DyKnow is a research prototype and that those characteristics therefore have not been prioritized during development.

Apart from the evaluation using the ISO 9126-1 quality model, Chapter 5 also discussed a number of questions pertaining to the usage of DyKnow. What was asked was, among other things, how easy it is to integrate DyKnow into an existing environment, how one would add a fusion algorithm and how temporary loss of communication channels can be handled. All of these questions were discussed in depth to give the reader a chance to make his or her own judgment. The author believes and hopes it is a positive one.

The performance evaluation in Chapter 6 showed how the performance and system requirements of DyKnow change when different variables are varied. The three variables tested are the number of computational units, fluent streams and chronicles used. The results show that memory requirements of DyKnow was dominated by the amount of CUs and chronicles registered and that the CPU usage was affected the most by the number of CUs and streams and to a lesser extent the number of chronicles. The results also showed that a reasonably sized DyKnow application, such as the scenario implementation used in this report, should run without problems on systems at least half as fast as the one used in the tests.

Overall this report and the results herein show that DyKnow is well suited for the task of tracking and reason about the relationships of objects in a traffic monitoring application using multiple UAVs.

Appendix A

Acronyms

AIICS
: Artificial Intelligence & Integrated Computer Systems Division at IDA.

IDA
: The Department of Computer and Information Science at Linköping University.

UAV
: Unmanned Aerial Vehicle.

CORBA
: Common Object Request Broker Architecture, provides an architectural framework for developing distributed systems. Defined by the OMG.

OMG
: Object Management Group, an international industry consortium.

ORB
: Object Request Broker, the central component of any CORBA implementation.

ACE
: ADAPTIVE Communication Environment, a framework for concurrent communication software.

JADE
: Java Agent DEvelopment Framework.

TAO
: The ACE ORB, a real-time CORBA implementation built upon the ACE framework.

FIPA
: The Foundation for Intelligent Physical Agents.

FIPA-ACL	The FIPA Agent Communication Language.
DOR	Dynamic Object Repository, a soft real-time database used by DyKnow to store streams.
CRE	Chronicle Recognition Engine, a DyKnow-wrapper around CRS.
CRS	Chronicle Recognition System, developed by France Telecom.

Appendix B

Listings

```
domain CarRelations = ~{}
domain Car = ~{}
domain Boolean = {unknown, true, false}

attribute car_relations.left_of[?car1, ?car2]
{
  ?car1 in Car
  ?car2 in Car
  ?value in Boolean
}

attribute car_relations.right_of[?car1, ?car2]
{
  ?car1 in Car
  ?car2 in Car
  ?value in Boolean
}

attribute car_relations.in_front[?car1, ?car2]
{
  ?car1 in Car
  ?car2 in Car
  ?value in Boolean
}

attribute car_relations.behind[?car1, ?car2]
{
  ?car1 in Car
  ?car2 in Car
  ?value in Boolean
}

chronicle overtake[?car1, ?car2]
{
  event(car_relations.behind[?car1, ?car2]:(false,true), t1)
  event(car_relations.left_of[?car1, ?car2]:(false,true), t3)
  event(car_relations.in_front[?car1, ?car2]:(false,true), t5)

  t1 < t3 < t5
}
```

