Participatory Middleware Design

Experimental Development of Semantic, Service Oriented, Context Aware Middleware for Ubiquitous Computing

Kristian Ellebæk Kjær

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Participatory Middleware Design

Experimental Development of Semantic, Service Oriented, Context Aware Middleware for Ubiquitous Computing

A Dissertation
Presented to the Faculty of Science
of Aarhus University
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Kristian Ellebæk Kjær
September 2009

Abstract

In modern agriculture embedded and mobile computers are ubiquitous. Most modern machinery and control systems have an embedded computer as well as some form of remote administration interface.

Furthermore, a modern farm both produces and consumes a large amount of information, gathered from heterogeneous sources. This includes information pertaining to the daily running of the farm, as well as external information which may or may not be of importance to the individual farm, such as weather forecasts, market prices, and advisory information.

To manage this information efficiently, something better than current practice is necessary. One method for managing information is context awareness. If the current context of the farm and its workers is gathered, it can be used to determine what information is relevant in any given situation. To accomplish this, there are two major challenges: Collecting the context in a way that allows for aggregation and reasoning, and providing developers with appropriate abstractions for creating applications based on context.

To accomplish this, we need a context model and a middleware supporting both the gathering of context and building applications suitable for agricultural settings. For this, we explore the use of participatory design techniques in the design of middleware, the use of first class connectors as abstractions for coordination in service oriented architectures, and the use of semantic web technologies for modelling and accessing context and architecture description.

We show that participatory design techniques lend themselves favourably to the design of middleware, how first class connectors are not only useful for middleware, but also for prototyping systems, and describe our use of semantic web technologies.

Acknowledgements

The work described in this dissertation has been carried out with the involvement of persons not formally tied to the project, but who have nevertheless offered their time. Among those, the author would like to thank: Nis Skau for allowing us access to his farm and workers, Jens Bliggaard from DAAS for cooperation on the second prototype and Ulrik Østergaard from Vikingegaarden for taking time to talk with us.

Per Nielsen worked as a student programmer on some parts of the first prototype allowing the author to concentrate on other, and in the authors opinion, more interesting work.

Furthermore, Klaus Marius Hansen acted as thesis advisor, and his input has been highly valued, both during the project and especially his comments on the dissertation. Rasmus Ellebæk Kjær also took the time to read drafts of the dissertation, and provided valuable input.

The research presented has been funded in part by the ISIS Katrinebjerg project "KILO" (ISIS; `www.isis.alexandra.dk`), Danish Agricultural Advisory Services (DAAS; `www.landscentret.dk`), Aarhus University (AU; `www.au.dk`), and the EU project "Hydra" (IST-034891; `www.hydramiddleware.eu`).

Kristian Ellebæk Kjær
Århus, September 2009

Contents

Abstract i

Acknowledgements iii

I Experimental Middleware Development 1

1 Introduction 5
1.1 Project Overview . . . 8
1.2 Research Methods . . . 9
1.3 Research Subjects . . . 10
1.4 Research Questions . . . 11
1.5 The rest of the dissertation . . . 13

2 Background 15
2.1 Middleware . . . 15
2.1.1 Web Services . . . 18
2.1.2 Architectural Views . . . 19
2.2 Ubiquitous Computing . . . 19
2.3 Semantic Web Technology . . . 22

3 Participatory Design 27
3.1 Research and Development Methods . . . 27
3.2 Participatory Design in Kilo . . . 31
3.3 Using Artifacts . . . 33
3.4 Results . . . 37
3.5 Summary . . . 38

4 Prototypes 39
4.1 Kilo.One . . . 41
4.1.1 Application Prototype . . . 42
4.1.2 Middleware Prototype . . . 44
4.2 Kilo.Two . . . 47
4.2.1 Application Prototype . . . 48

4.2.2 Middleware Prototype 50
4.3 Summary 54

II Middleware for Context Awareness 57

5 Context Awareness 61
5.1 Context 63
5.1.1 Using Context 68
5.2 Context models 70
5.3 The Role of Middleware 72
5.4 Context Aware Middleware 74
5.4.1 Categorising Middleware 75
5.4.2 Middleware Systems 78
5.4.3 Gaia 82
5.5 Context Awareness in Agriculture 88
5.6 Ontology Based Context Models 91
5.6.1 Context Model 91
5.7 Summary 93

6 First Class Connectors 95
6.1 First Class Connectors 96
6.1.1 Architectural Description 100
6.2 First Class Connectors in Programming Languages 101
6.3 First Class Connectors in Middleware 104
6.4 Prototyping Service Oriented Systems 106
6.4.1 Example Systems 108
6.4.2 Usage 109
6.4.3 Performance 110
6.5 Summary 111

7 Semantic Web Technology 113
7.1 Semantic Publish Subscribe 113
7.1.1 Conceptual Design 115
7.1.2 Implementation and Design 116
7.1.3 Expressiveness 118
7.1.4 Performance 119
7.1.5 Example of Use 121
7.1.6 Future Work 122
7.2 Semantic Architecture Description 123
7.3 Semantic Service Discovery 124
7.4 Summary 127

8 Summary and Conclusion 129
8.1 Results in research areas 130
8.2 Research Questions 130
8.3 Hypothesises 132
8.4 Future Work 133
8.5 Conclusion 133

A Requirements 135

B Code 139
B.1 Configuration Schema 139
B.2 Context Service WSDL interface 141
B.3 Echo Service OWL-S Profile 142

C GUI Mockups 153

Tables

3.1 Participatory Design Activities. 33
3.2 Use of artifacts in the development process. 35

5.1 Categorisation of Context-Aware Middleware Systems. 78
5.2 Excerpt from Crop Protection Online [1]. Season plan for crop protection in Spring Barley. 90

6.1 Marshalling between Java and XML types are according to this table. It is intended to use the same marshalling as JAX-RPC and Axis2, although they support additional types [82]. 108
6.2 Average, minimal, and maximal invocation time of a call to an operation for Axis, Axis2, and **WSConnector** between a client and a remote web service. 111

Figures

1.1 Conceptual view of the virtual desktop from the original project description. . . . 9
1.2 Activities and research areas in relation to Ubiquitous Computing. . 9
1.3 The five research subjects provide input to the research areas defined for the project. . . . 11

2.1 Middleware layers. Adapted from Schmidt [106]. . . . 17
2.2 Layered use of description technologies in Web Services. . . . 19
2.3 Kilo.Two middleware in relation to the middleware layers defined by Schmidt [106]. Details in section 4.2.2. . . . 20
2.4 Components and connectors in a simple application using a **Context** service and remote **Backend** web service. . . . 21
2.5 Tabs and pads in a ubiquitous computing environment. Adapted from Weiser [118]. . . . 21
2.6 The FAO Geopolitical ontology [48], excluding data properties, the OWL Thing concept, and individuals. Arrows without a label denotes the rdf:subClassOf property, which has been left out for readability. . . . 23
2.7 SUMO upper ontology, as defined in Niles and Pease [91]. . . . 24
2.8 An upper ontology for services as defined in OWL-S. Adapted from Martin et al. [22]. . . . 26

3.1 Future practice can be triangulated by fixing either means, environment, or job-at-hand, and varying the two others. Adapted from Büscher et al. [24]. . . . 31
3.2 Picture from field studies at a farm in southern Denmark - agricultural worker examining a sow for pregnancy. . . . 32
3.3 Artifacts in participatory design. Adapted from Mogensen and Trigg [89]. . . . 34
3.4 Participatory design of middleware. For middleware, the artifact becomes the middleware itself. . . . 34
3.5 Mockup of GUI for PDA application supporting daily tasks. Adaptation of original mockup, included in appendix C. . . . 36

4.1 The most important sub-domains in Danish agriculture. For other countries or regions, the important sub-domains will be different. . 40
4.2 Chart for recording information about a particular sow. 42
4.3 Ontology of the context model of **Kilo.One**. The central concept is *Location*, and the actual context model is implemented in an object oriented manner. 43
4.4 The **Kilo.One** situation concept. 43
4.5 Screenshot of the **Kilo.One** prototype, showing the "Small pigs stable" location and the entry for the "medication" task. 44
4.6 Overview of services and applications in the Kilo.One prototype. . . 45
4.7 Deployment of applications and services in the Kilo.One protoype. 45
4.8 Determining current task in plant production based on available information. 49
4.9 Simplified ontology of the task concept for tractors in fields. 50
4.10 Part of the context ontology of **Kilo.Two** prototype. 52

5.1 Sample hierarchy of traveling situations. 62
5.2 Context categories, as defined by Zimmermann et al. [121]. 66
5.3 Levels of interactivity are orthogonal to the type of context aware system. With decreasing interactivity and increase in proactive choices, the user gradually looses control. 69
5.4 Ontology of Entities in a Context Aware System. 73
5.5 Middleware provides abstractions between applications and the network stack in the operating system. 74
5.6 Taxonomy of Context-Aware Middleware concepts. 76
5.7 The Gaia meta-operating System architecture. Adapted from Roman et al. [99]. 83
5.8 The MiddleWhere system. Adapted from Ranganathan et al. [97]. . 84
5.9 MobiPADS architecture. Adapted from Chan and Chuang [29]. . . 86
5.10 The SOCAM architecture. Adapted from Gu et al. [59]. 87
5.11 Simplified UML model of the **Context Service**, showing the main classes involved. 92

6.1 Ontology of Coordination Concepts. 95
6.2 A Tuple Space is a persistent store where tuples can be written and retrieved. 98
6.3 Structure of an architectural language, consisting of a specification part with units of association for system composition and an implementation part. Adapted from Shaw [107]. 99
6.4 ArchJava programs compiles into Java which is then compiled to Java bytecode using the standard compiler. 101
6.5 The result of instantiating and EchoSystem component. 102
6.6 Relation between coordination and connector types in SAJ. 104

6.7 Connecting a client and a service through an ArchJava connector using Axis2. Axis2 instantiates the **ServerRole** object, when a message for the service is received. **ServerRole** then instantiates the actual component. 107
6.8 Comparison of Axis1, Axis2, and WSConnector. WSConnector is somewhat slower, mostly due to two layers of reflection in each call, whereas both versions of Axis use auto generated, specialised code for marshaling. 111

7.1 High-level system overview. 116
7.2 The publish-subscribe ontology with sample subclasses of event. *swrl:Imp* is the main concept for SWRL rules which represents the filters. . . 117
7.3 Example of a Topic Hierarchy. 118
7.4 Initialisation time in ms as a function of ontology size 120
7.5 Evaluation time in ms of a single subscription as a function of the ontology size. 121
7.6 Time in ms for Evaluating subscriptions as a function of the number of subscriptions for an ontology of size 411. 122
7.7 Part of the Kilo additions for the existing Hydra architecture ontology. Classes with solid borders are part of the Kilo ontology and arrows without label are rdfs:subClassOf properties which are left out for readability. 124
7.8 Defining a, very simple, architecture of an application using the Kilo architecture ontology. Individuals are underlined. 124
7.9 Sequence diagram showing the registration process of the Echo service. 126

C.1 GUI mockup used in initial design process (1). 153
C.2 GUI mockup used in initial design process (2). 154
C.3 GUI mockup used in initial design process (3). 154

Listings

4.1 The interface of the **Kilo.One** global location service in pseudo code. 46
4.2 The interface of the **Kilo.One** alarm service in pseudo code. 46
4.3 Standard **Kilo.Two** component with a `Component` attribute defining the external description. 54
5.1 Creating a new individual of type `Event` named "ContextEvent" and setting its data property to point to the "ContextData" individual which is already present in the ontology.. 92
5.2 The interface of the **Context** service as Java-like pseudo code. The WSDLinterface is provided in appendix B.2. 93
6.1 Simple Echo component in ArchJava. The component offers a port with three methods. 102
6.2 Simple Echo client in ArchJava. The client requires three different methods. 102
6.3 Defining relationship between components in ArchJava. 103
6.4 Static model of a simple echo system in SAJ. 105
6.5 `services.xml` defining the Echo operation implemented by the class **EchoService** with **ServerRole** as message receiver. 107
6.6 Using **WSConnector** to connect an **EchoClient** component and an **EchoService** component. The code defines a connect pattern, and will be called to instantiate a new **WSConnector** when **EchoClient** creates a new EchoPort. The Echo port of **EchoClient**. `connect` matches the constructor of the connector used, while the rest of the methods defines which methods the port require from the port is should be connected to. 109
6.7 The Port of the Google Client. The required methods match the WSDLdescription of the Google API. 110
7.1 XML serialisation of a simple Kilo application consisting of two components, **ContextClient** and **ContextStore**. 125
7.2 The interface of a simple **echo** service. 126
7.3 OWL-S service description of the Echo Service. 128
B.1 "XSD Schema for Kilo.Two Component Configuration." 139
B.2 WSDL Interface of the **Context** service 141
B.3 **Echo** service OWL-S description. 142
B.4 **Echo** service OWL-S profile. 144

B.5 **Echo** service OWL-S grounding. 146
B.6 **Echo** service OWL-S process. 148
B.7 **Echo** service OWL-S WSDL file. 150

Part I

Experimental Middleware Development

This part of the dissertation provides an introduction and overview of the experimental middleware development carried out in the project. The main focus is on describing the work, and not on presenting scientific results.

Chapter 1 Introduction introduces the research project and the main research questions and areas covered in the thesis and provides an overview of the rest of the thesis.

Chapter 2 Background provides description of some related work.

Chapter 3 Participatory Design starts with an overview of participatory design practice and discusses its relation to middleware development. It continues with describing the participatory design activities carried out and concludes with discussions of the outcome of these activities.

Chapter 4 Prototypes provides an overview of the prototypes created in the Kilo project, detailing the elements in the prototypes and their architecture.

Chapter 1

Introduction

Across our society, computers have become ubiquitous. Starting with the revolution of the personal computer in the late seventies and early eighties, desktop computer came to be everywhere and revolutionised the everyday work of office workers. As computers grew smaller and more powerful during the nineties, this led to small computers that we carry with us; laptops, mobile phones and PDAs. While these computers are visible and easily recognisable, we now have see an explosion in the number of invisible computers in the form of embedded devices. This has moved the use of computers outside the constrains of the office setting and into industry.

In industries that rely on large machinery, embedded devices have been used for control systems, automating a large number of processes. Even in more traditional industries, like agriculture, computers are ubiquitous. Computers and PDAs are used for management while embedded computers control equipment like HVAC[1] and feeding systems. Even modern tractors have a large number of embedded devices, providing control of other embedded devices controlling attached equipment like pesticide sprayers through on-board computers, location through GPS, and even automated driving systems.

The current state of computer systems in agriculture leads to lack of interoperability between the different systems used in agriculture. All these computer systems are typically created by different companies. Furthermore, management systems, even though they might be created by the same company, are often created with modules for different types of agriculture production, like pig or plant production which are detached from each other. This is partly due to difference in law and tradition in different areas. For instance, in Denmark, all cows are registered in a government mandated central database, while pigs are only registered in terms of number of births, deaths, and how many are slaughtered, so data has so far been kept locally at the farm since there is no reason for a central database. For plant production, a central database is also used, since it allows for central authorities to collect information about pesticides and fertiliser used, something that has to be reported to central authorities. However, unlike the registry for cows, this is not mandatory,

[1] Heating, Ventilation, and Air Conditioning

so not all farmers use it.

Since management systems use different storage, and may be produced by different companies, in general it is not possible to exchange data between systems. For embedded systems, the situation is even more severe, since there is a large number of producing companies. Furthermore, these manufacturers rarely have an incentive to open up there APIs, and instead opt for creating their own systems for remote management.

As an example of why this may be a problem, consider that it is actually possible to buy a weight for sows, which will recognise the sow based on an RFID chip in the sows ear-mark, and then register the changes in weight for that particular sow. However, currently the only way of transferring that information to an administration system is to print it out and enter it manually into the system. Another example could be that a modern pesticide sprayer measures flow in each nossle. Together with information about the contents of the tank, i.e. the concentration of pesticides, this is enough to determine how much pesticide has been used. If this is coupled with a GPS unit, which is present in most modern tractors anyway, it can even determine automatically how much pesticide has been used at each field, and provide evidence that the farmer has not broken laws by, for instance, spraying too close to a stream.

The ideal scenario to counter these problems would be like the following:

> ***Pig production***
> *A pig farmer has a farm with three houses. The first and second house have five pens each, while the third house has seven pens. Each pen contains from one up to four sub-pens. To help him he employs a supervisor and two agricultural workers.*
>
> *Each house is equipped with automatic ventilation systems and automatic feeder units. The feeder units are equipped with weighing machines, and each pig has a unique id tag, which can be read by the feeder units.*
>
> *The supervisor on the farm monitors the pigs by walking to each house. When he enters a house, his PDA immediately displays the temperature and settings of the ventilation system, and provides him with an interface to adjust the settings. Using his PDA, he access the information about the first pen. He can see that all the pigs in the pen are eating properly, and are growing as they should. When he enters the second house he notices from the information on his PDA that the pigs in pen no. 3 are slightly underweight. Walking over to the pen, the display on the PDA switches to an overview of the individual pigs in the pen, and he can see that the pigs develop fine, and apparently they just grow a bit slower than the other pigs.*
>
> *The temperature monitor in one of the houses registers that the temperature rises and sends out a notification. This causes the ventilation system to increase the flow, to keep the temperature constant at the specified set-*

ting. The outside temperature has not changed, and the pigs are drinking more water than usual. This causes an alarm to the farmer, notifying him that the pigs might be sick. To investigate the issue, the farmer accesses the current information on the pigs in the house to determine that the increase in water use is isolated to two of the pens in the house. He imideately notifies one of his agricultural workers to isolate the pigs in the two pens and then calls the veterinarian.

The veterinarian

Before driving to the farm, the veterinarian accesses the information about the pigs, and read a short report from the agricultural worker. He decides that they might suffer from a particular illness and makes sure to pack a suitable amount of medicine in his bag, derived from the weight of the pigs, and notes this in the journal.

Arriving at the farm, the veterinarian inspects the pigs and determines that his remote diagnose was correct and distributes the medicine. Afterwards he confirms the entry in the journal, and this is recorded in his own journal and in the medicine records of the farm.

When the pigs are well again, and the temperature drops, the ventilation system automatically reduces the airflow to normal.

Plant production

The pig farmer also has a large area of fields for distributing organic fertilizer from the pigs and for growing food crops.

During a work day, he receives a message on his cell phone from a central register, that meldew has been observed less than five miles from his farm and he decides to inspect his own fields. Taking his car, he drives out to the first of his fields. His PDA, using its built-in GPS unit, notices that he is near one of his fields, and displays a map of the area, with the individual fields overlayed.

In one of his fiels he discovers meldew. His PDA displays information about the current field, and he use it to find the appropriate pesticide. He then combine the information about the size of the field and other data, including knowledge about the soil, and the data on the pesticide to determine the amount he needs. His PDA informs him, that he does not have enough in stock, and asks if more should be ordered. After answering "yes", information about the task of distributing the pesticide is transferred to the common work plan, and one of the agricultural workers are notified that they must do the task when the pesticide arrives. The information about the field is also uploaded to the tractor, so that it can vary the dosage according to the soil in the different areas of the field. After the pesticide has been distributed, the exact dosage used along with the date and time and number of the field is recorded in the pesticide journal of the farm.

To enable such a scenario, management systems as well as the embedded devices in machinery would have to be integrated, allowing information to be exchanged between them. Preferably, the communication should be online, as opposed to off-line synchronisation, since this enables additional benefits, such as the ability to change plans on-the-fly based on the variables like the current position of equipment, current situation of workers, weather reports, and other external information.

Current state-of-the art in agricultural systems does not allow for this. The closest we have so far, is ad-hoc integration of control systems and the possibility of manually transferring information from one system to another. For instance, it is possible to buy systems which unify a number of product-specific remote management tools into a single web-page, with a single logon. However, this merely provides convenience for the user, and offers no actual integration of the devices.

While some properties of the agriculture domain are similar to other domains, agriculture differs from most other domains in the variety of the sub-domains. While a large body of the previously produced middleware is intended for somewhat homogeneous domains, agriculture needs a middleware that takes the heterogeneity of the sub-domains into account. For instance, it would be beneficial to take into account that some applications will be used almost entirely in a domain where a static communication infrastructure can be assumed, while other applications will mainly be used in a domain where communication is intermittent. Static environments might also provide services that are not available in elsewhere, and the middleware should be able to take advantage of these. On the other hand, these problems are not confined to the agricultural domain. Modern mobile devices will often be moved between environments providing some form of infrastructure, but will also be used in environments without a static infrastructure.