Listing B.1. The definition of a chronicle capable of detecting if one car overtakes another car.

```
domain CarRelations = ~{}
domain Car = ~{}
domain Boolean = {unknown, true, false}

attribute car_relations.left_of[?car1, ?car2]
{
  ?car1 in Car
  ?car2 in Car
  ?value in Boolean
}

attribute car_relations.right_of[?car1, ?car2]
{
  ?car1 in Car
  ?car2 in Car
  ?value in Boolean
}

attribute car_relations.in_front[?car1, ?car2]
{
  ?car1 in Car
  ?car2 in Car
  ?value in Boolean
}

attribute car_relations.behind[?car1, ?car2]
{
  ?car1 in Car
  ?car2 in Car
  ?value in Boolean
}

chronicle overtake[?car1, ?car2]
{
  event(car_relations.behind[?car1, ?car2]:(false,true), t1)
  noevent(car_relations.behind[?car1, ?car2]:(true,false),
                                                (t1+1, t3-1))
  event(car_relations.left_of[?car1, ?car2]:(false,true), t3)
  noevent(car_relations.behind[?car1, ?car2]:(false,true),
                                                (t3+1, t5-1))
  event(car_relations.in_front[?car1, ?car2]:(false,true), t5)
  noevent(car_relations.left_of[?car1, ?car2]:(false,true),
                                               (t5+1, t5+50))
  noevent(car_relations.in_front[?car1, ?car2]:(true,false),
                                               (t5+1, t5+50))
  t1 < t3 < t5
}
```

Listing B.2. The definition of a chronicle capable of detecting if one car overtakes another car. This version is capable of filtering out most duplicate recognitions.

Appendix C

Tables

Samples #	Memory consumed (MB)	CPU Activity (%)
100	18	25
200	19	24
500	18	24
1000	18	26
2000	18	25
5000	19	25
10000	19	26

Table C.1. The memory and CPU requirements of DyKnow when the number of samples is varied. The number of CUs is fixed at 10. Figure 6.2 shows a diagram of the data.

Computational Units #	Memory consumed (MB)	CPU Activity (%)
1	13	7
2	14	7
5	16	12
10	19	25
15	21	40
20	24	63
25	27	94
30	30	100
35	33	100

Table C.2. The memory and CPU requirements of DyKnow when the number of computational units is varied. The number of samples is fixed at 1000. Figure 6.3 shows a diagram of the data.

		CPU Activity (%)	
# of chronicles	Memory consumed (MB)	CRE	DOR
1	20	1	22
2	19	1	23
5	21	1	24
10	22	2	25
20	24	3	30
40	29	4	37
60	34	6	45
80	40	7	53
100	44	9	60

Table C.3. The memory and CPU requirements of DyKnow when the number of chronicles is varied. The number of CUs and samples are fixed at 10 and 1000 respectively. Figure 6.4 shows a diagram of the data.

# of samples	Lowest (ms)	Highest (ms)	Mean (ms)
100	119	294	125
200	120	164	124
500	119	138	121
1000	119	165	121
2000	119	269	124
5000	118	182	123
10000	119	182	123

Table C.4. The number of milliseconds it takes for a sample to traverse 10 CUs when the number of samples is varied. Figure 6.5 shows a diagram of the data. Note: the logging method for obtaining these numbers is responsible for a delay of roughly 100 ms.

# of samples	Lowest (ms)	Highest (ms)	Mean (ms)
100	5.5520	256.9390	99.8171
200	56.8610	143.2760	99.9862
500	82.9100	117.0730	99.9968
1000	55.1580	145.1320	99.9978
2000	7.0820	249.1790	99.9994
5000	37.9040	162.0810	100.0059
10000	38.0980	162.0320	99.9998

Table C.5. The number of milliseconds between each sample when they reach the last CU in the sequence when the number of samples is varied. Figure 6.6 shows a diagram of the data.

# of CUs	Lowest (ms)	Highest (ms)	Mean (ms)
1	101	282	103
2	102	451	104
5	106	110	107
10	119	165	121
15	136	199	142
20	159	265	166
25	489	698	524
30	4142	30949	18690
35	50246	92723	76016

Table C.6. The number of milliseconds it takes for a sample to traverse all CUs when the number of samples is fixed at 1000. Figure 6.7 shows a diagram of the data. Note: the logging method for obtaining these numbers is responsible for a delay of roughly 100 ms.