1.1 PROJECT OVERVIEW

The PhD project described in this dissertation is part of the Kilo project, a three year collaboration between ISIS Katrinebjerg, Danish Agricultural Advisory Services (DAAS), University of Aalborg, and Aarhus University as a development and research project for next generation information and management systems for agriculture. Central to the project is the idea of a unified "virtual desktop" where farmers can maintain an overview of a modern farm. A part of this idea is an underlying assumption that context awareness is useful for helping farmers maintain an overview when they can access a large amount of information through this desktop. As can been seen from the conceptual overview in figure 1.1, conceptually the desktop is supported by an *intelligent layer* and a *communication layer*. The intelligent layer provides filtering of the information retrieved from various sources through the communication layer. Compared to the research areas in figure 1.2, showing the research areas of the project in relation to activities, the intelligent layer corresponds to research in context awareness , and the communication layer corresponds to research in integration , while software architecture is a cross-cutting concern on all

three layers.

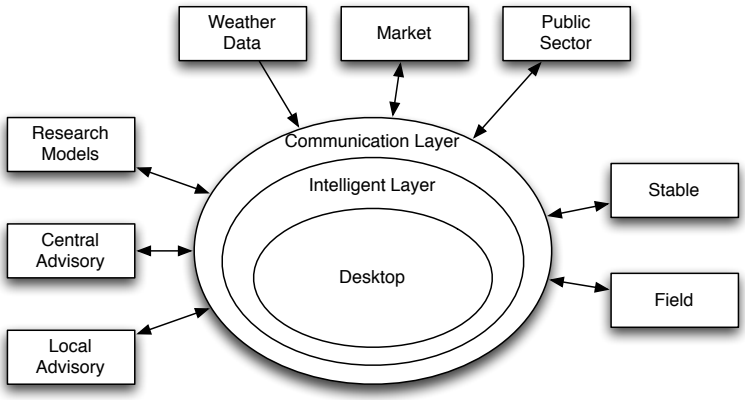

Figure 1.1: Conceptual view of the virtual desktop from the original project description.

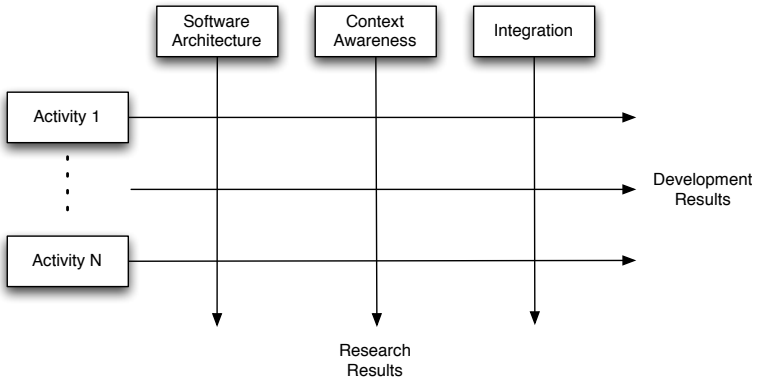

Figure 1.2: Activities and research areas in relation to Ubiquitous Computing.

The project started in August 2006 and ended in June 2009. During that time, two major activities where undertaken. The first dealt with development and research for supporting work in stables, while the second researched support for workers in the field. This dissertation describes the part of the work carried out at Aarhus University.

1.2 Research Methods

The research carried out in the PhD project focus on an *empirical* and *qualitative* approach. We aim to create design that is usable in a real world setting, and therefore we need to empirically study this setting and empirically evaluate the results. Furthermore, the methods used are qualitative rather than quantitative. As observed by March and Smith [80], the goal of such work is that problems should be properly

conceptualised and represented, appropriate techniques for the solutions constructed, and solutions should be implemented and evaluated.

The main research methods employed are inspired from Participatory Design [58]. These include:

- Field studies,
- interviews,
- workshops,
- focus groups, and
- prototyping.

Field studies and interviews were mainly used for initial requirements engineering and obtaining knowledge about the domain. Workshops and focus groups were then used to refine requirements, and for design activities. Finally, prototypes where used for feedback for the next iteration in the project. That is, we use field studies and interviews to conceptualise problems, workshops and focus groups for defining the appropriate techniques and prototyping for implementing and evaluating solutions. We have chosen these techniques because they support our qualitative approach. Field studies, interview, and workshops allows for in-depth understanding of the problems of the domain, while workshops, focus groups and prototyping supports the empirical implementation and evaluation of solutions.

As such, the research process employed is *experimental* and *iterative*. Experimental in that we develop software and evaluate it using experiments, and iterative in that each activity yields input to the next activity, which improves on the results of the previous activity. The different research methods and how we have used them are covered extensively in chapter 3.

1.3 RESEARCH SUBJECTS

While we have defined three overall research areas, these areas are have overlapping activities. Instead, we organise the research into five research subjects that impact the different research areas:

1. Participatory design for middleware,
2. context awareness,
3. architectural models,
4. first class connectors, and
5. semantic web technology.

While participatory design and context awareness are explicit in the focus of the project, the other three arised as a result of initial analysis of the domain and design processes. Semantic web technology was seen as a natural fit for describing services, since the existing services were available as web services, and because it is suited for modelling context. Architectural models and first class connectors are related in the sense that they are used for defining applications using the middleware.

Participatory design is used in the research process, and as such influence all three research areas. The research provides input to the context-awareness research area, but also integration since a context model implies either a common understanding of the model between systems, or translation from heterogeneous formats to the context model. The research in architectural models mostly deals with the use of models in software architecture, but also focuses on using models in middleware, and thus influences the architecture of the middleware. First class connectors is of course a part of the software architecture research area, but is also relevant as a method for handling integration. Finally, semantic web technology is used in the project for describing software architecture, modelling context, and providing tools for integration, so this also provides input to all research areas.

The research on context and architectural models are related to the areas of context awareness and software architecture respectively, while first class connectors and semantics relate to both software architecture and integration. Context also relates to integration, in that a context model describes a common understanding of the world, and how to represent it.

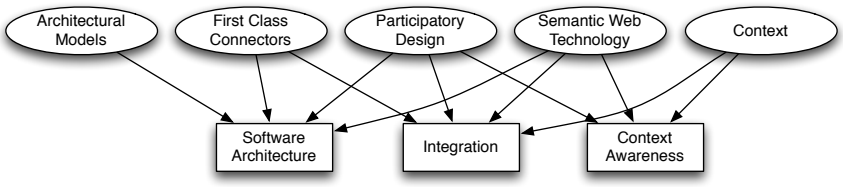

Figure 1.3: The five research subjects provide input to the research areas defined for the project.

1.4 Research Questions

The research areas and subjects give rise to a number of basic research questions that we wish to explore in what can be categorised as a "software architecture for middleware supporting context awareness and service oriented integration". The service orientation stems from the fact that a lot of the external services depicted in figure 1.1 are in different administrative domains, indicating that some form of service orientation will be beneficiary or even necessary. Furthermore, we set out to determine, whether first class connectors are useful in a service-oriented setting. Several authors, among them Shaw [107], have argued that connectors deserve first class status. They have already proven their worth in reified software architectures,

in fact, almost all architecture description languages include connectors in some form[84]. Reification of a software architecture in middleware shows that first class connectors are useful, also as a run-time entity. Semantic web technology as a research subject originates in the need for descriptions in the middleware. At least, we a way to describe context, but also architecture and services for use in integration. Thus, the main research questions of this thesis are:

Research Question 1 *How well do techniques from participatory design work for designing context aware middleware?*

Research Question 2 *Is a reified software architecture, including first class connectors, useful for deployment and integration for context aware middleware?*

Research Question 3 *Can the use of applications developed for a specific domain be used to design a general purpose middleware?*

Research Question 4 *Are semantic web technologies suitable for describing context, services, and architecture?*

The first research question deals with the research methods employed. It has been established that participatory design provides useful techniques for development in general, but to our knowledge it has not been widely used for engineering middleware before, although it is been used in the related field of developing a common software architecture e.g. by Büscher et al. [24] which also has some middleware aspects.

The second research question is concerned with using a formal software architecture description on runtime or deployment time. It is established that creating a binding between described architecture and runtime behaviour lead to better and more reliable software, especially in distributed systems. For instance, maintaining a view of the run-time architecture is important for implementing self-adapting systems [78]. It also concerns the integration between first class connectors and service oriented architectures. While several researchers have suggested that first class connectors are useful in software architecture as well as in programming languages, they seem to be somewhat overlooked at the middleware level, at least as formal entities in the middleware, although there are previous examples of their use.

The third research question deals with the fact that we do not wish to create a middleware that is too narrow in its scope. Rather, it should be usable across the different domains where context awareness and service oriented architectures are suitable.

The fourth question expresses our wish to use related technologies for different purposes in the middleware.

The main hypothesises of the thesis are:

Hypothesis 1 *Participatory design provides useful techniques for designing and evaluating middleware.*

Hypothesis 2 *A service oriented middleware using first class connectors provides abstractions that are useful for the implementation of applications that access information and services from heterogeneous administrative domains.*

Hypothesis 3 *Semantic web technologies can be used for a variety of purposes in a context aware middleware, including architecture description, integration, and context modelling.*

Together, these hypothesis express how a middleware supporting the vision formulated as the "Virtual Desktop" can be designed. The first hypothesis is concerned with the development process itself, while the two other hypothesises are concerned with the product being developed. We want to test a development process that is usually used for application development for the design of middleware. For the middleware itself, it has to work in a service oriented environment, so we explore the use of first class connectors in middleware for service orientation and use semantic web technologies for several purposes in the middleware. The purpose of the latter is to use technologies that are similar for different purposes, to provide better usability by reducing the knowledge required to use the middleware.

1.5 The rest of the dissertation

The rest of this dissertation details our work in experimental middleware development and discusses methods for carrying it out in part I. Part II describes middleware for context awareness and the scientific results. Parts of the thesis are based on work that has previously been published or submitted. Noticeably, parts of chapter 3 are rewrites of papers previously published by Kjær [71, 74], section 5.4 is based on Kjær [73], section 6.4 is based on Kjær [72], and section 7.1 is based on Kjær and Hansen [75].

Chapter 2

Background

As an introduction to the subject at hand, this chapter provides descriptions of some related work in the areas of middleware, ubiquitous computing, and semantic web technology.

2.1 Middleware

While middleware is a somewhat generic term, it is usually reserved for software systems which provide common abstractions for processes in distributed systems. We use the following, rather informal definition:

> Middleware *is a set of reusable components which provide services and common abstractions for processes in a distributed system.*

Other definitions include software that abstracts the differences between hosts, for instance virtual machines like the Java Virtual Machine (JVM) [79]. ObjectWeb provides another definition:

> *In a distributed computing system,* middleware *is defined as the software layer that lies between the operating system and the applications on each site of the system.*

This definition focuses on the fact that the middleware is neither part of the application nor part of the operating system. To clarify the concept of middleware, it is useful to distinguish it from the concept of a *framework*. While a middleware provides services and abstractions at run-time, a framework is a development environment defined by an API and possibly a user interface and a set of tools designed to simplify application development for a domain or set of domains [15]. As such, a framework may include middleware services for use on run-time. Furthermore, not all definitions of middleware exclude systems that are not distributed. For instance, some consider abstractions of the host environment as middleware.

Middleware emerged with the growth of distributed systems as a model for distributed services providing interoperability between heterogeneous devices and systems. Bernstein simply defines middleware as a service that sits between the platform and the applications, and describe what middleware should do [15]:

- Middleware meets the requirements of applications across domains,
- must have implementations for several platforms,
- is distributed,
- ideally supports standard protocols, and
- should support a standard API.

That middleware must meet requirements across domains is intended to ensure that it is reusable. However, these days several middleware systems are actually more or less domain-specific, in that they include services that are only usable in specific domains. Middleware systems that are not domain-specific are typically relatively low-level, for instance OSGi which implements component handling and communication, but not any actual services [114]. The requirement for implementations for several platforms has the same intention. The ideal of supporting standard protocols and standard APIs are intended to improve the chance of acceptance. As for APIs, Bernstein distinguish between *transparent* and non-transparent middleware systems. A transparent middleware systems uses an existing, standard API, so that it is usable in existing systems. As an example, most Unix implementations of distributed file systems use the standard POSIX file system API, allowing applications to access the distributed files without changes.

The traditional approach has been to *hide* heterogeneity and distribution by providing ways of treating remote resources as if they were local. In wired, static environment, this has proven useful, but in dynamic, wireless environments it breaks down, since applications often need to base decisions on information about distribution and the environment. Instead, middleware systems for ubiquitous computing focus on providing suitable abstractions for dealing with heterogeneity and distribution *without* hiding them.

A more organised view of middleware is provided by Schmidt by organising a middleware system into several layers (figure 2.1) [106]. While this classification of the layers in middleware is intended for embedded systems, it is useful for classification of other types of middleware too.

The host infrastructure layers hides differences between operating systems by providing common abstractions for communication primitives and other resources. The distribution layer provides high-level distribution models allowing location transparent services, allowing communication, method invocation and the like while hiding lower-level issues like addressing and discovery. Common middleware services are services which are domain-independent. The role of the common services is to handle global considerations like QoS provisioning, resource allocation, and

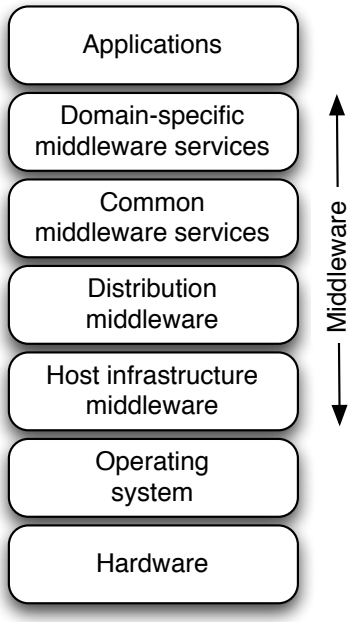

Figure 2.1: Middleware layers. Adapted from Schmidt [106].

persistence. Programmers can access these services instead of relying on lower-level distribution abstractions. Domain specific services are tailored for a particular class of systems, and are usually only reusable within the domain as they embody knowledge of this particular domain. For instance, a service that maps coordinates to symbolic locations can be used by most location-aware applications, but not by applications that do not need location. A service that maps coordinates to fields would only be useful in plant production.

A layered view on middleware is useful for design, but does not necessarily carry over directly to actual middleware systems. In fact, many middleware systems are somewhat domain specific and mix common- and domain-specific services without considering the difference.

In ubiquitous computing, middleware is especially important since it removes the burden of handling heterogeneity, mobility, and rapid changes in the environment. One obvious indication of this is the number of research papers describing various specialised middleware for ubiquitous computing. For instance, Ranganathan and Campbell describe a middleware using agents in a context-aware manner specifically targeted for their smart space environment Gaia [96] (described in section 5.4.3). Their observation is that the agents need to be context-aware to fulfill their role. Others, including Sivaharan et al. [108] have looked at publish/subscribe for use in middleware targeted at ubiquitous computing. In their case, they also investigate reconfiguring the middleware on deployment or run-time, allowing for use with heterogeneous networks and different publish/subscribe mechanisms allowing

developers using the middleware to choose between topic-based, content-based, or other available mechanisms. An example of a middleware that goes in a different direction is the Hydra middleware [101]. Hydra is intended to provide a widely deployed middleware for intelligent networking of embedded systems, and for this reason has a strong focus on supporting systems of different type, i.e. centralised and decentralised, and different domains.

2.1.1 WEB SERVICES

To overcome challenges in computing on an Internet scale, *Web Services* have been suggested as a communication model suitable for integrating data and services from heterogeneous sources across the Internet [34]. As such, Web Services are seen as the next-step for integration following the emergence of the World Wide Web in the nineties. The stated goal of Web Services is to enable Service-oriented-Architectures using existing web technology. As such, web services build on previous specifications by the World Wide Web Consortium and others, including

- XML Schema [47],
- XML [20], and
- HTTP [50].

Using these existing technologies, a number of new specifications have been created forming a framework for implementing Web Services.

The Web Services framework consists of three different standards at the basic level, but one of them, the Universal Description, Discovery, and Integration (UDDI [93]) is only used sparsely. The remaining, more important standards are WSDL [19] and SOAP [88]. WSDL uses XML Schema and XML to define services, including descriptions of data and operations on the service. SOAP is a communication protocol which defines how messages can be marshaled, typically, but not necessarily, embedded in HTTP messages.

Besides these basic standards, there are a number of extensions for various tasks which are useful especially in enterprise settings. These include standards for security, messaging, and transactions. However, in general only the basic Web Service standard can be assumed to be implemented in any given solution, and only a few of the other standards are in common use. Notably, SAWSDL (previously known as WSDL-S [3]) is an extension which adds semantic descriptions to WSDL [49] and is described in more detail in section 2.3.

From a middleware perspective, Web Services are simply another, relatively low-level, set of coordination protocols. The large number of extensions in the WS-* family, however, provide protocols on a higher level of abstraction, although still at the domain-independent layer, as described in section 2.1. Some of the members of this family include WS-Resource for sharing access to a common resource in the form of data[35], and WS-Notification for publish/subscribe[57].

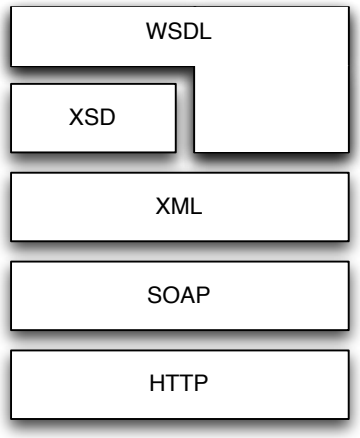

Figure 2.2: Layered use of description technologies in Web Services.

2.1.2 Architectural Views

Traditional views of middleware include layered models, for instance the one presented by Schmidt [106]. The layered model is useful, since it ties directly into the OSI reference model [122], and clearly defines where middleware fits into this model (figure 2.3). However, this is by no means the only useful view of middleware. While layers gives us an overview, it does not necessarily show how the middleware is used or its internal structure.

Another useful view is the *Component and Connector* view [12]. This view shows the run-time organisation of the middleware, by showing the components of the middleware and the connections between them (figure 2.4). However, the component and connector view is also useful for representing the run-time configuration of an application using the middleware. This can be more fully utilised by designing the middleware around the notion of components and connectors. The middleware can encapsulate coordination in connector abstractions, and let services be implemented as components. As an example, OSGi uses *bundles* that plug into the middleware, which in turn provides all coordination between components. A bundle is simply a Java .jar file with extra meta-data. This essentially corresponds to the middleware implementing a single connector and the bundles being components implementing services.

2.2 Ubiquitous Computing

While ubiquitous computing is becoming a reality, it is nevertheless interesting to look into the historical development and visions, and how they are, or are not, becoming reality today.

With the emergence of wireless networks and a plethora of small, network-enabled devices, computers have become a part of everyday lives. The trend for com-

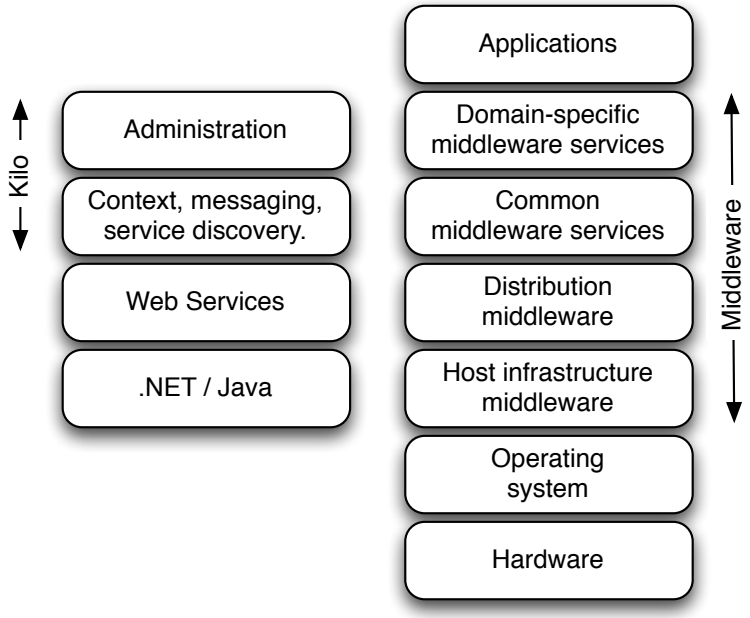

Figure 2.3: Kilo.Two middleware in relation to the middleware layers defined by Schmidt [106]. Details in section 4.2.2.

puters to become "invisible" was first observed by Weiser [118]. In this case, invisible means that the technology itself is no longer the focus of attention, in the same way that we do not focus on writing technology, but merely see the words everywhere in our environment. This is a function of human psychology and not technology, but technology can use this knowledge to design computer systems which are suited for the way human psychology works.

To investigate the disappearing technology, Weiser and his colleagues looked into a typical office setting. Their suggestion was to use three different kinds of computers:

- tabs,
- pads, and
- boards.

Tabs are small computers the size of post-it notes, pads are the size of note-books and boards are the size of white boards. The fundamental idea is that these three different kinds of computers meet different goals.

Tabs are intended for accessing notes and other small pieces of information. Ideally, one would have a large number of pads on the desk, each showing a different piece of information, in much the same way as post-it notes could be used.

Pads can be used for taking notes, but also for accessing information. Pads should be available in the environment, so people can pick them up when needed

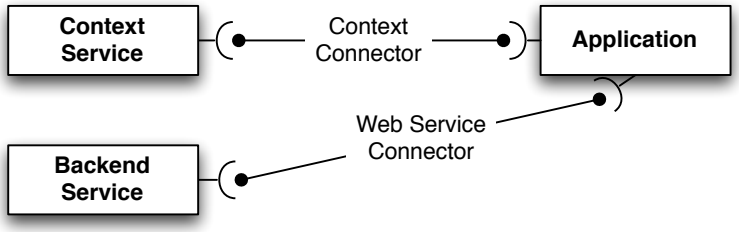

Figure 2.4: Components and connectors in a simple application using a **Context** service and remote **Backend** web service.

and use them. For instance, a meeting room may contain a number of pads for the participants in a meeting.

Boards replaces white-boards in meeting rooms and offices. They are large displays which can be used for displaying the same information to a group of people and collaboratively editing documents.

Pads, tabs, and boards are connected in a wireless network using radio-frequency and infrared technology. Due to constraints of computers at the time, most computations take place in central computers, with devices in the environment only showing a graphical interface (figure 2.5).

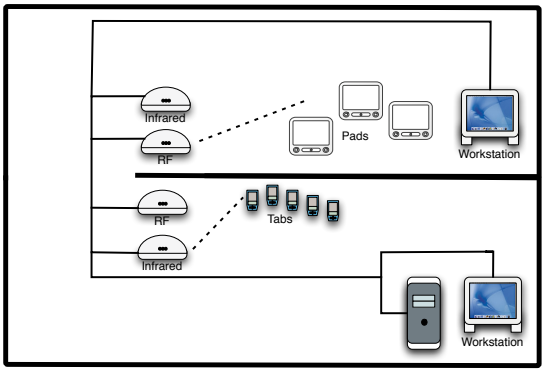

Figure 2.5: Tabs and pads in a ubiquitous computing environment. Adapted from Weiser [118].

Active Badges play an important role in the ubiquitous environment. An Active Badge replaces name-tags, and broadcasts the identity of its wearer to the environment. This allows for the system to know where workers are, and is used for instance to forward calls automatically to the office the worker is in.

While the technology employed seems ancient by now, with small and relatively powerful devices common, and wireless ethernet everywhere, the ideas still remain valid. Furthermore, about a decade after Weiser first formulated his vision, technology finally began to catch up to the vision, as observed by Satyanarayanan [102]. He described the new challenges and possibilities that had arisen in the previous

years. Notably the rise of mobile clients, at then mostly in the form of laptops. This opened up new areas of research, including

- mobile networking,
- mobile information access,
- adaptive applications,
- energy-saving on devices powered by battery, and
- location-sensing

in the area of mobile computing. Most of these problems are related to problems that arise specifically from carrying devices around without access to a static infrastructure, or at least moving between different static infrastructures.

Today, mobile computers often take the form of devices that are easily carried, like netbooks and advanced mobile phones. Interestingly, these two device categories almost corresponds to Weiser's tabs and pads, in that netbooks are suitable for note taking, while mobile phones allow for quick note taking, but fall short of Weiser's idea of cheap devices of which and individual user might use several at any given time. The need for a large number of devices is largely offset by the increase in processing power and graphical capabilities of todays hardware. Similarly, boards ad described by Weiser have partly become reality in the form of SmartBoards that allow for simultaneous interaction by several participants, but typically only runs a standard Windows environment, and do not as such provide easy integration in a larger environment, although it is certainly possible to use them for creating such environments.

2.3 Semantic Web Technology

Semantic web technologies has shown promises of helping to organise the increasing amount of data available today. First envisioned by Berners-Lee et al. as a technology for allowing computers to understand and manipulate information available on the Web [14], the technologies are now used for other purposes, like services. Conceptually, the *Semantic Web* is an extension of the existing World Wide Web. Additional information is encoded into web-pages, allowing agents and other software to make meaning of the contents of the page.

To accomplish this goal, some standards need to be in place:

- Structured collections of information,
- sets of inference rules, and
- systems for automatic reasoning.

Contrary to traditional knowledge systems, which are closed in the sense that things that are not part of the system do not exist (or in terms of logic: if it is not true, it is false) [98], the Semantic Web is open. Some questions might not be answerable using the available data, but this does not imply that it is false or does not exists, simply that it cannot be answered. That is, the fact that something is not known to be true does not make it false.

The suggestions put forth by Berners-Lee et al. involve using XML and RDF to describe arbitrary structured data and meaning respectively, and URIs to identify entities.

Another component of the Semantic Web is *ontologies*. The need for ontologies arise from the tendency of human language to use different identifiers for the same concept. To enable computers to understand that two different identifiers actually refer to the same concept in two different systems, an ontology describing this fact must be available for the reasoning engine. The typical ontology is a taxonomy of concepts along with inference rules.

Of course, for ontologies to be useful, there has to be an agreement on which ontology to use. At this time, this may be the most obvious obstacle to the widespread implementation of the Semantic Web. Furthermore, developing ontologies is a time-consuming and expensive process, involving compromises and a resulting danger of "death by committee" due to different understanding of concepts that are either the same or similar enough to be confused.

For instance, the Food and Agriculture Organization of the United Nations (FAO) maintains a geopolitical ontology written on OWL that providing names and codes for territories and groups [48]. The ontology tracks geopolitical information and changes since 1985, and includes information about groups and areas in the world, like the relationship between non-governing regions and the entity that controls them. The classes and object properties are relatively few, and shown in figure 2.6.

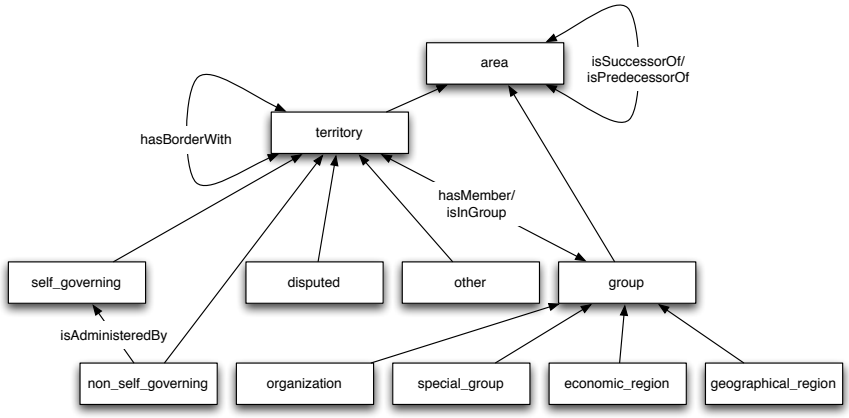

Figure 2.6: The FAO Geopolitical ontology [48], excluding data properties, the OWL Thing concept, and individuals. Arrows without a label denotes the rdf:subClassOf property, which has been left out for readability.

However, some work has been done to establish standard ontologies. For instance, the Standard Upper Ontology Working Group under IEEE is working on the Suggested Upper Merged Ontology (SUMO) for use in information science [91]. The goal is to have a base for more specific domain ontologies. The upper ontology is shown in figure 2.7.

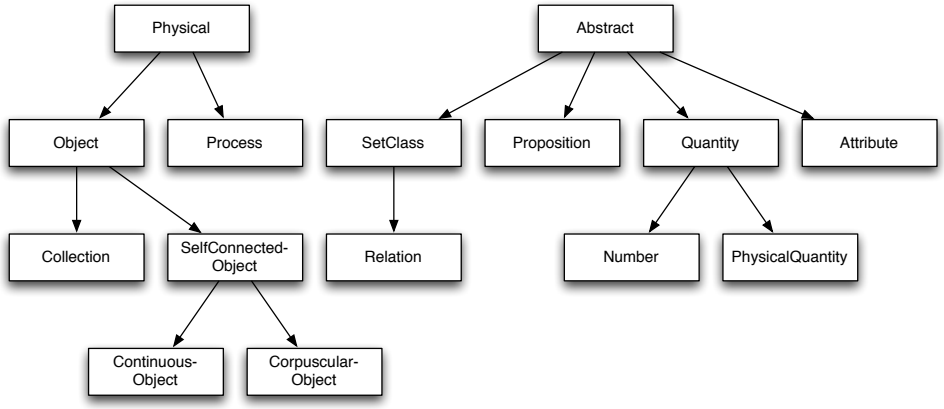

Figure 2.7: SUMO upper ontology, as defined in Niles and Pease [91].

While the RDF use of triples to describe concepts is useful, e.g. for ontologies, it has a number of limitations [60]:

- It is not possible to choose between conflicting definitions of the same concept,
- it has poor support for versioning,
- it has only minimal support for interoperability,
- it cannot express inconsistencies,
- it is not very expressive (e.g. no cardinality constraints, equivalence, etc.), and
- RDF Schema is very complex.

The suggested solution to this problem is the Web Ontology Language (OWL) [83]. OWL builds on top of RDF, in the sense that every RDF document is also an OWL document. OWL comes in three different sub languages:

- OWL Lite,
- OWL DL, and
- OWL Full,

with each variant being an extension of its predecessor. OWL Lite allows for simple hierarchies and simple constraints. OWL DL is more expressive, but still allows for algorithmic completeness for inference in the ontology. That is, it all conclusions are

computable and finish in finite time. OWL DL closely corresponds to description logics, hence the name. In fact, it can be seen as a variation of SHOIN(D) description logics, with the ontology being the knowledge base [65]. OWL Full is an extension of the expressibility of RDF Schema and allows for meta-modelling, like concepts of concepts and properties on concepts. Besides providing formal semantics, OWL provides a number of language features not available in RDF. For instance, it is possible to define cardinality constraints on properties, that properties are inverse, and datatypes.

While OWL provides descriptions of classes and instances in an ontology, a separate language is needed to describe inference rules. SWRL is based on OWL Lite and OWL DL and extends it with a binary Datalog subset of RuleML [64].

SWRL has a formally defined abstract syntax, as well as a human readable syntax and an XML serialisation, and describes inferences using Horne-like rules on the form

$$antecedent \Rightarrow consequent$$

where both antecedent and consequent are of the form $a_1 \wedge \ldots \wedge a_n$, where variables are indicated by prefixing them with a question mark, as in $?x$. This allows for expressing properties that are functions of other properties. For instance, if in an ontology of genealogy, there is a *parent* relation and a *brother* relation, we can express the *uncle* relation as ([64])

$$parent(?x, ?y) \wedge brother(?y, ?z) \Rightarrow uncle(?x, ?z)$$

Other built-in functions allows for expressing more complex inference rules, including simple mathematical operators, comparison operators, and functions that create new individuals in the ontology.

While OWL provide a foundation for adding semantics to the web, it does not address semantic annotation of web services. To this end, OWL-S was developed as an ontology for describing not only the structure of a service, but also how it is accessed [22].

OWL-S consists of an upper ontology for services (figure 2.8), and a model for each of the concepts of the upper ontology:

- **ServiceProfile**,
- **ServiceModel**, and
- **ServiceGrounding**.

The service profile describes what the service does in a way that is appropriate for matching agents, including quality of service, limitations, and requirements for the requester of the service. The service model describes how to use the service, by describing the semantic content of requests and the step by step process to achieve outcomes. The service grounding describes in detail how the service can be accessed. This will normally include port number, message formats, communication

protocols and the like. There are two cardinality constraints in the upper ontology: A service is described by at most one service model, and a grounding is associated with exactly one service.

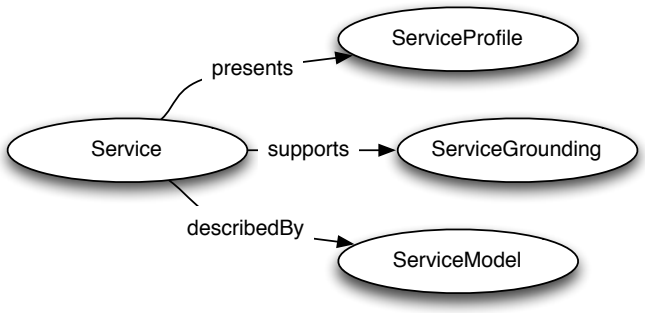

Figure 2.8: An upper ontology for services as defined in OWL-S. Adapted from Martin et al. [22].

OWL-S intentionally only defines a single upper ontology for profile, grounding, and model, but these are merely suggestions. For instance, the suggested grounding only supports web services defined by WSDL. For services not using WSDL another grounding will have to be used. The proposed ontology for the service models allows for describing services as processes, by defining **Process** as a sub-class of **ServiceModel**.

SAWSDL is nother suggestion for adding semantic information to web services [2]. Unlike OWL-S, SAWSDL has a bottom-up approach, in that instead of creating a framework describing services and grounding, the only defined grounding being WSDL, SAWSDL adds semantic description to the existing WSDL specification independently of the language used to represent the semantics. The goal is to include the semantic information into the existing syntax of WSDL to avoid building a new model of services on top. SAWSDL utilises the fact that WSDL is extensible to include extra elements that map input and output types to concepts in ontologies and an element for categorising services according to ontologies specified by service registries.

Chapter 3

Participatory Design

Participatory design has a long tradition in computer science, especially in the Scandinavian countries. Early experiments with user involvement would involve interviewing the users about how they thought computer systems should work. More important, developers would study how daily work was carried out and would use prototypes or mock-ups of systems to develop systems which would function better once completed [44]. Other employed methods include ethnographically studying workers during field studies even before development, to create systems which will fit into their daily work as it is carried out already, ideally providing a system which will allow for a relatively flat learning curve, and which will not disrupt work when it is deployed.

Later uses of participatory design have attempted to use it for non end-user applications. For instance, Corry et al. [32] have successfully employed participatory design in creating an architecture developed across several physical locations. Extending participatory design to other disciplines involves changing the methods, with some changes more obvious than others. As an example, when the end-user is no longer the participant in the design process, the artifacts used in the process will change from focusing on the application or the system being developed for the end-user, to artifacts that focus on e.g. software architecture.

In this chapter we look at the participatory design processes in the Kilo project. We also discuss the impact it has had on the middleware design, and how well participatory design techniques match the development of middleware.

3.1 Research and Development Methods

Originally, participatory design was mainly used as a technique for involving end-users in designing the computer systems they would later be using in their work. This was seen as essential to creating systems which would support existing practice, instead of disrupting them [17]. This was closely related to the Scandinavian ideal of democracy in the work place, and the involvement of workers was seen as essential for allowing democracy to continue in the light of new technologies being

introduces. In this use of participatory design, it is carried out using artifacts which represent the envisioned user interface or perhaps even work environment of the end-users for instance by using mock-ups. Ehn and Kyng have looked into the use of mockups extensively [44]. In the late seventies or early eighties, when a large portion of development projects dealt with computerising work that was previously not carried out by computers, mockups might include "building" the workstations in cardboard, and then letting workers pretend to carry out work as they would do with this new system, while researchers observe them and later interview them about their experience.

Similarly, as computers became everyday items, but ubiquitous computing was still unknown to most people, mockups using several devices can be used to introduce users to the possibilities. Thus, to investigate what role ubiquitous computing might play in everyday settings, mockups of future systems can be used to investigate how workers might interact with them. Bardram has used this for investigating the use if ubiquitous computing in health care [8].

Corry et al. describe how methods from participatory design can be used in the large, distributed software development project *PalCom* [32]. In the project, development teams at different physical locations work together to create a common architecture. To ensure cooperation towards the common goal, a team of *Travelling Architects* was created which traveled between development groups and employed active user involvement methods to further reaching the goal. The technique is created for a project with the following central challenges:

- Development teams are distributed,
- development is iterative, experimental, and incremental, and
- central control is limited.

More precisely, they define Traveling Architects as:

> *A group of architects responsible for maintaining software architectural assets in a distributed development project by visiting development sites in order to design, evaluate, and enforce a software architecture in active collaboration with developers and possibly end users.*

In participatory design of software architecture, the design artifact is the software architecture itself, in the form of UML diagrams, architectural prototypes, and verbal accounts, using different views depending on who is present at meetings.

The authors observe, that the most important responsibility of the traveling architects turned out to be communicating the architecture, not actually designing it.

Their experiences show, that participatory design techniques can be used for other things than originally envisioned. While it has traditionally been used for creating functionality and architecture of prototypes, the PalCom project extends the methods for use in architecture of the framework [23].

Some of the techniques used in participatory design include:

- Field studies study users as they work to provide insight into the domain by studying current practice.
- Interviews with individual practitioners can provide in-depth understanding and clarification of phenomena observed during field studies.
- Focus groups are invited groups of experts who are gathered to discuss a specific problem, whether to gain insight into the problem or solve it.
- Prototypes are early implementations of the end-system which can be used for evaluation of various aspects of systems.
- Future labs are focused sessions where practitioners are asked to perform tasks using specific tools developed for the purpose to determine their usability.
- Bricolage brings together different technologies to test them for different tasks.

Field studies have been used extensively in recent years for exploring how people work before designing systems for them. This has partly been driven by anthropologists with interest in computer science, and the technique is similar to the one an anthropologist would use when observing people. For instance, Blomberg et al. use field studies as a basic element of "Work-Oriented Design" [18] . The main observation leading them to use Work-Oriented Design is that *usability* is not the same as *usefulness*. While usability can, to some degree, be studied in the computer lab, the usefulness of the of computer system depends on how good it is at supporting the daily work of the users, and it is therefore necessary

Interviews are often combined with field studies, and can be used for clarifying observations made during field studies. They can also be used before field studies for preparation. Typical subjects of interviews include end-users as well as developers and managers. Interviewing both end-users and management will often reveal schisms in the view of work procedures between the two groups and observance towards these schisms can be crucial in providing a system which support work as carried out, as opposed to work as imagined by management and to allow workers influence on the design process [17].

Focus groups are well-suited to determine requirements that do not immediately follow from field-studies and interview. Conducting controlled focus groups with relevant participants may provide an idea of how a domain might evolve in the coming years, or insights that users are not aware off [21]. For instance, end users typically do not have any insight into technology above a basic level like knowing how to use a particular system. Focus groups on the other hand gathers people who can provide an insight into the problem at hand. Thus, focus groups can be tailored for the specific purpose. If the need is for experts to help develop appropriate interfaces to systems, a focus group might consist of HCI experts, while if the need is for insight into existing systems, developers of those systems may be included into the focus group.

Prototypes are the main artifacts in most participatory design approaches, and often also the goal of the design process. One of the main advantages of prototypes is that they can be evaluated through actual use, for instance by using in-situ prototype experiments, where end-users use the prototype in their daily work or in acting out a scenario in a real setting or by using the prototype in a future lab setting. How a prototype can be used depends on what type of prototype it is. In particular, Floyd distinguishes between *horizontal* and *vertical* prototypes [51]. A horizontal prototype explores the structure of the system under development, but does not include any functionality, while a vertical prototype implements some functionality, but typically not the complete system. In this respect, a horizontal prototype as an artifact in participatory design is akin to a mockup, but may offer some level of interactivity which may help end users envision how it may be used when completed, for instance by allowing the user to browse through the user interface.

Future labs are controlled environments, where end users and other actors in the design process together explore prototypes by simulating authentic processes from their daily work to discover if the prototype is appropriate. Ideally, end users should go through complete activities they know from their daily work, but using the new prototype to carry it out.

Bricolage is the process of bringing technologies together in order to make them work, and is used to determine the usefulness of technologies in performing an array of tasks in different environments, thus testing if the proposed solutions are up to the task.

When using the above techniques, it is important to understand their properties, including their limitations. When using the techniques, the activities focus on a single instance of current practice in the case of field studies and interviews, or a single vision of future practice in the case of prototypes, future labs, and bricolage. Focus groups might discuss different possible futures or provide a wider overview of current practice, but this depends entirely on the focus of the individual session and who are participating.

Büscher et al. describe a participatory design process [24], using prototypes, future labs, and bricolage for "triangulating" the current and future practice. The goal is to maintain a collaboration between "design" and "use" and grounding imagination. The idea is that future practice can be triangulated between:

- *Means*, the tools and technologies available.
- *Environments* in which the future tools will be used.
- The *job at hand*, which task is being carried out.

This is illustrated in figure 3.1. Specifically, they propose using bricolage as a way of fixing the means, while changing environments or the job at hand. Similarly prototypes in-situ can be used as a way of fixing the environment, while allowing changes to means and the job at hand, and future laboratories can be used for exploring what happens when the job at hand is fixed, but the means and the environments can be changed.