# of CUs	Lowest (ms)	Highest (ms)	Mean (ms)
1	1.0080	280.5580	99.9983
2	1.4420	447.9100	99.9984
5	97.7510	102.8250	99.9980
10	55.1580	145.1320	99.9978
15	39.9520	159.1300	99.9971
20	5.3900	200.9800	99.9676
25	3.6300	299.1090	100.0311
30	8.0400	1014.7740	123.8165
35	4.7540	558.5860	118.5290

Table C.7. The number of milliseconds between each sample when they reach the last CU in the sequence when the number of CUs is varied. The number of samples is fixed at 1000. Figure 6.8 shows a diagram of the data.

# of chronicles	Lowest (ms)	Highest (ms)	Mean (ms)
1	122	181	128
2	122	199	129
5	122	190	130
10	121	157	128
20	122	169	130
40	122	197	133
60	122	191	136
80	123	195	137
100	122	176	136

Table C.8. The number of milliseconds it takes for a sample to traverse 10 CUs when the number of samples is fixed at 1000 and the number of chronicles is varied. Figure 6.9 shows a diagram of the data. Note: the logging method for obtaining these numbers is responsible for a delay of roughly 100ms.

# of chronicles	Lowest (ms)	Highest (ms)	Mean (ms)
1	40.8690	156.2380	99.9951
2	27.8670	173.8600	99.9983
5	33.9400	166.4560	99.9954
10	67.0450	132.8700	99.9926
20	54.7370	145.1800	99.9947
40	30.1490	173.0620	100.0247
60	33.7490	160.0640	100.0478
80	41.4360	165.1670	100.0586
100	60.2040	145.7030	99.9645

Table C.9. The number of milliseconds between each sample when they reach the last CU in the sequence when the number of chronicles is varied. The number of CUs and samples are fixed at 10 and 1000 respectively. Figure 6.10 shows a diagram of the data.

Bibliography

[1] JADE - Java Agent DEvelopmend Framework. "`http://jade.tilab.com`", 2007.

[2] Welcome to FIPA! "`http://www.fipa.org`", 2008.

[3] A. S. Andreou and M. Tziakouris. A quality framework for developing and evaluating original software components. *Information and Software Technology*, 49(2):122–141, 2007.

[4] E. P. Blasch and S. Plano. JDL Level 5 Fusion Model "User Refinement" Issues and Applications in Group Tracking. volume 4729, pages 270–279, 2002.

[5] Bloch and Hunter, et al. Fusion: General Concepts and Characteristics. *International Journal of Intelligent Systems*, 16:1107–1134, 2001.

[6] Tim Chamillard. Tailoring the 9126 Quality Model. "`http://www.cs.uccs.edu/~chamillard/cs536/Papers/9126Handout.pdf`", 2005.

[7] Fredrik Heintz, Tommy Persson, David Landén and Patrick Doherty. Report on extending DyKnow for Multi-Node Distributed Networks - A Prototype Implementation. 2008.

[8] Fredrik Heintz. *DyKnow: A Stream-Based Knowledge Processing Middleware Framework. Linköping Studies in Science and Technology Dissertation 1240. Linköpings universitet.* 2009.

[9] Fredrik Heintz and Patrick Doherty. DyKnow: A Framework for Processing Dynamic Knowledge and Object Structures in Autonomous Systems. In *Proceedings of the International Workshop on Monitoring, Security, and Rescue Techniques in Multiagent Systems (MSRAS)*, 2004.

[10] Fredrik Heintz and Patrick Doherty. Distributing and Merging Information using DyKnow. 2008.

[11] Michi Henning and Steve Vinoski. *Advanced CORBA programming with C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[12] Douglas C. Schmidt. Overview of ACE. "`http://www.cs.wustl.edu/~schmidt/ACE-overview.html`", 2006.

[13] Alan N. Steinberg, Christopher L. Bowman, and Franklin E. White. Revisions to the JDL data fusion model. *Proceedings of SPIE - The International Society for Optical Engineering*, 3719:430–441, 1999.

[14] Michael Wooldridge. *Introduction to MultiAgent Systems*. John Wiley & Sons, June 2002.