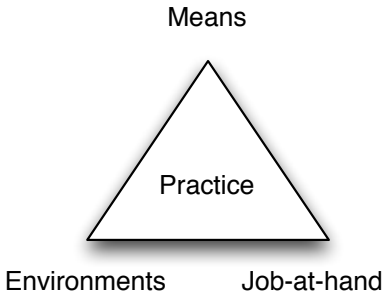

Figure 3.1: Future practice can be triangulated by fixing either means, environment, or job-at-hand, and varying the two others. Adapted from Büscher et al. [24].

3.2 Participatory Design in Kilo

During the 36 months of the Kilo project, participatory design have been used several times in every iteration. Since the Kilo project is a middleware project, the end users are developers, so they are participants in the design of the project. To evaluate the middleware, prototypes of applications where developed for which the end users where users of applications. The end users of the middleware on the other hand are developers, so there are two domains: The agriculture domain where applications will be used, and the developer domain where the middleware will be used.

Initial insight into the developer domain was acquired by meetings with developer groups. The developers would explain their existing products, as well as their development process and visions for the future. To give the developers an insight into the domain and the environment where the middleware as well as applications will be deployed, field studies were carried out.

To gain further insights into the agriculture domain, interviews with individual workers where carried out. Workers were, for instance, asked to describe a typical work day and to come up with and "exceptional situation" they might encounter. The description of the typical work gave us a baseline for interpreting observations during field studies, while the descriptions of exceptional situations gave us an idea of the type of scenarios we might have to support. Especially exceptional situations can be difficult to discover from studying workers performing their daily tasks, unless the studies take place for an extended period of time, which is often not possible due to time or budget constraints.

Field studies were used to determine both the usefulness of context awareness in general, and to give input to scenarios. This is meant to ensure that the research carried out later, as well as the prototypes developed have a solid grounding in reality. Philosophically, this is similar to Grounded Theory in that the focus is on observing *reality* and then proceed based on those observations [55].

After prototypes were developed, they were deployed in the daily environment of the workers, and feedback from their use was then used for refinement and input to the next iteration.

Table 3.1 shows each instance of participatory design activities in the project.

Figure 3.2: Picture from field studies at a farm in southern Denmark - agricultural worker examining a sow for pregnancy.

If more resources had been available, it would have been beneficial to also do field studies of developers of existing applications, since they are the end users of the middleware.

An important aspect of the project is that there are actually two kinds of end-users: Agricultural workers and developers of agriculture applications. The workers are end-users of applications that use the designed middleware, while developers are end-users of the middleware and accompanying development frameworks. This implies the importance of designing the middleware so that it will be useful for developers in that is should allow for creation of desired applications in an way that is easy to understand and use. The design activities with end users of applications focus on artifacts related to those applications (figure 3.3) while design activities with developers focus on the middleware itself (figure 3.4). The middleware can be represented by an architecture description, whether formal or informal, or a prototype of the middleware.

The agricultural workers have little or no knowledge of the technology employed in the project, which is to be expected in participatory design processes. However, for historical reasons, a lot of developers do not necessarily have a formal education in software engineering. They are experts in the sense that they have a large experience in developing applications for agriculture, but do not necessarily have knowledge of development and design processes besides those they use in their daily work. On the other hand, the developers are often domain experts in their own right, either due to an education in agriculture or due to experience.

Table 3.1: Participatory Design Activities.

Activity No	Type	Purpose
1	Interviews	Interviewing developers
2	Focus group	Input from end developers
3	Field studies	Studies of agricultural workers
4	Focus group	Evaluation of scenarios
5	Focus group	Input from external developers
6	Field studies	Studies of agricultural workers in pig production
7	Design workshop	User interface
8	Design workshop	User interface and functionality
9	Prototype test	Test with end-users
10	Prototype deployment	Initial deployment
11	Prototype evaluation	Initial evaluation
12	Prototype deployment	Deployment of refined prototype
13	Focus group	Input from end developers, second prototype
14	Focus group	Evaluation of second prototype design

3.3 Using Artifacts

During the participatory design process, we employed an array of different artifacts for the different design activities:

- *Scenarios*. Textual descriptions of how a tasks or series of tasks would be carried out using new technology. Scenarios are used for communicating visions of the future.
- *Mock-ups*. Models of a future systems. This can be anything from a drawing of a suggestion of a future user interface, to a full-scale model of a future work place with non-functional versions of future devices. Mock-ups are useful for communicating future technology, especially how it is supposed to be used.
- *Middleware prototypes*. Executable prototypes of the future middleware, which may be more or less complete. They are used for testing aspects of the middleware, but also for communicating architecture.
- *Application prototypes*. Executable prototypes of the future application. They are used for testing application design.
- *Architecture descriptions*. More or less formal descriptions of the future software architecture. This can be an informal drawing, a textual description, or a formal description using a graphical model such as UML or a formal architecture description language. Architecture descriptions are used for communicating architecture between developers.

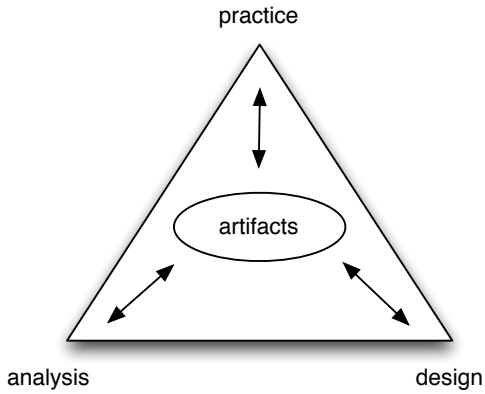

Figure 3.3: Artifacts in participatory design. Adapted from Mogensen and Trigg [89].

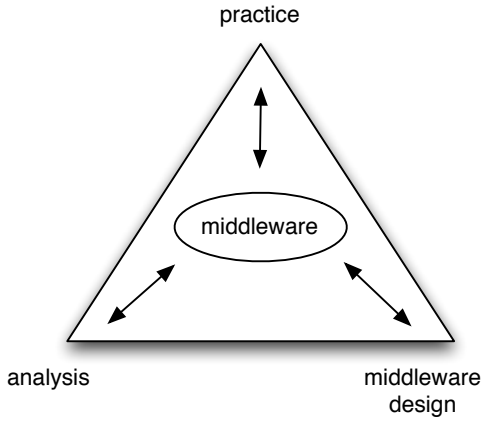

Figure 3.4: Participatory design of middleware. For middleware, the artifact becomes the middleware itself.

Each participatory design activity typically included the use of a particular artifact (table 3.2), although for instance field studies obviously do not, since the idea is to study existing work practice. For some activities, the purpose of the activity is to work on the artifact, for instance making changes to a scenario or mock-up. For other activities, the artifact serves as a basis for other discussions.

While initial field studies gave the developers an overview and "feel" of the domain, it is important to ensure that this initial insight gives rise to useful applications. Scenarios are useful in this respect, since they describe the envisioned application in a language that is familiar to both our developers and our end-users. Thus, it becomes important to use scenarios that speak to both end-users and developers. Preferably, the scenario should use technology in a way that does not confuse application end-users, while allowing the developers to envision how work will be carried out using the new technology, as well as the technological consequences of the scenario.

Table 3.2: Use of artifacts in the development process.

Activity No	Type	Artifact type
3	Field studies	Scenarios
4	Focus group	Scenarios
5	Focus group	Scenarios
7	Design workshop	Mockups
8	Design workshop	Mockups
9	Prototype test	Horizontal application prototype
10	Prototype deployment	Vertical middleware prototype
11	Prototype evaluation	Vertical application prototype
12	Prototype deployment	Vertical applications and middleware prototype
13	Focus group	Scenarios
14	Focus group	Architecture description

Mockups are useful as a discussion point of details of the application. In our case, we used GUI mockups as starting points for discussions with developers as well as users. Specifically, a mockup like the one in figure 3.5 (the original mockups are included in appendix C) where shown to developers to tap into their existing expertise in creating PDA applications. Similarly, the mockups where used as a starting point for discussing functionality with application end-users. These mockups allowed for very concrete discussions of the functionality of the system. At a concrete meeting, number 8 in the tables, a farmer was shown the mockups and asked to comment. This resulted in a discussion of interface as well as functionality. For instance, the farmer asked questions like:

- "what happens when I press here?",
- "where is the data stored once I enter it here?", and
- "is it possible to retrieve the data for use in other systems?".

Besides explaining our envisioned application by answering these questions, it gave rise to further discussions, such as which other systems the farmer would like to export the data to. In this case, he would like to export it as an Excel spreadsheet or access it from a web page, and where it would be most appropriate to store the data. Where the data is stored might seem irrelevant to the farmer, but actually becomes important, since it requires a physical computer somewhere, which has an impact on the cost of running the system, depending on which solution is used. This incidentally also shows another important aspect of developing for agriculture: Acceptance is to a large degree determined by one of two things:

- Can it save money, or
- can it save time?

Basically, a farmer will invest in new technology for either of these two reasons. While the first is obvious, the second is a bit more subtle. A modern Danish farm is to a large degree similar to a factory with the farmer as managing director, and a large part of the farmers work day is used on administrative tasks, so technology that reduces the responsibilities of the farmer or allows him to perform his tasks easier might also lead to acceptance.

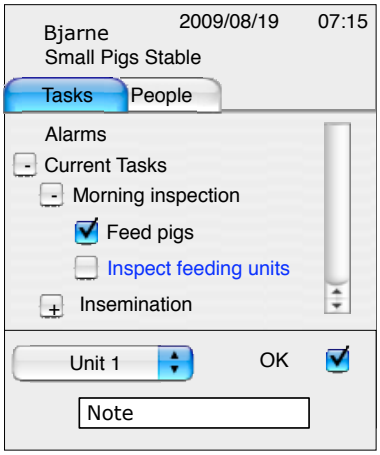

Figure 3.5: Mockup of GUI for PDA application supporting daily tasks. Adaptation of original mockup, included in appendix C.

The middleware prototype is especially important, since it is the actual product of the development process. However, if we look at the participatory design process, it has only been used directly in two activities (three if we include the architecture description from activity 14, which was a description of the second iteration architecture). The reason for this is that it is difficult to use middleware as an artifact directly. In fact, in the case of activity 10, it could be argued that what we actually used was the application *and* the middleware prototype. In that case, we used a very simple application to test the middleware in a real-world setting, by having users enter notes into an application during an ordinary work-day. Thus, the main tests of the middleware prototype were actually deployment of the complete prototype system. We also had an intention of testing the middleware by having actual developers use it to implement applications, but this had to be abandoned due to constraints on the part of the developers and a focus group was used instead. This would have allowed us to do a sort of bricolage with the development process being the practice studied instead of studying the practice of agricultural workers.

Since it is difficult to use the middleware prototype directly, the application prototype becomes the important artifact in the design process. Application end-users can use and comment on the prototype without necessarily having an understanding of the underlying middleware, and it becomes the responsibility of the designer to sort results and comments into things that affect the middleware, and things that affect the application prototype. We used the application prototype by actually de-

ploying it at a farm, and then observing its use and interviewing workers about their experience. However, horizontal prototypes were also used at earlier stages of the development process as a kind of interactive mock-up, which allows for the user to actually browse through the user interface to get an idea of whether it is suitable for actual use.

While architecture descriptions are not traditionally used as artifacts in participatory design, with Corry et al. as a notable exception [32], they are useful when the users participating are either software architects themselves, or at least developers with an understanding of software architecture. One of the goals of our second prototype was integration with existing systems, and descriptions of requirements and architecture of middleware proved useful in communicating with developers.

The above illustrates, that an important aspect of any participatory design activity involving the active involvement of users, and not just observations, is choosing the right artifact. While architecture descriptions are useful for discussions with system architects and developers, application end-users need something more tangible, or at least more readily recognisable.

3.4 Results

The various participatory design activities in the project yielded results on several levels. For instance, interviews with professionals in the domain and field studies of agricultural workers, along with focus groups and analysis of the domain, allowed us to create an initial list of requirements (appendix A) and initial suggestions and refinements of scenarios expressing the intended middleware and prototype application.

An example of very concrete results of field studies and interviews is an initial list of contextual concepts that must be supported in an eventual context model. At this time, it had not been decided how context awareness should be supported, although it was clear the some form of context awareness was necessary. However, it soon became apparent that applications would run on PCs as well as PDAs, meaning we would potentially have to support both internal and external context (see chapter 5), and we were also able to create a list of concepts we would have to support. This provided a grounded starting point for creating the eventual context model.

Prototype deployment in actual work settings proved useful for testing both the application and the middleware prototypes. For the middleware, some requirements were not discovered until the middleware was put to the test by actual workers. This included quirks due to the actual PDAs being used being slightly different from the ones used for development, and for instance discovering problems with battery life due to the way PDAs reacted to the application. Specifically, we discovered that some PDAs would turn on the screen every time a network connection was made, increasing the battery consumption of frequent network connections significantly.

3.5 SUMMARY

In this chapter we have looked into existing and previous practice for participatory design, and described how some of the techniques have been used during our project. Especially, we detailed how artifacts proved useful, the importance of selecting the right artifact for the job at hand, depending on the participants in the participatory design activity, and the usefulness of application prototypes for middleware development.

While participatory design has mostly been used for creating end-user applications, we have used it for driving the requirement gathering for middleware, giving us a slightly different focus. However, the techniques still proved useful when used with appropriate artifacts. For instance, when discussing scenarios with developers, we where able to determine requirements based on their knowledge of existing systems and infrastructure. Similarly, field studies and interviews with application end-users provide insight into requirements for the middleware as well as the application. As an example, how workers move during a day has an impact on which assumptions can be made about connectivity. I.e. some workers only work in stables, and constant connectivity can be assumed, while others move between environments with constant connectivity and environments without or with limited connectivity.

Participatory design techniques does not solve all problems though. The techniques we have used for application end-users are inherently qualitative, which implies that results obtained does not necessarily carry over to the entire domain. This is partly offset by focus groups and interviews with application developers and domain experts, who can be expected to have a broader insight into the domain. A full evaluation would require a wider deployment though.

Chapter 4

Prototypes

This chapter describes the prototypes developed in the Kilo project. It does not describe the scientific results from the creation or application of the prototypes, which are described in part II.

In the Kilo project, prototypes have been used as an important artifact in the development cycle, previously described by Kjær [74]. The Kilo project includes a number of challenges which originate in the heterogeneity of the domain:

- Heterogeneous devices and systems,
- heterogeneous deployment environments,
- heterogeneity of applications.

That devices and systems are heterogeneous is not really a surprise, since they come from a large number of different vendors. There is no incentive for vendors to use the same, or even compatible devices, to control machinery. Similarly, administration systems are created by more than one vendor, and unless external circumstances require it, there is no interoperability with other systems.

What may be a surprise is the heterogeneity of environments and the applications which will be deployed in the environments. This is mainly due to the fact, that while agriculture may be seen as a domain, it actually consists of a number of distinct, but interconnected, sub-domains (figure 4.1). In Danish agriculture, the most important domains are:

- Pig production,
- dairy and beef production, and
- plant production.

While these are the most important, there are other domains such as poultry, goats, sheep, nurseries, fur, and horses. However, since these are of lesser importance, there are fewer automated tools and administration systems available for these.

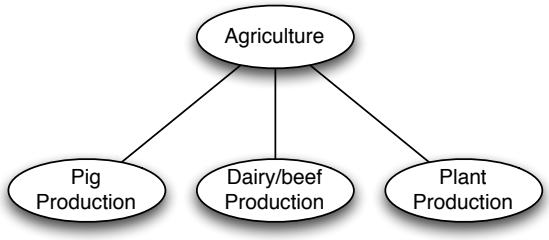

Figure 4.1: The most important sub-domains in Danish agriculture. For other countries or regions, the important sub-domains will be different.

As stated, these domains are interconnected. While it was common as little as twenty years ago for farms to produce at least pigs, cows, and plants, most farms today are large-scale, industrial production facilities, and will typically produce either pigs or cows. However, they will also have some plant production for two reasons. First, animal production produces manure, which can be disposed by spreading it on the fields. For this reason, Danish law requires farms with animal production to have a certain area of fields based on the number of animals they produce. Secondly, the plant production can be used for fodder, so the output of the plant production is an important management aspect, since there is a relation between output and the fodder needed to be bought.

An important part of the project has been to involve users in requirement gathering and evaluation. Prototypes play an central role, since they implement the requirements, and allows for users to experience the result. Observations and interviews with users gives input to the evaluation of each iteration.

Since we were tasked with designing middleware for a, to the researchers, unknown domain, we employed an empirical approach to software development, by building prototypes and testing them. Each iteration of prototypes consists of a middleware prototype as well as an application prototype built using the middleware.

In each case, the middleware consists of middleware services for Windows Mobile PDAs, as well as services to be deployed in the environment. The applications run on Windows Mobile PDAs, and are used for four primary purposes:

- Testing the middleware,
- test user reaction to the ideas presented,
- exploring the design space, and
- evolution of the middleware.

While the first purpose is the most important, the second purpose uses the applications to validate underlying assumptions in the project, mainly that context awareness is useful for the domain. Exploring the design space was noted by Floyd as one

of the useful features of prototypes [51]. It allows for testing ideas without building complete systems, making development cheaper. Finally, prototypes allows for feedback, resulting in new requirements for future iterations of the middleware.

4.1 Kilo.One

The first application prototype, **Kilo.One**, is relatively simple and tailored for use in pig production. The application itself supports the gathering of administrative data. Some of this data has to be collected due to legal requirements, while other data is gathered for purposes internally to the farm. Examples of data include:

- Live sucklings born to each sow,
- amount of medicine used in each stable,
- deaths among pigs,
- pigs sent to slaughterhouses,
- number of insemination attempts for each sow,
- sucklings moved from one sow to another,
- repairs carried out on equipment,
- orders made to suppliers,
- orders received from suppliers.

While this list is merely examples, it gives an idea of the different types of data collected on an everyday basis. The list contains both data which must be collected for external parties, mostly government agencies, and data which is only for internal use. For instance, to track production, births, deaths, and pigs sent to the slaughterhouse has to be collected due to a requirement to measure production size, while repairs on equipment is solely used in decision making. E.g., if the same machine continuously breaks down, it is time to replace it. Current practice is to collect data by taking notes, either in a notebook, or in dedicated charts. For instance, all information related to a sow that is pregnant or have sucklings will be recorded on a chart hanging above its sty, such as the one pictured in figure 4.2. The data is then later added to administration systems or ad-hoc systems. For sow-charts, this might happen once a week. The prototype application instead allows agricultural workers to enter information into a PDA, while using situation-awareness to limit the number of available options to the worker.

Figure 4.2: Chart for recording information about a particular sow.

4.1.1 APPLICATION PROTOTYPE

The application prototype supports a context- and a situation-model. While the context model is specifically tailored for the prototype, the situation model is similar to the one described by Meissen et al. [86], with a simple hierarchy of situations with increasing granularity. The current situation is then the most precisely defined based on available data.

Context in **Kilo.One** is relatively simple, and consists of a user profile containing the name of the user, and a simple location model, as illustrated in figure figure 4.3. As shown, the location model is hierarchical, and the situation model is in this case more or less a direct translation of location to situation, although this is mainly due to the limitation of the context model, and not a limitation of the situation model.

The situation model is a hierarchical graph of more refined situations, where each situation is defined by a location, and has a number of possible actions for the situation (figure 4.4). Often, context-aware systems would use tasks where we use situations. Our rationale for using situations instead of tasks is based on observations during field studies: First, agricultural workers will often carry out tasks in an overlapping manner. For instance, a worker carrying out the task "feed pigs", might discover that a pig is sick, and start carrying out the task "move pig to another sty", and then continue with the first task. However, short of having the user inform the system that a different task is being carried out, there is no way of detecting this. Furthermore, having the user inform the system would defeat the purpose of building a tool for assisting workers, and would instead force them to interact with it, even when not necessarily needed. A screenshot of the prototype is shown in figure 4.5. Instead, we use situations based on the current context of the user, in this

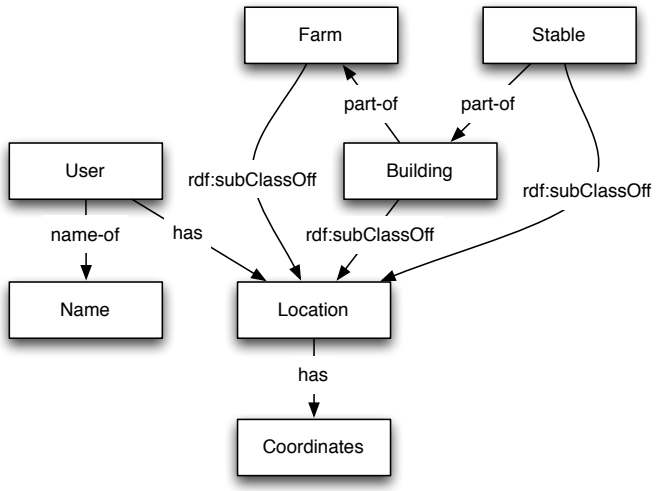

Figure 4.3: Ontology of the context model of **Kilo.One**. The central concept is *Location*, and the actual context model is implemented in an object oriented manner.

case the location. This is due to the observation that each location is associated with a number of possible actions. For instance, the action "sucklings born" is only relevant in the sow stable, while "sow inseminated" is only relevant in the insemination stable. So each situation has a number of relevant actions.

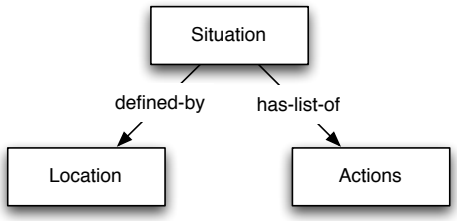

Figure 4.4: The **Kilo.One** situation concept.

To improve acceptance of the prototype, it was also integrated with an existing alarm system present at the farm. The system monitors various parameters on the farm and by default sends alarms as text-messages when values fall outside predefined limits. The integration simply consists of providing a service where alarms can be retrieved by the application. This could be done in a context-aware manner, but in the prototype, we simply show all alarms, since the number of alarms during a day is relatively small. An overview of the resulting system is shown in figure 4.6 with the deployment shown in figure 4.7.

For connectivity, we assume a static infrastructure with constant network access. While this is a simplyfication, the prototype is intended to be used at farms that have wireless network coverage of the entire farm. Workers may travel between farms, but when they carry out work, they will be within network coverage.

Figure 4.5: Screenshot of the **Kilo.One** prototype, showing the "Small pigs stable" location and the entry for the "medication" task.

Some of the ideas for the application prototype originate in existing software for PDAs that support pig production (e.g. DLBR Minigrisen; [37]). However, these are, to our knowledge, not in common use.

4.1.2 MIDDLEWARE PROTOTYPE

At the middleware layer, the prototype consists of a component and connector based framework for building applications. This approach was selected because of the benefits of reuse and adaptability in such a design, and because of research suggesting that first class connectors are useful for bridging the gap between interfaces at the class level and interfaces at the service level, as described in section 6.4. To support applications built within the framework, there are a number of middleware services running either in the environment or on the device:

- Local location service,

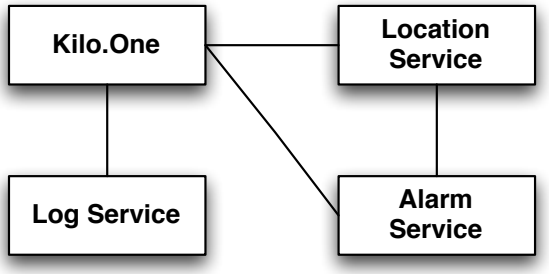

Figure 4.6: Overview of services and applications in the Kilo.One prototype.

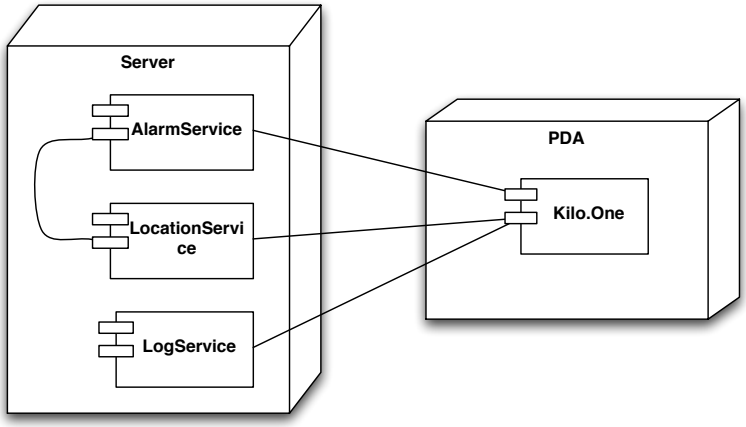

Figure 4.7: Deployment of applications and services in the Kilo.One protoype.

- global location service, and
- alarm service.

The *local location* service provides access to Bluetooth GPS devices from Windows Mobile applications. For this, we used the existing PlaceLab implementation [63] which provides translation between the GPS coordinates and locations. PlaceLab includes a library for accessing Bluetooth enabled GPS receivers from .NET applications, which has been integrated into the framework, and runs as a separate thread in the application. The *global location* service allows for sharing location information between users. The *alarm* service collects alarms from the preexisting alarm systems, and allow the PDAs to retrieve them.

The global location service is a simple web service, its interface is shown in listing 4.1. The service has two operations; one for updating the current location of a user, and one for retrieving the location of a user. The service itself merely keeps a time stamped list of current location.

The alarm service stores previous alarms and allows for retrieving them. The existing alarm system allows for three methods for interaction:

Listing 4.1: The interface of the **Kilo.One** global location service in pseudo code.

```
interface GlobalLocation {
    void myLocation(Use user,Location currentLocation);
    (Location,TimeStamp) getLocation(User user);
}
```

Listing 4.2: The interface of the **Kilo.One** alarm service in pseudo code.

```
interface AlarmService {
    Alarm[] getAlarms(Location loc,TimeStamp from,
       TimeStamp to);
    Alarm[] getAlarms(Location loc);
    Alarm[] getAlarms(TimeStamp from, TimeStamp to);
}
```

- Text messages,
- email, and
- a web interface,

where text and email allows for receiving alarms, and the web interface allows for configuration and browsing previous alarms.

To integrate it with the prototype, we use the email interface to the system by setting up a separate email account for storing alarms. The alarm service then uses IMAP for accessing the alarms. This has the dual advantage, that alarms will continue to be stored at the mail server while additional information can be added to their headers. This is for instance used to register if alarms have been signed off. Listing 4.2 shows the interface of the alarm service as pseudo code. The three methods retrieve alarms from a specific location, or all alarms, possibly bounded by time stamps.

The framework for building applications is component and connector based, with a collection of pre-built components and connectors available as a starting point. It is a simple implementation, intended for the rapid creation of a prototype application to test the basic ideas and design. Besides standard implementations of a component and a connector, and a hierarchy of connectors needed for the purpose of building the application prototype, it consists of a simple context-model, that directly supports the concepts needed in the application using objects for representing context. During the development of the prototype, we experimented with a series of different connectors and their interaction, so the connectors included in the framework vary from simple **MessageRole** and **MulticastRole**, which implements asynchronous message passing for unicast and multicast respectively, to and **IMAPMailConsumerRole** which implements IMAP functionality. Some connectors build on lower-level connectors, but this is mainly to investigate techniques for

reusing connectors to build more advanced connectors. Most connectors actually used in the final prototype where specifically created for the purpose of the prototype. As for context, the model is implemented with an object-oriented approach, with a class for each type of context in the system.

4.2 Kilo.Two

The second prototype, aptly named **Kilo.Two**, was initially designed to investigate the use of context-awareness in plant production. Later, the focus was shifted slightly to increase work on middleware issues, with a smaller focus on the application prototype. Hence, the application prototype is not complete.

There are three main differences from the first prototype caused by the move from pig- to plant production:

- A static infrastructure cannot be assumed,
- workers carry out a single task at any given time, and
- there are existing systems which the prototype should be able to integrate with.

The lack of static infrastructure means that we cannot rely on "always-on" communication, prompting the need of a new communication model. The second implies that we can again look at specific tasks, and how to support them. The reason for the first difference is, that fields are often spread out over a large area, not necessarily even in the vicinity of the farm. This can be alleviated using data transmission provided by cellular networks, however, not all land is covered by cellular networks. In the case of Denmark, about 95% of the country is covered by cellular networks. The remaining 5% are typically areas with no or few residents, and is typically used for farm land[1]. That workers typically only carries out a single task is due to the nature of the work. Typical tasks in plant production could be to sow crops on a field or spray it with pesticides. Task like these are carried out with an agricultural worker sitting in a tractor, with some kind of machinery attached to it, thus only allowing for one task to be carried out at a time. The existing systems are both back-end systems in the form of a central database that stores information about plant production, and systems local to the farm or equipment. For instance, some pesticide sprayers allows for retrieving information about current state, such as the current flow in each nozzle.

[1] In many ways, Denmark is not representative for most countries in this area. Denmark is a very small and densely populated country, and all of the land area has, at some time or another, been farm land. Currently, some land has been allowed to be reclaimed for nature, but most is still farm land.

4.2.1 APPLICATION PROTOTYPE

The goal of the application prototype was to extend the first prototype with support for carrying out tasks in the field. Contrary to the pig scenario, existing PDA software for ordering and registering tasks not only exists, but are in some use (e.g. Lommebedriften Online ; [36]). Another difference from pig production is that most data is stored in a central, nationwide database. The reason for this is simply that most treatments of a field have to be registered with authorities, and the central database allows for retrieving the information when needed. However, the database also allows for storing additional information, such as upcoming tasks, and whether they have been carried out. Thus, we can assume and existing list of tasks, and only need to determine which one is being carried out.

Existing solutions typically provide a list of all fields belonging to a single farm, allowing the worker to tick of tasks as they are carried out, and administration to review information about each field. The solutions overcome the lack of communication by being off-line solutions, where data can be synchronised on request, either by wireless network, or by using a USB cradle. This suffers from lack of real-time information, which can be used for planning, and also from user interface problems. In the case of the user interface, consider that fields are typically identified by an id. A typical farm in Denmark today has upwards of fifty fields, meaning that to find a particular field in the PDA program requires the user to remember the id of the field. This becomes even more difficult, if the field is divided into plots, as is typical for nurseries.

To improve the existing solutions, we intended to extend the **Kilo.One** prototype with support for plant production. Initial field studies provided the following central observations:

- There are preexisting lists of fields in digitised form,
- information about tasks for each field are preexisting,
- only one task is carried out at any time,
- tasks are performed using certain equipment,
- tasks are carried out using a tractor,
- modern tractors have built-in GPS units.

Furthermore, the information about fields is readily available in an existing, central database which provides a service interface for access, allowing for integration with our prototype. The central database also supports geospatial information about the fields. This allows for simply querying the database to determine if a coordinate belongs to a field, or retrieve the coordinates of the field from the database and determine it locally. Only one task can be carried out at any time, due to the nature of the tasks. For instance, it is only possible to harvest one field at a time for a worker driving a harvester. On the other hand, multiple workers can participate at carrying

out the same task simultaneously. Furthermore, nearly all tasks in a field are carried out using a tractor pulling some equipment. Although there are some deviations from this, such as surveying fields to determine infections of disease or weed, it makes sense for us to determine the current task of the *tractor* and not the task of the user, while maintaining the user oriented tasks of the first prototype. We do this as shown in figure 4.8. The equipment type can be determined based on information retrieved from the ISO bus in the tractor for newer equipment, but for the purpose of the prototype we require the user to select it by hand.

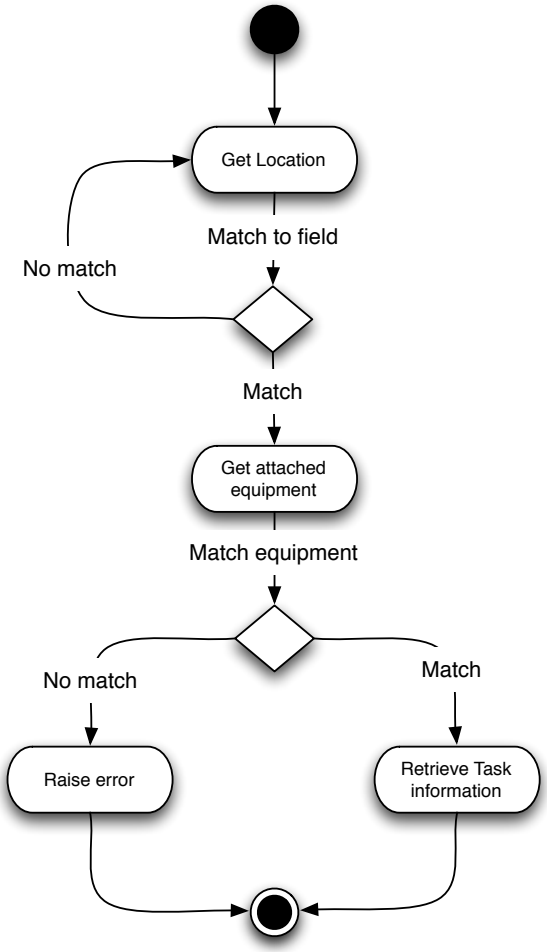

Figure 4.8: Determining current task in plant production based on available information.

As stated above, we can determine if the tractor is in a field by accessing the built-in GPS unit and available geospatial information. To determine if the current situation matches a task, we need information about the next task associated for the field, and the equipment needed to carry out the task, determine if the equipment matches the task. Figure 4.9 shows a simplified ontology for tasks. Depending on

the exact task and level of technological support, additional information may be included in the ontology. For instance, the dates where the tasks are carried out may be used to determine if the current tasks would be carried out at the correct time, or weather information can be checked to determine if it is an opportune time to carry out tasks like pesticide spraying.

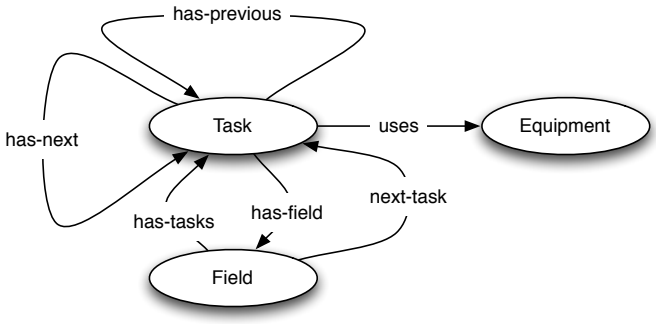

Figure 4.9: Simplified ontology of the task concept for tractors in fields.

As with the first application prototype, one of the main goals is to record information which needs to be reported. This includes the crop and the use of manure and pesticides. When the current task is know, we can provide a user interface suitable for entering only the information which cannot be determined automatically. However, an many cases it is possible to do better than that. A modern pesticide sprayer can detect the amount of liquid sprayed from each nossle. If we record this information, we can determine the exact amount of pesticide used, providing we also know the dose used in the sprayer.

For other types of work in the field, such as inspecting crops, the task can often be determined by location and the fact that *no* tractor is present. In this case, location would be determined by a GPS unit in or paired to a PDA. However, we will most likely not be able to determine the specific task, but could support a situation-like approach, limiting the number of possible tasks.

The second application prototype was never completed during the project. The needed middleware for supporting the application was designed, including a context model and an initial design of the applications was created, but it was not implemented beyond a horizontal prototype.

4.2.2 MIDDLEWARE PROTOTYPE

To obtain the data needed in the application prototype, we need middleware that supports some coordination model and a component model which allows for loading components on demand. The coordination model is needed for service discovery and integration, while the component model is needed for flexibility. We also need some representation of the data in the form of a context model. Thus, for the second version of the middleware, the responsibilities where extended to in-

clude a context model that would be general enough to potentially support the entire agriculture domain and created a dynamic, declarative component and connector model.

To support the application, we need a number of services:

- **Administration** service,
- **Context** service, and
- **Discovery** service.

The **Administration** service corresponds to the existing, central database, where information about fields are stored. Ideally, this would be integrated with other administrative services, allowing for a single, central service. While other services are not available at the time of writing, additional application specific service would include services local to a farm, for instance for storing information about equipment. For the purpose of the prototype, we have simply used the existing service, instead of creating our own.

The **Context** service replaces the **Location** service of the first version of the middleware, and extends it to provide a full context model.

The **Discovery** service provides service discovery in the local network, and allows for registering remote services, so they may accessed by local devices.

Together, these services provide for implementation of the plant production case, by using existing adminsitrative systems for storing and retrieving data about fields. The context service allows for retrieving and dissemination of contextual events. The service can be used for updating location of users and machinery, and for triggering the delivery of alarms and other notifications for the user. For subscriptions to context, the service uses a publish/subscribe type system described in section 7.1. The discovery services ties the application with services provided both locally and globally, using a semantic service matcher, as described in section 7.3.

Context Model

The context model is ontology based. Ontologies where chosen for a number of reasons:

- Extendability,
- expressibility,
- interoperability.

Extendibility refers to the fact that ontologies can be extended when needed to represent new types of context, while maintaining the previous structure. Expressibility refers to the properties of ontologies for expressing context compared to other approaches. For instance, it is easy to add meta data to an ontology. This is discussed

in more detail in section 5.2. Interoperability comes from the use of standard notation and tools for creating and interacting with a context model described using an ontology.

Ontologies can be extended by domain experts, using publicly available tools, such as Protege-OWL [77], allowing for adding new concepts to the context model for new applications. For testing purposes, we have mostly used an existing ontology by Zhang [120], but extended it for use with the prototype. This mostly involved adding concepts specific to agriculture (Figure 4.10).

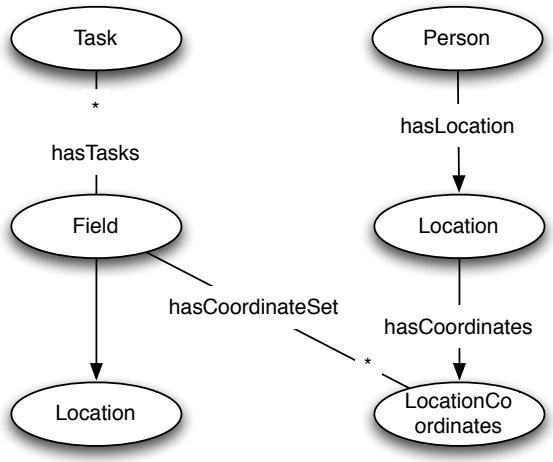

Figure 4.10: Part of the context ontology of **Kilo.Two** prototype.

Context is updated in the context service by having context sensors send instances of concepts in the context ontology to it. Similarly, consumers of context subscribe to changes in context using statements written in SQWRL [94]. SQWRL allows for a simple syntax for defining queries of ontologies. It is basically an extension of SWRL [64], with SQL like selection. Subscriptions to context are in principle the same as subscriptions to published messages for the **Messaging** service, which is described in section 7.1.

SERVICE MODEL

In the **Kilo.Two** prototype services are externally represented as OWL-S services (described in section 2.3). That is, a service is defined by

- a profile,
- a model, and
- a grounding.

The profile describes what the services does, the model describes how to use the services, and the grounding describes actual access including message format and address.

OWL-S where chosen for two reasons: It includes a grounding for web services defined with WSDL, and it uses OWL to describe relevant ontologies, meaning we can use our context ontology for describing profiles of the services. Furthermore, OWL-S allows us to match services by providing a, possible partial, service description of the service we need, and then have match it to available services using semantic matching, as described in section 7.3, and has existing tool support.

Thus, the interface for a service is a WSDLdescription of a web service. To implement this, the middleware provides support for exposing an interface as a web-service, using the built-in web service technologies of Microsoft's .NET [87] in a way that is similar to our work in using ArchJava for service oriented prototyping, described in section 6.4.

Component and Connector Model

The connector model is, in essence, the same as for **Kilo.One**. The main difference is the addition of a declarative model, which basically defines the run-time component and connector view of the application using an external description. The connector model is essentially the same as for Kilo.One, except for the addition of a loading mechanism.

While accessing different types of equipment, which may have to be handled in different ways, should be encapsulated in connectors, a dynamic component model allows for supporting dynamic loading of components in the style of component models like OSGi [114]. There is no existing OSGI-like component model for .NET, although other authors have implemented similar component models, for instance Escoffier et al. [45]. The main problem for developing this kind of model in .NET is the fact that .NET does not support unloading individual classes from an application domain, meaning that while we can dynamically load new code, doing so will lead to a bloated application domain since unused code cannot be unloaded. Is also makes it impossible to update at run-time to a new implementation of an existing name-space. Technically, an application programmer could load components into separate application domains in the same program. The advantage of this is, that while components cannot be unloaded from an application domain, it is possibly to unload the entire domain. The main drawback is that application domains can only communicate by passing messages to each other through a loopback network interface, making communication slow. The **Kilo.Two** middleware does not provides support for loading components into separate application domains, although it is certainly possible to extend it with this support. In the middleware, components are stored as dll files, which can be loaded on demand.

The component model instead assumes that components are loaded start-up, using an external description. Currently, this description can be either of two options:

- A customized XML format descriping components, connectors, and their relations or

Listing 4.3: Standard **Kilo.Two** component with a `Component` attribute defining the external description.

```
namespace Kilo.Two.Components
{
    [Component("http://localhost:9999/ontologies/kilo.owl
       #Component")]
    class Component : IComponent
    {
        ...
    }
}
```

- an ontology describing the structure of the application at run-time.

The customized XML format is a simple description of components and connectors to load, and an "architecture" which simple describes a list of connectors which should be bound to at least two components. For reference, the XSD schema is included in appendix B.1.

Using an ontology instead in principle allows for describing the same, but using OWL instead. That is, instead of being types in XSD, components, connectors, and architectures are concepts in an ontology, and relations between them are expressed as relations in the ontology. This approach is described in section 7.2. To define where these descriptions can be loaded, we employ C# attributes as illustrated in listing 4.3 with the use of a `Component`.

While both descriptions allow for describing the local structure of a program, the OWL-based description includes the possibility of describing required services as well as components. This allows for a more declarative description of the structure of the application, since required services and components can be expressed in terms of component and service types in an ontology, while the XML format requires specific information about where a particular component can be loaded. As such, they serve slightly different purposes.

Whether using the customised XML format or an OWL description, at runtime a loader retrieves the descriptions and use them to setup correct connectors and, in the case of the OWL description, locate appropriate remote services using the **Discovery** service, and then load the appropriate connectors and components.

4.3 SUMMARY

During the course of the project, we have created four related prototypes; two application prototypes and two middleware prototypes. The first middleware prototype had a main focus of supporting the first application prototype, while the second middleware prototype attempted to generalise the lessons learned from the deployment of the first prototype, while supporting the new requirements that originated

from the second application prototype.

The development of prototypes drove the project. First by gathering requirements for the prototypes with field studies, interviews and focus groups and later by being test-beds for technologies and design suggestions and the focus of evaluation.

While this chapter has described the prototypes developed during the project, the scientific results are described in part II.

Part II

Middleware for Context Awareness

The following part describes the research carried out in the project, providing descriptions of previous work where appropriate and concludes with summarising the results of the project.

chapter 5 Context Awareness describes context and context aware middleware. To facilitate the discussion, it also includes an overview and definition of middleware and describes a layered middleware model. It also provides a survey of some existing context aware middleware systems, and describes context awareness in agriculture and how it can be supported using standard semantic web technologies.

chapter 6 First Class Connectors discusses first class connectors in various forms, and relate them to notions of coordination and coordination languages. It concludes with describing how first class connectors can be utilised for prototyping service oriented architectures.

chapter 7 Semantic Modeling describes semantic web technology and how it can be used for various areas of middleware and context awareness, including architectural description, context models, and publish/subscribe.

chapter 8 Summary and Conclusion restates the research questions and hypothesises and summarises and concludes upon the results.

Chapter 5

Context Awareness

Context awareness has become an increasingly important part of computer systems. This is in part due to the increased mobility of computers and other computational devices. By context awareness, we understand the use of *context* on runtime, to provide better or more precise service to the user. As an example, consider the emerging popularity of location aware systems, currently the most common type of context used in real world applications, thanks to the increased ubiquity of cheap GPS devices. Modern PDA applications utilise the awareness of the current location to provide more valuable or easier to use services, for instance by immediately providing a list of nearby points of interest, without requiring the user to enter the address of his current location. Another example of a context aware application is car navigation systems, which also relies on knowledge of the current location. Systems using other kinds of context are less common, but a number of applications using *proximity* are available, although this "proximity-awareness" might sometimes be as simple as defining two devices as being in proximity of each other because they are within the Bluetooth range of approximately 10 meters.

By context, we understand information about the environment in which the computer application or system is executing:

> Context *is any information about the environment, whether the computing environment or the physical environment, of a computer application, system, or person which can be used at runtime to determine the behaviour of the application or system.*

By Contrast, the following definition is due to Dey [40]:

> Context *is any information that can be used to characterise the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.*

While the two definitions may seem similar, Dey refers to the *situation* of an entity. However, this clashed with a definition of *situation awareness*, due to Meissen et

al. who, by using the definition of context due to Dey, defines situation awareness as context related to an individual, and defines as hierarchy of situations [86] for these individuals. Consequently, we prefer to separate context- and situation awareness as two distinct types of "aware" systems. While context awareness includes any kind of information relevant to a system, situation awareness is only concerned with the context of individuals, and includes a hierarchy of situations with increasing precision, as illustrated in figure 5.1. Furthermore, our definition of context does not mandate that context information is related to a specific entity. Also, the definition by Dey focuses on interaction between the user and the application, thus not including applications that do not interact with any user, but may still nonetheless benefit from context awareness. Thus, our definition is more general than the one by Dey. As an example of context which is covered by our definition, but not by Dey's, consider the current Unix time. Since the current Unix time is part of the environment, it is context. However, it is not related to a specific entity[1].

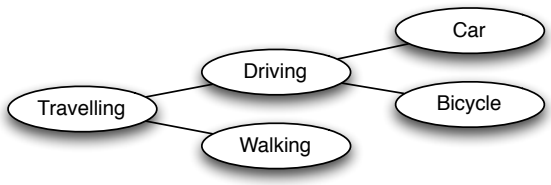

Figure 5.1: Sample hierarchy of traveling situations.

To further characterise context, we divide it into different types:

- Internal,
- external, and
- profiles.

Internal context refers to information which is internal to the device or computing system the application is executing on. This could include processing power, battery life, available services and so on. *External context* refers to information outside devices, for instance the current location of a device or person, the current situation of the user or other users and devices in the vicinity. A *profile* is information related to a specific entity. In the case of a user, this might include a users name, address, and preferences. In the case of an application, it might include requirements for the execution environment of the application, such as minimum processing power or minimum network bandwidth.

[1] Purists might argue that the current Unix time is related to a single hardware entity, since exact synchronisation is impossible.

5.1 Context

From Weiser's and his colleagues' early prototypes of ubiquitous systems, context has played an important role. In their system, the only real context was location, but this has proven to also be the most widely used.

Schilit et al. describe a model for what they term *mobile distributed computing* [103], where context plays an important role. In their model, users interact with embedded, mobile, and stationary computers during the course of a day, meaning that computing will not take place in a single context. In an earlier paper, Schilit and Theimer introduced the term *context-aware* and *context-aware systems* as systems which adapt to changes in location and the collection of nearby entities, whether they are people, devices, or other resources.

Schilit et al. do not present a formal definition of context, but restrict themselves to stating that context has at least three important aspects [104]:

- Where you are,
- who you are with, and
- what resources are nearby.

Resources could be things such as connectivity bandwidth and cost, but other information like lighting- and noise-level might also be of interest. That is, they provide a prototypical definition of context.

The work is based on the ParcTab mobile device [117], which is basically a small graphical terminal which communicates through an infrared network. The location of a user is then simple to identify, since infrared receivers for the network do not span multiple rooms. While this relatively simple device implies relatively simple experiments, it nevertheless allows for experiments with *proximity selection*, for instance changing the focus of the user interface to display nearby devices or services, allowing the user to make a faster selection. Other experiments are *contextual reconfiguration* and *contextual information and commands*. Reconfiguration is the process of adapting the mobile distributed computing system to the changing context by adding or removing components, while contextual information and commands refers to helping the user make informed choices or even act on behalf of the user depending on context. The classical example is having a *print* command print to the nearest printer based on location information. Finally, systems may trigger actions based on context. This is similar to contextual commands, but happens automatically.

While the prototypical approach described above to defining context has its merit, and some philosophical underpinning starting with Wittgenstein [119], it has some limitations in our opinion. While they provide an intuitive understanding, they do not necessarily lead to stringent research methodology. More formal definitions based on the characteristics of the concepts provide concise definitions allowing for affirming that an observed phenomenon belongs to a specific concept. This is approach derived from the philosophy of Plato's world of *ideas* [95].

Dey and Abowd presented an early survey of context-aware computing, and present definitions and categories of context and context-awareness [41]. The expressed goal is an *operational* definition of context, and they observe that the most operational definitions all include as part of the environment:

- The computing environment,
- the user environment, and
- the physical environment.

However, they also observe that these enumerations of context are too specific since, depending on the application, some context may be important, while other context may be completely immaterial.

Besides the previously mentioned definition of context due to Dey, he also provide a definition of context awareness [40]:

> *A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task.*

In this definition too, interactions between the application and the user are in focus. We believe that this limits the scope of context-aware applications unnecessarily. This is described in more detail in chapter 5.

Dey also suggests a *situation* abstraction, as a collection of state of devices and aggregators, as an addition to the concept of context, this differs slightly from our use of the term.

Zimmermann et al. observes that the two methods introduced so far for defining context both have flaws [121]. They argue that the prototypical approach suffers from incompleteness of the definition, while the classification approach suffers from generality, and often includes self-referencing definitions, by using terms like *context*, *situation*, and *environment* with similar meaning.

Instead, they suggest a formal and operational definition, which can be used to communicate with users during development. The definition has three parts:

- A general definition,
- a formal description of the appearance of context, and
- an operational definition of the use and dynamic behaviour of context.

To come up with a definition that is general, Zimmermann et al. assume the above definition by Dey, but extends it by defining five categories in which context may belong:

- Individuality,
- activity,

- location,
- time, and
- relations,

where the activity defines the relevant context, and time and location drives the creation of relations. The idea is to order context by maintaining that any information about an entity belongs to one of the above categories. *Individuality* describes the entity itself, *activity* contains all tasks the entity might be involved in, *location* and *time* contains spatial-temporal information, and relations contains information about any possible relation the entity can establish with other entities.

The individuality context can further be refined into context groups depending on four entity types:

- Natural,
- human,
- artificial, and
- group.

Natural entities occur naturally. This includes plants and rocks. The group also includes information about interaction with these entities. The *human* group includes information related to humans. This could include preferences and personal favourites needed by applications to make decisions on behalf of the user. The *artificial* group includes phenomena that exist because of human actions or technical processes. It includes anything created by humans, such as buildings, computing hardware, and sensors. The *group* group is used for representing the structure of groups of entities and their common properties. That is, the entire individuality context describes what Dey defined as entities in his definition of context.

The activity context represents current and future tasks and goals. This context is highly domain dependent, so we will not go into further detail.

The location context represents locations, whether physical or virtual, as well as information like speed and orientation. Physical location refers to the placement of entities in the physical world, either defined in a quantitative or qualitative way. Quantitative location information is defined in terms of coordinates, while qualitative information is defined in terms of symbolic names and their relations, for instance the layout of rooms in a building. Quantitative and qualitative location information mostly serve two distinct purposes. While qualitative information is easy to understand for humans, computers in general need quantitative information for most purposes.

The time context basically represents anything to do with time, such as time stamps, periods, and more general terms, such as workday and weekend. It is generally used to order context and events. It may also include process-oriented concepts, like work flows.

The relation context captures relations established by an entity with other entities. Relations themselves may be determined by either temporal or spatial properties, like proximity, but the set of relations make up the relation context. The relation context is further divided into three relation contexts:

- Social relations,
- functional relations, and
- compositional relations.

Social relations include any social aspect of the entity such as friend, co-worker, and relative. Importantly, an entity always has a *role* in the relationship, since relationships differ in intimacy and sharing. A *functional* relationship indicates that one entity uses another entity for some reason. For instance, a human could *use* a computer or *give* a presentation. *Compositional* relations describe the relationship between the whole and their parts. This includes the common notions of *association* and *aggregation*. A visual overview of this taxonomy of context categories is provided in figure 5.2.

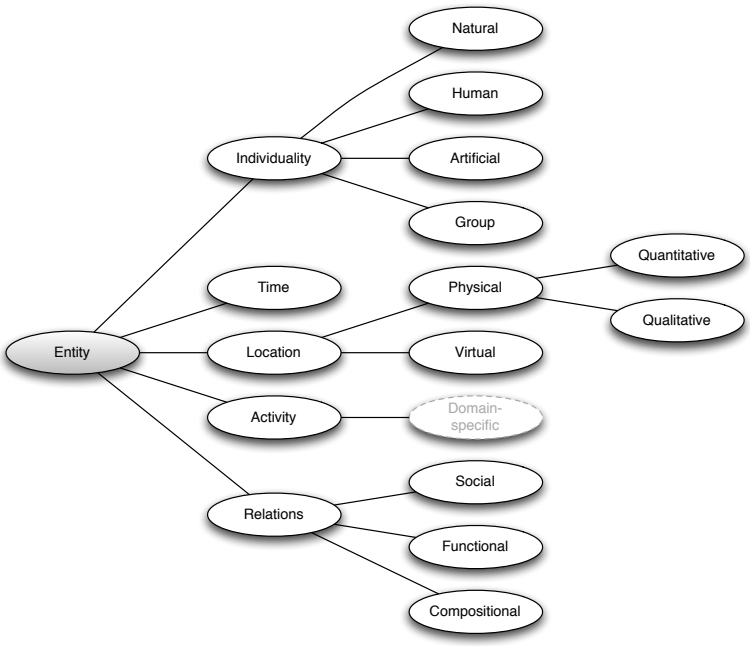

Figure 5.2: Context categories, as defined by Zimmermann et al. [121].

Another approach to organising context is to look at different levels of context. An early attempt at distinguishing between "raw" context and concepts at a higher level was done by Hull et al. [66]. In their work, they assume a scenario with wearable computers, thus the focus is on the immediate situation surrounding the wearer. However, other context also becomes important such as power consumption and

the capacity to transfer data. However, the most interesting aspect, which the authors only briefly touch, is the distinction between context- and situation awareness. Their use of the term seems to match our use, as described previously.

Another look at situations by Meissen et al. is concerned with modelling situations as being a collection of context variables [86]. They take offset in a definition due to McCarthy and Hayes [81], which can be paraphrased as:

> *A situation is a presentation of the current state of the world.*

The context variables in their model are all simple values, which can be measured and do not require any reasoning. They distinguish context and situation by noting that context, as defined as the collection of all context variables, may change slightly over time even though the situation does not change. Instead, situations are defined by mapping context variables to *characteristic features* of a situation. For instance, the context variable `velocity = 2 km/h` may be mapped to the characteristic `kindOfMovementSlow`. Thus, a domain- or application specific mapping between context variables and characteristic features of situations must be defined. For generalisation and specialisation purposes they furthermore define graphs of relationships between characteristics for use in inference rules. I.e. a meeting taking place in a certain conference room also takes place at a certain address. Similarly, for situations there is always a former, current, and upcoming situation, where the upcoming situation is an expected situation, for instance based on the calendar of the user.

This definition of situation allows for creating chains of situations for a single user, and defining operations on characteristics and situations. For characteristics, they define the following operations:

- $\texttt{generalise}(\texttt{cs}_1, \texttt{cs}_2) \rightarrow \texttt{cs}_\texttt{r}$,
- $\texttt{fulfills}(\texttt{cs}, \texttt{p}) \rightarrow \texttt{True}, \texttt{False}$, and
- $\texttt{compare}(\texttt{cs}_1, \texttt{cs}_2) \rightarrow [0, 1] \subset \mathbb{R}$.

$\texttt{generalise}(\texttt{cs}_1, \texttt{cs}_2)$ finds the first common generalisation $\texttt{cs}_\texttt{r}$ of $\texttt{cs}_1$ and $\texttt{cs}_2$ in the hierarchy of characteristics. $\texttt{fulfills}(\texttt{cs}, \texttt{p})$ determines whether a set of characteristics fulfill the requirements of a situation pattern p. Finally, $\texttt{compare}(\texttt{cs}_1, \texttt{cs}_2)$ calculates a comparison based on subsumes-paths in the graph.

On situations, they define the operations:

- $\texttt{previous}(\texttt{s}) \rightarrow \texttt{s}_\texttt{p}$,
- $\texttt{next}(\texttt{s}) \rightarrow \texttt{s}_\texttt{s}$, and
- $\texttt{combine}(\texttt{seq}) \rightarrow \texttt{s}_\texttt{r}$.

$\texttt{previous}(\texttt{s})$ and $\texttt{next}(\texttt{s})$ determine the predecessor and successor of a situation respectively, while $\texttt{combine}(\texttt{seq})$ finds the a generalisation of the situations in seq according to the graph of situations.

5.1.1 USING CONTEXT

One of the first examples of context aware applications are the applications described by Weiser [118], which utilise location information for automatic re-routing of telephone calls or locating users. This is an example of *proactive* middleware , in that it makes decisions on behalf of the users based on context information. In contrast, *reactive* middleware does not make decisions on behalf of the user, but may change behaviour in its interaction with the user, based on available context. As a simple example, consider a map application on a mobile device with a GPS unit. When no location information is available, it might open with the map centred on the home of the owner. However, if it has a more up-to-date information about the current location of the user, it can instead open the map centred on this location. This is actual common behaviour for many map applications. Furthermore, it might decide how much of the map to show, for instance based upon knowledge of the precision of the location information or the current speed. If the map is used for navigation in a car,

Barkhuus and Dey have examined the reaction of users to different levels of interactivity with context-aware applications [10]. They define three levels of interactivity:

- Personalisation,
- passive context awareness, and
- active context awareness.

For personalisation, there is no context-awareness taking place. Instead, individual users tailor or customise the applications they use for their needs, perhaps depending on the current situation, but always by conscious choice. For mobile computing, devices are typically personal, and therefore potentially open to the widest array of personalisation options, since they will not affect other users. For passive and active context awareness, the authors use as an example a mobile phone which updates the time zone based on information received from the cellular network. In their definition, an active context aware application would do so automatically, while a passive application would present the user with a choice of updating the time-zone. At first, this might seem similar to our definition of proactive and reactive context aware systems. However, there is a profound difference. Reactive systems changes their interaction with the user, based on available information, whereas proactive systems make choices on behalf of the user, so there are levels of interactivity, and levels of *control* of the user, as illustrated in figure 5.3. This gives rise to four different types of context aware systems:

1. Active interactivity and reactive context awareness.
2. Active interactivity and proactive context awareness.
3. Passive interactivity and reactive context awareness.

4. Passive interactivity and proactive context awareness.

If we consider the above mentioned examples, the map application is an example of a reactive, active context aware system. It is reactive since it does not decide to show the map, the user does, but it is an active application if it does not ask the user if he wants to centre the map on the current location. For the mobile phone example, a device that automatically sets the time-zone is proactive, since the user does not ask to get the current time, but it may be either passive or active as described above.

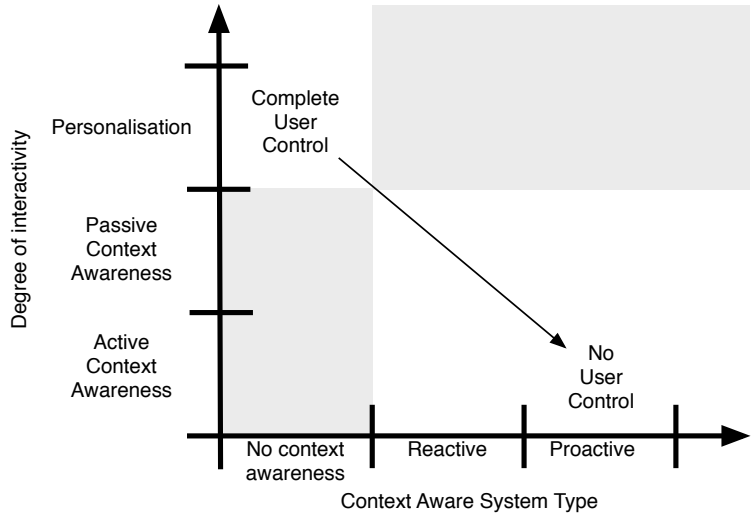

Figure 5.3: Levels of interactivity are orthogonal to the type of context aware system. With decreasing interactivity and increase in proactive choices, the user gradually looses control.

Baarkhus and Dey note that there is a general agreement that different levels of interactivity exist, but researchers differ in where to separate them, with definitions similar to their passive and active systems, and our reactive and proactive systems. Combining these two approaches in describing interactivity and control of the user provides a better understanding of the amount of control left for the user, but we note that our resulting four categories do not have clearly defined boundaries. Baarkhus and Dey further concludes that users seem willing to loose some control but only as long as the usefulness of the application is greater than the cost of limited control, supporting our position of using participatory design for context aware systems. Cherverst et al. describe pull and push levels of interactivity [31], where the push system would fall into the category passive and proactive, and the pull systems is somewhere between personalisation and passive and is reactive, in that the user actively requests information.

5.2 CONTEXT MODELS

An important decision in systems for context awareness is the design of the context model. For simple systems, like the location aware systems currently becoming mainstream, a simple, static model might be sufficient. If you know that the only context supported will be location, simply implementing a location API will normally suffice.

In more general context-aware systems, we might have an idea about the kind of context which should be supported and, equally important, what kind of operations we want to be able to do on the context information. Typical operations include

- storing and updating context,
- retrieving context, and
- aggregating context.

While not all systems store context for later retrieval, updates must at least be propagated to interested parties, so in some cases, updating and retrieving is more like a single push-operation, similar to publish/subscribe. In both cases, clients must be able to express what context they wish to retrieve or obtain notifications about. For context aggregation, it must be possible to combine the low-level, sensed context into meaningful higher order context. While most context models support aggregation, they differ in how well existing tools support this. This ranges from having to program specific aggregation code for an application, to using off-the-shelf tools to perform it, although how to perform the aggregation, i.e. what should be combined to create higher-order context, will have to be defined in all cases.

The most common context models are

- key/value pairs,
- markup-based,
- object-oriented,
- logic-based, or
- ontology-based.

Key/Value pairs is one of the oldest, and arguably most widely used models. Schilit used key/value pairs to model context by allowing programs to access variables in the environment [105]. Key/value pairs are easy to implement and use and, if combined with some form of structure for grouping keys belonging to the same entity, can easily be used to describe e.g. profiles of entities. The main drawback of key/-value pairs is the lack of advanced structure, making retrieving and manipulating context more difficult.

Markup based approaches typically employ an XML-based model, allowing for semi-structured context-data. Depending on the exact model, the markup language

can be more or less restrictive. Approaches range from static hierarchies for expressing profiles to dynamic hierarchies where the interpretation is left to the application.

Object-oriented context modelling uses the techniques known from object-orientded languages to define a model. It does not necessarily have to be build using class-hierarchies directly, for instance UML can be used to define the model and operations on context can then be operations on the UML model.

Logic-based models are similar to key/value pairs, but use first-order logic for expressing higher-order or derived context. As such, sensed context is expressed as facts, while other context is expressed by means of rules and expressions. This gives a context-model with a high degree of formality. It also means that context will typically be stored, since the logic-engine will maintain a model of the current context for evaluations. Adding, updating or removing context then amounts to performing the same operations on facts, while deriving context amounts to inferring it using rules.

Ontology-based models are in some ways similar to markup based approaches, in that they offer a semi-structured context model, but they can also infer higher-order context using rules. In an ontology based model, context is instances in an ontology. Updating context is then done by operations on instances, adding, changing, or removing them. Like in logic-based models, it is possible to add rules and expressions to the ontology, which infer higher-order context by creating new instances of concepts.

Strang and Popien provide a suvey of the different context models [111], and provide a set of requirements that the models should ideally have:

1. distributed composition,
2. partial validation,
3. richness and quality of information,
4. incompleteness and ambiguity,
5. level of formality, and
6. applicability to existing environments.

Distributed composition is a requirements that originates with the use of context awareness in ubiquitous computing. Since devices are distributed, so is the gathering and use of context will be distributed. While not all context aware systems will actually need this, the context model should preferably support it. *Partial validation* originates in the previous requirement. If the gathering and use of context is distributed, complete knowledge cannot be assumed for any given device, it should be possible to validate part of a structure. Preferably, it should be possible to support indication of *richness and quality of information*, since richness and quality of information varies over time for a single sensor and between sensors. Since different sensors may provide conflicting information and all information is not necessarily available at all times, the model should support *incompleteness and ambiguity*

of information. *Level of formality* covers the need for providing interpretations of terms used in describing context. That is, it should allow for meta-level descriptions of the terms used. Finally, *applicability to existing environments* means that it should be suitable for whatever infrastructure already exists for ubiquitous computing. The authors examine the different models used for context and observe that they fulfill different subsets of the requirements. For instance, the only real benefit of key/value pairs is that they fit into existing environments, since they mainly rely on tools available in all programming languages to interpret. Markup based models provides partial validation since they are defined be schemas, but suffer for describing for instance ambiguity of information. The main drawback of object-oriented models is the lack of formality, since information is encapsulated. On the other hand, they provide easy distribution, allows each object to validate itself and can be integrated directly into existing object-oriented systems, although they may be heavy on resources. For logic-based models, the level of formality is extremely high and values can be distributed, but they are not suited for describing incompleteness, ambiguity, or quality of information. Furthermore, adding the model to an existing environment requires addition of tools for processing the logic. Finally, they observe that the only real drawback of ontology-based model is that adding the model to an existing requirement requires tools for manipulating and querying the ontology, that they are especially good for partial validation and that quality of information and ambiguity can be added as data in the ontology.

To actually use context, applications need a method for accessing it. While the application may collect the context directly, conceptually, context is collected by *sensors*. From a software perspective, a sensor is a module which collects information. If the sensor collects external context it might also refer to a specific hardware device, such as a thermometer for temperature. In the simplest version, an application contains sensors or access them directly, to collect context information which it may then use. However, context collected directly by sensors from the environment is usually very simple, and too simple to use directly in most cases. Instead of relying on the application to make use of this simple context, *aggregators* are modules which collect context information from sensors and aggregates it into *higher level context*. As an example, a simple aggregator may use information from the calendar of a user and the current time to deduce which task a user is, or at least should, currently be working on. Notice, that since an aggregator provides context information, in this case a task, it is conceptually also a kind of sensor. Aggregation may be more or less advanced. When referring to aggregation with extensive computations, we will often refer to it as *reasoning about* context. Figure 5.4 shows an ontology of the concepts in the context aware system.

5.3 The Role of Middleware

For distributed systems using context awareness, middleware should provide common services for all clients using context. These services include, but are not neces-

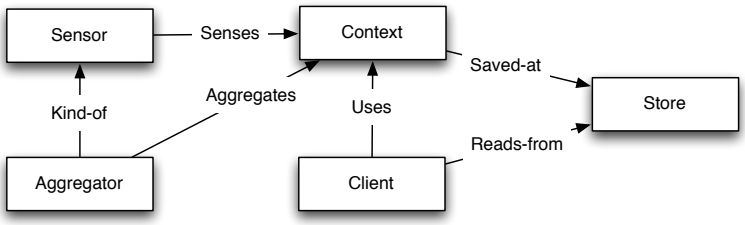

Figure 5.4: Ontology of Entities in a Context Aware System.

sarily limited to:

- Gathering context,
- aggregation and reasoning, and
- sharing context.

Other services which are common, but not necessarily available in all context aware systems, are context history and context-aware storage. Other systems, like JCAP [9] also focus on security of context information, allowing for using it in settings where context data might be confidential.

For gathering context, the middleware should control sensors in the environment by discovering sensors and setting up aggregators to provide higher level context information. The middleware should also provide an API for letting clients access context based on their needs. It is then the responsibility of the middleware to gather the context from appropriate sensors, and possibly set appropriate reasoning rules.

Sharing of context can basically be either a push or a pull model. In the push model, a client registers interest in certain context with the middleware, and whenever updated context information becomes available, the middleware pushes the information to the application. In the pull model, the application retrieves context information from the middleware whenever it is needed.

These two models are related to two distinct types of context aware middleware:

- Proactive, and
- reactive,

as discussed in section 5.1. *Proactive* middleware supports applications where events in the form of changes in context results in actions by the application. In this case, the push model is the most appropriate, since it allows for applications to use an asynchronous model to create reactions to changes. *Reactive* middleware supports applications which alter their behaviour based on current context. In this case, a pull model is often the most appropriate, since applications will only have to lookup context when they have a use for it.

Several middleware systems for context awareness have been developed, as detailed in section 5.4. If we focus not on the individual systems, but there intended use, we can make a number of observations:

- Context aware middleware is often intended for a specific domain.
- The middleware is often created for use in specific locations, where some infrastructure can be assumed.
- The middleware is often designed using an underlying technology which may or may not be suitable in the domain.

That context aware middleware is often created with a specific domain , or a number of related domains, in mind is not surprising. If we look at the layers of middleware, as defined by Schmidt [106], which is described in section 2.1, context aware middleware will often concentrate on common middleware services and domain-specific services. The reason for this is, that what types of context should be supported highly depends on the domain, making services that uses this context an important part of the middleware.

5.4 Context Aware Middleware

The work in this section is based on and extends work previously published [73].

As described in section 2.1, we take offset in a rather informal definition of what constitutes context aware middleware, but notes that it is in general an abstract layer between the operating system and the applications in a distributed system (figure 5.5).

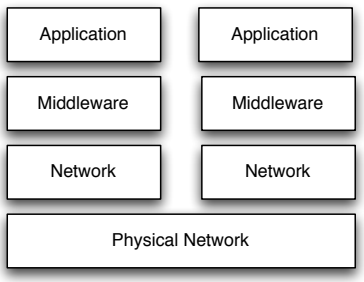

Figure 5.5: Middleware provides abstractions between applications and the network stack in the operating system.

Thus, the role of middleware may vary, but it is intended for a specific class of applications, which may contain a smaller or larger number of possible applications. Even though middleware may claim to be suitable for general distributed systems, underlying assumptions often imply a more narrow class of systems or systems for use in a specific domain or environment.

For middleware for context awareness, the intended class is obviously limited to applications that somehow use context, but they may still be further limited by additional assumptions. Other limitations may be due to the choice in context model. Different context models have different expressibility, and some systems might even use some form of static context model, limiting the application domain further. For instance, some systems only support location information.

5.4.1 Categorising Middleware

To categorise context aware middleware, we look at a number of properties discovered by surveying the middleware. To ensure that the properties where actually founded in the systems surveyed, we used the *Grounded Theory* method which is a generally accepted scientific method for qualitative exploration of real world phenomenas [55]. In grounded theory, observed phenomena are observed, grouped and then "coded", giving them names. In this way, we come to the following top-level properties:

- Environment,
- storage,
- reflection,
- quality,
- composition,
- migration, and
- adaptation.

The resulting taxonomy of properties is shown in figure 5.6.

Middleware systems often make explicitly stated or implicit assumptions about the intended environment of deployment. Middleware systems might be intended for environments with a static infrastructure to offer common services to the middleware or to applications. We say that these have an *infrastructure* environment. Other systems might only assume that devices have a method of communication between devices in the distributed system to implement coordination abstractions. Such systems are *self-contained*, in the sense that they do not rely on resources outside of the devices that the end-user applications are deployed on.

Some systems provide either storage for *context* or utilise context to store data, allowing it to be retrieved based on context parameters. For instance, some systems provide a file systems where data is stored according to context parameters as opposed to a hierarchical directory structure. In this way, applications can store and retrieve files based on their current context. Stores of context allows for applications to retrieve context from the store instead of retrieving it directly from the source, and might also provide context history.

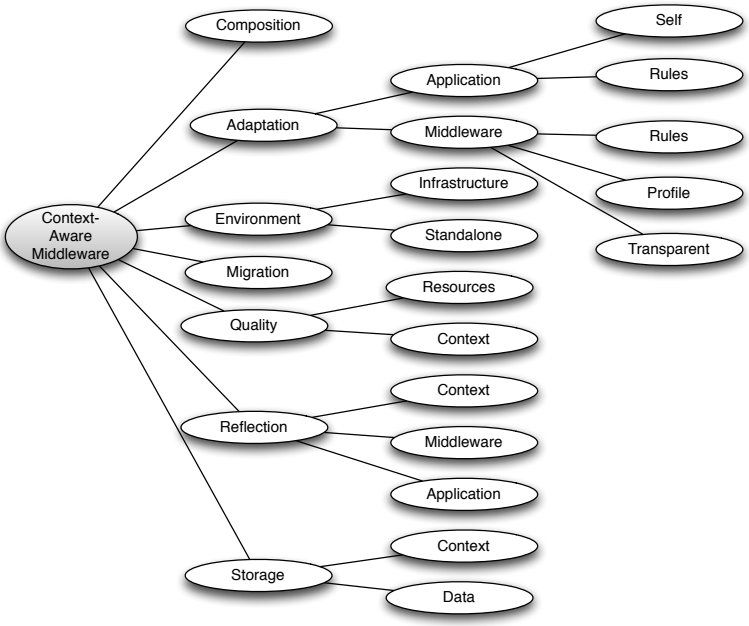

Figure 5.6: Taxonomy of Context-Aware Middleware concepts.

Some systems include reflective mechanisms, allowing applications to access meta data describing either

- the application itself,
- the middleware, or
- context information.

The meta data can allow for inspecting the information, or might allow for changing behaviour of various parts of the system by manipulating the meta data. For the application, this might alter its behaviour. For middleware it could alter behaviour, but also alter descriptions of the service the application expects from the middleware. Representing entities as meta data is sometimes referred to as *reification*, while the process of altering entities or their behaviour is referred to as *absorption* [33].

For our purpose, quality is a measure of how well a service can be performed, or how correct data is. For context aware middleware, quality usually refer to quality of service, in that applications might change their quality of service requirements depending on changes in context. Other systems provide quality measures on the context offered. One obvious quality measure for context is its *age*, but other measures have been used.

Some systems adapt to changes in context by offering context based component composition. For instance, entities might be automatically composed with other en-

tities in their vicinity by dynamically loading the appropriate components, or composition might be changed due to context events, for instance changing network access by loading another component.

Migration based on context is used in some systems to provide seamless transfer of applications supporting specific tasks to the current location of the user, or to transfer applications to access resources that are not available at the device they are currently residing on. The migration might be automatically decided by the middleware, or it might support the application in making the decision. This is somewhat related to adaptation.

Context aware systems typically provide different ways for adapting to changes in context, with different levels of support from the middleware. Context may be used by either the *middleware*, the *applications*, or both to adapt to the changes. While the applications may use any method to adapt to the middleware, we have observed three variations of adaptations performed on the middleware level:

- Transparent,
- profile based, and
- rule based.

In transparent systems, the middleware reacts to changes in context without notifying the application, often in a way that is not even observable by the application. In profile based systems, the application provides a profile describing its needs, typically in the form of a required quality of service. The middleware then uses the profiles of all applications to provide a service as close as possible to the desired service. This is mostly relevant to middleware systems that deal with internal context. Rule based systems typically allow the application or the user to provide rules of the form $a \rightarrow b$, where a is a condition and b is a consequence, instructing the middleware to perform b when a is fulfilled. The consequence can either be an action that the middleware *must* perform, or an action it *should* perform. A variation of rules is the use of Event-Condition-Action (ECA) rules, which is basically another notation that is particularly useful in event driven systems, since rules are triggered only when a specific event occurs, meaning that the condition only has to be checked on matching events. Some middleware further provides help for the application developer in providing call-backs for specific changes in context.

Other aspects of context aware middleware are also important for understanding different systems. These include the basic entity of the system and the underlying operating system. By "basic entity", we refer to the entity that constitutes an application in the system. Some systems support general purpose applications to be programmed using them, while others group existing applications into for instance a task entity, which is then the unit manipulated. Other systems assume specific programming paradigms to be used, allowing manipulation of entities that are components of the application. Some systems only run on specific operating systems, while others run part of the system on servers and part of the system on mobile

devices like mobile phones or PDAs. In other cases, some services are provided by web applications.

Furthermore, for the systems surveyed we also discuss other differences in the systems. For instance, some are pure middleware, while others are more complete systems for providing services in e.g. an office setting.

5.4.2 MIDDLEWARE SYSTEMS

Context-Aware Middleware systems have very different characteristics. To better understand systems, we have applied the taxonomy introduced in section 5.4.1 to a number of middleware systems in order to show its usefulness.

Table 5.1 provides an overview of the different systems, and shows which parts of the taxonomy each of them support. The rest of this section describes each of these systems.

Table 5.1: Categorisation of Context-Aware Middleware Systems.

Middleware	Environment		Storage		Reflection			Quality		Composition	Migration	Adaptation				
												Middleware			Application	
	Infrastructure	Self-contained	Context	Data	Context	Middleware	Application	Resources	Context			Transparent	Profile	Rules	Rules	Self
Aura	✓		✓	✓				✓		✓	✓	✓				
CARMEN	✓		✓							✓	✓		✓			
CARISMA		✓				✓		✓					✓			
Cooltown	✓		✓							✓						✓
CORTEX		✓	✓				✓	✓						✓		
Gaia	✓		✓	✓				✓		✓	✓					✓
Kilo	✓		✓	✓											✓	
MiddleWhere	✓		✓						✓							✓
MobiPADS		✓			✓	✓	✓			✓	✓		✓			✓
SOCAM	✓		✓												✓	

AURA

The basic entity in Aura is *tasks* [52, 61, 69, 110]. A task is an abstract description of the work currently undertaken by a user. The principle behind the system is the *Personal Aura*. Aura provides services which manage tasks, applications, and context. Contrary to most other middleware systems, applications are not necessarily built on top of the middleware, but are existing applications which can *provide* a service the task needs. As an example, consider a task involving editing a text and creating a presentation. In a Microsoft Windows environment, this task can be accomplished by using Microsoft Word and Microsoft Power Point as providers of the services *edit document* and *edit presentation* respectively. Tasks can move between Aura environ-

ments, so in a Unix environment, the service providers might be emacs and Open Office instead.

As such, Aura is intended to support everyday work in a context aware manner, by integrating applications on desktop PCs.

Tasks are controlled by a *Task Manager*, which handles migration of tasks, while services are provided by an *Environment Manager*. Since the tasks are merely abstract representations of a collection of services comprising the task, they can easily be moved to any environment which can provide the services they need.

Aura provides a *Context Observer* to manage context. Context information is used to derive the *intent* of the user. The context observer merely collects context, and reports changes to the Task - and Environment Manager. Depending on the detail of the collected context, the Context Observer might be able to derive the current task, location, and intent of the user. The current task is used for proactively loading the current task, while location is used to migrate tasks e.g. from the home to the office of the user. The intent is used for both tasks and location. If the user is working at home, but have a meeting scheduled at 10am and leaves the home computer, the Context Observer might derive that the user is leaving for the office, and migrate the current task without intervention from the user. If the Context Observer is unable to derive the location or intention of the user, e.g. because of insufficient sensors in the environment, the user must explicitly indicate this to Aura.

The main feature of Aura is the ability to guess the intent of the user, and prepare the system for the task at hand. The dynamic migration expects constant network connections between the different environments, which limits the applications of the system.

CARMEN

CARMEN primarily deals with resources in wireless settings with temporary connection loss [13]. CARMEN is based on the idea of *proxies*, mobile agents which reside in the same environment as the user. If the user moves to another environment, the proxy will migrate, thus allowing the mobile client to access it in the new environment. Each mobile user has a single proxy. The primary function of the proxy is to provide access to resources needed by the user. When the proxy migrates, resources must be made available in the new environment, and this can happen in either of 4 different ways:

1. Moving the resources with the agent.
2. Copy the resources when the agent migrates.
3. Using remote references.
4. Re-binding to new resources which provide similar services.

The method chosen depends on the *profile* of the resource.

Each entity in CARMEN has a profile. *User profiles* contains information about preferences, security settings, and subscribed services, among other information. *Device profiles* define the hardware and software of devices. *Service component profiles* define the interface of services and finally, Site profiles group together the profiles which all belong to a single location. That is, agents in CARMEN keep track of the location of the user, and uses the profile of the entities to migrate accordingly.

CARISMA

CARISMA explores the use of application provided profiles to adapt middleware to the needs of the applications [27, 28].

Profiles are kept as meta-data of the middleware and consists of *passive* and *active* parts. The passive parts define actions the middleware should take when specific context events occurs, such as shutting down if battery is low. The active information defines relations between services used by the application and the policies that should be applied to deliver those services. The active part is thus only used when the application requests a service.

Different environmental conditions may be specified, which determine how a service should be delivered. At any time, the application can use reflection to alter the profile kept by the middleware through an XML representation.

To deal with conflicts between profiles, CARISMA adopts a micro economic approach [26], where a computing system is modeled as an economy where consumers makes a collective choice over a limited set of goods. In this case, the goods are the *policies* used to provide services, not the resources providing them. The middleware plays auctioneer in a action protocol, where each application submits a single, sealed bit on each alternative profile. The auctioneer then selects the alternative which maximises the sum of bids. To determine the bid each of the applications are willing to pay, functions which translate from profile requirements to values are defined. Like profiles, these functions may be changed at any time through reflection. This type of protocol makes sense because CARISMA delivers the *same* service to all participating applications.

COOLTOWN

The Cooltown project is intended to enable wireless, mobile devices to interact with a web-enabled environment [39]. The basic principle is that devices, people, and things have a web-presence identified by a URL, which provides a "rich" interface to the entity. Users interact with the web-enabled environment using PDAs to interact with the available web-services. As such, Cooltown expects wireless Internet access when users interact with the system. URLs are passed between devices in local device to device interaction. E.g. a projector might receive a presentation by receiving a URL to the file.

Context in the system is closely tied to the physical environment. For example, an infrared beacon at the entrance of a room will emit a URL which points to the

page of the room [11]. When a PDA loads this page, the PDA acts as an interface to the room, thus changing behaviour based on location context. Other types of context might be used by web-applications by providing web-applications with other context like time or activity. The main principle in the collection of context is that it is provided by web-clients. Depending on which sensors the clients have, web interfaces can adapt to the context they provide. Types of context about the physical world includes:

- Where,
- when,
- who,
- what, and
- how.

The context is integrated with a model of the physical world, consisting of places, people and things, and relationships between them. Relationships include

- Contains,
- isContainedIn,
- isNextTo, and
- isCarriedBy,

and is extensible. Relationships are directional, so like hyperlinks they can be navigated in one direction, making them suitable for presenting as web pages. Relationships have properties, and can be subtypes of other relationships. The state of the model is updated automatically by sensing mechanisms ranging from infrared beacons to GPS.

The main modules in the architecture are

- Web Presence Manager,
- Description,
- Directory,
- Discovery modules,
- Autobiographer,
- Observer, and
- Control.

Besides the modules, Cooltown offer tools to build web-presence services (applications).

CORTEX

The CORTEX project have proposed a middleware [43]. The goal is to support

> "Autonomous mobile physical objects that cooperate with other objects, either mobile or static, by capturing information in real-time from sensors event messages propagated in a MANET" [109].

The middleware is based on sentient objects. A sentient object senses and views the behaviour of neighbouring objects, reason about it, and manipulates physical objects accordingly. Sentient objects dynamically discovers each other, and share context information.

To support sentient objects, CORTEX provides a middleware based on component frameworks, each of which provides a service to the sentient objects:

- Publish-Subscribe,
- Group Communication,
- Context, and
- QoS management.

Publish-Subscribe is used for discovery. While the other component frameworks support communication, context retrieval and inference, and arbitration of resource allocation.

The resulting middleware is configured at deployment time and can be reconfigured at run-time through a reflective API to adapt to changes in the environment.

5.4.3 GAIA

The Gaia Operating Systems is intended to be a *meta-operating system* [99]. That is, a distributed middleware system providing functionality similar to an operating system. Gaia builds on the notion of an *active space*, coordinating heterogeneous devices in a physical space, typically a single room. Like operating systems, it provides

- program execution,
- I/O operations,
- file-system access,
- communications,
- error detection, and
- resource allocation.

Program execution is supported by the *component manager core*, which allows any application to upload components to any node of execution in the active space. I/O operations are supported by device drivers on each node, and the functionality is exported to the rest of the active space using distributed objects. Gaia utilises the *Context File-System*, which stores files based on representation of the context. Both synchronous and asynchronous communication is supported through RPC and events. Applications can register for event notification in case of errors, and react accordingly. Finally, Gaia manages resources throughout the active space.

Gaia is structured like traditional operating systems with a kernel providing the necessary services and applications build with a application framework on top, as illustrated in figure 5.7.

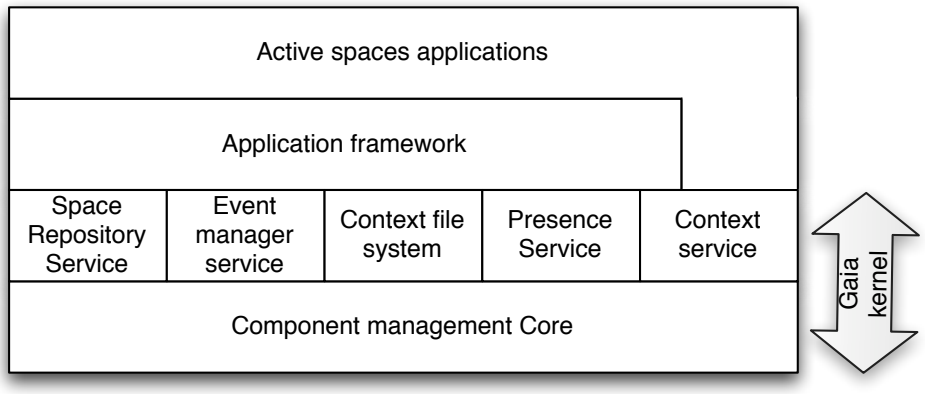

Figure 5.7: The Gaia meta-operating System architecture. Adapted from Roman et al. [99].

Gaia differentiates between location, context, and events and although they can all be seen as different kinds of context, they are handled by different components. Context is collected by *context providers* and higher level context, such as activity, can be inferred from low level context. An additional presence service deals with which entities are present in an active space. Four basic types of entities are supported: application, service, device, and person.

Context is represented by first class predicates and more complex context is represented by first order logic operations, such as *and* and *or*. Applications are notified of changes in context through events, and can react accordingly.

The Context File System builds a virtual hierarchy based on context-information tags on files and presents a directory structure based on context predicates. For example, files associated with the context **location=hopper-338 && present=Kristian** can be retrieved by entering the directory **/location:/hopper-338/present:/Kristian**.

The Context File Systems is built with a single server governing the namespace, while actual files are imported into the active space using existing distributed file systems, such as NFS.

MIDDLEWHERE

MiddleWhere provides advanced location information to applications and incorporates a wide range of location sensing techniques in a model for location [97]. Location information originates in *Location Providers* and is stored in a spatial database. A reasoning engine uses the location information from different providers to determine location and a location service uses the spatial database and the reasoning engine to provide location with a certain probability. The system is illustrated in figure 5.8.

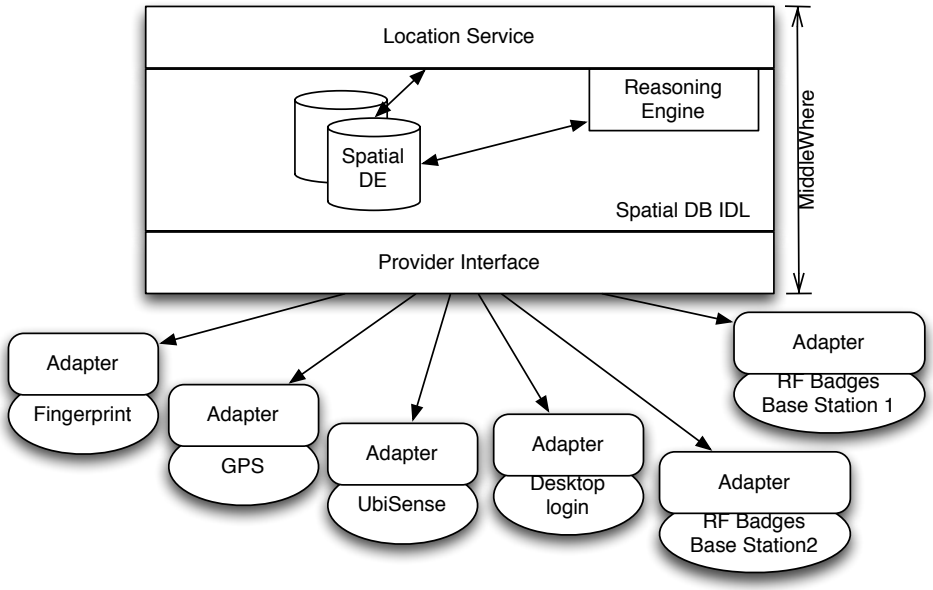

Figure 5.8: The MiddleWhere system. Adapted from Ranganathan et al. [97].

The location model is hierarchical and deals with three different kinds of location:

- points,
- lines, and
- polygons.

Each is represented by coordinates and a symbolic name. Location is represented as GLOBs (Gaia LOcation Byte-string). For example, a desk could be represented as **Hopper/3/338/Desk1** or as **Hopper/3/338/(2,4,0)**, meaning that Desk1 in room 338, floor 3 of the Hopper building is located at coordinates (2,4,0) with respect to the coordinate system of the room. The room will have coordinates with respect to the floor, the floor with respect to the building, and the building with respect to global

coordinates. In this case, the desk is represented by a point. Polygons are used for representing rooms, hallways or spaces within rooms, while lines can be used for representing doors between two rooms.

The systems deals with quality of the location information. The quality is measured according to *resolution*,*confidence*, and *freshness*. Resolution differs widely between different location sensing techniques. For example, a person using a card-reader to enter a room will tell the system that the person is somewhere inside the room while GPS has a resolution down to perhaps 10 meters. An RF badge might have a resolution of 1 meter. Confidence is a measure of how precise the sensor is in terms of probability that the object is within the sensed area. This probability originates in the sensors which register the object and in the case of multiple sensors, the information is fused to yield a single value. Freshness is based on the time since the last sensor reading, and each type of sensor has an associated temporal degradation function which, based on freshness, degrades the confidence in the information.

MobiPADS

MobiPADS is a middleware system for mobile environments [29]. The principle entity is *Mobilets*, which are entities that provide a service, and which can be migrated between different MobiPADS environments. Each mobilet consists of a slave and a master. The slave resided on a server, while the master resides on a mobile device. Each pair cooperate to provide a specific service. Services are composed by chaining them together in specific order, and the slave mobilets on the server are nested in the same order. This provides reconfiguration based on different requirements.

MobiPADS is concerned with internal context of the mobile devices, which is used to adapt to changes in the computational environment. Thus, context types include

- processing power,
- memory,
- storage,
- network devices,
- battery etc.

Each of these have several subtypes, e.g. *size* and *free_space* for memory and storage. Mobilets are provided with changes through context events, which they subscribe to. Higher order context is derived by an *Environment Monitor*, which subscribes to event sources and has the same characteristics as other event sources.

Adaption takes place in either the middleware based on system profiles, or by letting mobilets adapt to the events they receive. Based on the requirements in the profile, the service chains can be reconfigured to deal with e.g. a constrained environment, based on programmer provided alternatives service chains. Applications

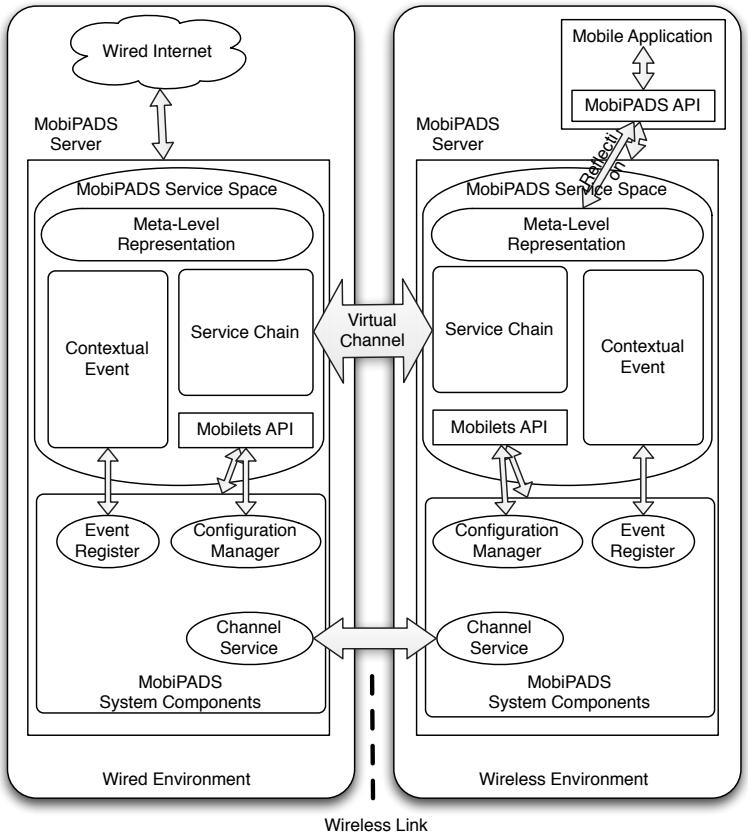

Figure 5.9: MobiPADS architecture. Adapted from Chan and Chuang [29].

have access to reflective interfaces for context, service configuration, and adaption strategies, and can change them to obtain a different service from the middleware.

SOCAM

SOCAM is based on the idea of using ontologies to model context [59]. The model is then used by an interpreter to reason about context. The SOCAM architecture (figure 5.10) consists of

- Context Providers,
- Context Interpreters,
- a Context Database,
- a Service Locating Service, and
- Context-aware Mobile Services.

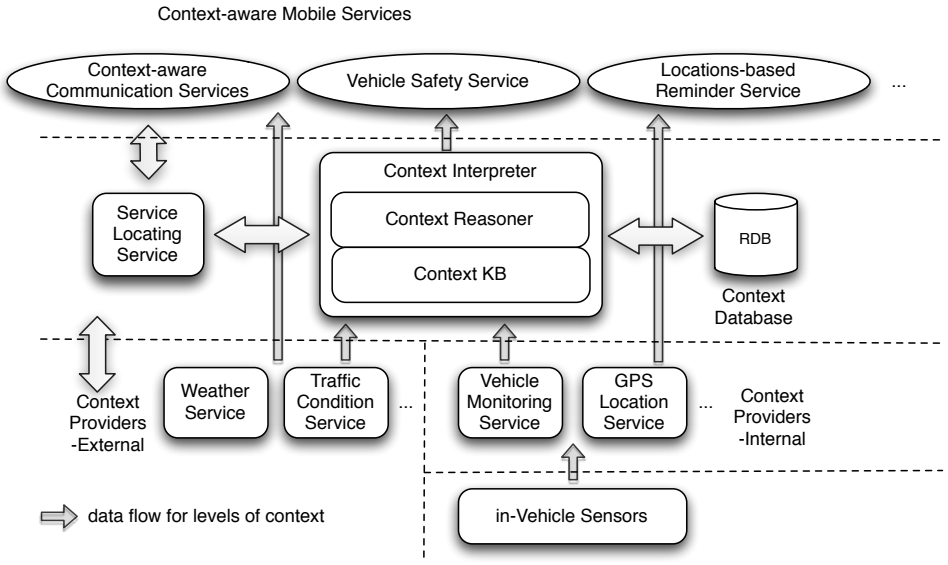

Figure 5.10: The SOCAM architecture. Adapted from Gu et al. [59].

Context Providers provide external or internal context, which can be used by the mobile services directly or by *Context Provider* to provide higher-order context. Externally, the Context Interpreter acts as a Context Provider. Context is represented as instances of the ontology model.

The Context Interpreter consists of a Context Reasoner, and a *Context Database*, which contains instances of the current ontology, either from users or Context Providers. The context is updated by a triggering mechanism with different intervals.

Context Providers register with the *Service Location Service*, thus allowing Mobile Services to locate them. The *Mobile Services* can obtain context either by querying the located Context Providers, or by registering for specific events. SOCAM supports *rules* for specifying which methods should be invoked on when an event is received. The rules are predefined and loaded into the Context Reasoner.

SOCAM represents context as a formal ontology described in OWL. The middleware supports reasoning about context, so that high level context can be derived from observed context by the Context Interpreter. The ontologies are either a generalised ontology, or a domain specific ontology which is *bounded* with the generalised ontology, or *re-bounded* if the context changes. The domain specific ontology may, for example, be re-bounded if the context shifts from an office location to a car. It is the responsibility of the Service Locating Service to load new context ontologies when applications ask for location context. The generalised ontology contains concepts like computational entity, locations, and activities. The domain specific ontologies describe concepts in specific sub-domains, like "office", "home", or "vehicle". The idea is that by only binding the ontology of the current domain, computational burden on resource constrained devices can be reduced.

5.5 Context Awareness in Agriculture

For context-aware applications in agriculture, there are a number of questions to ask:

- What are the sources of context information?
- Which context is useful?
- How do we aggregate and use context?

The first two questions should be answered using participatory design techniques, as described in chapter 3, while the third question is experimental in nature.

To discover the sources of context, we mainly used two kinds of participatory design techniques: field studies and focus groups. The field studies consisted of visits to farms and advisory centres, while focus groups where carried out with professionals from the industry. From these visits, a number of sources of context where identified. Overall, the sources can be divided into internal and external, from the point of view of the individual farm. The local sources include:

- Production data,
- thermometers,
- water meters,
- state from control systems.

Production data can be the current state of crops in fields, information about weed discovered in fields, possibly by an external advisor, and information about livestock. Thermometers are typically placed in stables since changes in temperature can be fatal to livestock. Water meters can be used to monitor for leaks in pipes, but also has other uses, like early disease warning, ad described in the scenario in chapter 1. Control systems for the large number of machinery on a farm can also provide valuable information. For instance, HVAC systems may provide information on air-flow and humidity, while the embedded control system in a pesticide sprayer might record the amount of pesticide used.

The external sources include, but are not limited to:

- Advisory services,
- market data, and
- weather data.

Advisory can be static information about best practice or the results of new research that is relevant to the farmer. Different research institutions constantly perform research and tests with a wide variety of crops and cultivation methods, which may alter existing knowledge of best practice for a particular crop, perhaps depending on the type of soil the crop is growing on, or how it has been treated previously. This information is highly relevant to any farmer with fields that matches the prerequisites of the research being carried out, for instance because research is only relevant to a specific crop growing in a specific type of soil, but irrelevant to anyone else, thus making the relevance of the updated advisory information context-dependent. As another example consider an existing Danish service, *Crop Protection Online* [1]. This service is intended to provide plant producers with information that is relevant to any particular crop the might have on their fields, for instance providing information about tasks related to crop protection depending on the stage of the crop, as shown in figure 5.2. Besides giving information about the tasks, each task links to a service which will help determine the correct pesticide and amount of pesticide to use. While the current version is surely useful, if we have information about the current state of the fields on a farm, we could provide this information when it is needed.

Information from the market includes the current price of fodder and livestock. This is important, since it can affect business decisions at the farm.

Weather data is particularly useful for plant production, providing estimates for the best time to treat fields. It may be used to automatically update work schedules, or by raising alarms when a previously assigned task has to be changed, for instance because of incoming rain or too much wind.

Determining the usefulness of the various kinds of context is a crucial task. In general, we have found that it is easy to convince users, developers as well as end-users of applications, that a context aware integration of all available information relevant to a farm is useful. However, our project does not have the resources necessary to integrate all available information sources, so the immediate task is to

Table 5.2: Excerpt from Crop Protection Online [1]. Season plan for crop protection in Spring Barley.

Crop stage	Description of tasks	Precision in timing	Comments
10 – 13	Inspect for seeded weeds.	When weeds have about 2 true leaves.	Some fast growing species, e.g. crusifers, are allowed to have up to 3-4 true leaves. When applying herbicides to very small weeds, there may be a risk that new flushes of weeds will appear after the initial treatment. I cereals, additional treatments will normally be needed only in high densities of species, which kan germinate from relatively high soil depth, e.g. Cleavers.
...	...	...	...
30 – 39	Inspect for perennial weeds and new flushes of seeded weeds.	When perennial weeds have 6 leaves.	See 'Green leaflet' on Creeping Thistle (in Danish).

determine the most useful context, and then provide a model for that context, while keeping in mind that it should be extendable.

To enable this extendibility, we have looked into existing context models. The most compelling choices, which offers the best flexibility, seems to be either semi-structured models, logic-based models, or ontology-based models.

When it has be decided what context information to gather, it becomes crucial to decide *how* to gather it. For our domain, we observe that infrastructure is relatively static, in the sense that most sources of context do not move. This include sensors placed in buildings and most equipment. The main exception is equipment used in fields and equipment carried around by workers, like PDAs, which could be used to provide the location of workers. Another observation is the potentially large amount of data to process. The two observations leads us to conclude that some central processing of context is useful. This greatly simplifies aggregation, since it can take place without querying distributed sources of context. For these reasons, we have proposed a model where context-sensors and consumers of context are distributed,

while aggregation of context is centralised.

5.6 Ontology Based Context Models

Since the idea of ontology based context models is not new, as described in section 5.2, we have analysed the use of standard tools for creating and manipulating the model. For this, we have chosen an OWL based model using the freely available Protege-OWL tool and its Java API bindings [77].

In this model, types of context become concepts in an ontology, and specific information becomes instances. This allows us to express all of the various kinds of context previously discussed in section 5.1. Since ontologies allows us to represent the structure of data, profiles of entities can be described using a modelling technique similar to the one used in XML-based profile models.

5.6.1 Context Model

Our standard context-model is quite simple, mainly aimed at supporting our agriculture scenarios. However, it is extensible, and uses our publish/subscribe mechanism for accessing context, described in detail section 7.1. Furthermore, we have extended it by using a context model from the Hydra project [101], simply by addition of that context model using the ordinary import mechanisms of OWL. Our **Context Service**, shown in figure 5.11, is relatively simple when shown as a UML model of the main classes. The reason for this is that most of the functionality actually is implemented in the two classes provided by Protege-OWL, **OWLModel** and **SWRLFactory**. To start with the other classes, **OwlContextService** is simply a facade class suitable for exposing as a Web Service, which uses **BonjourDiscovery** for advertising its availability using Apple's Bonjour implementation [68]. **OwlContextModel** is an implementation of a generalised interface for Context Models, which encapsulates some of the specifics of using OWL for the model. While **OwlContextModel** uses concepts and instances for modelling context internally, **OwlContextModel** provides an interface that deals with context events. As would be expected, **OwlContextFactory** is simply a factory class. **OWLModel** is the actual semantic model which is manipulated by the context service. In our implementation, it is an instance of the **JenaOWLModel** provided by Protege-OWL for use with the Jena API [62] and the Jess reasoner [100] for processing SWRL. This class is relatively low-level, but supports operations for insertion, deletion, and updates on the model and includes factory methods for that purpose. As an example of the use of **OWLModel**, listing 5.1 shows the creation of an individual of the concept `Event`, with its `data` property associating it with the `ContextData`. In our model, the `data` property of events are not constrained, and may reference anything.

The context model is accessed through a web-service interface that allows for subscribing to and publishing events listing 5.2 (the current implementation does not allow for unsubscribing). A publish event is simply a string containing XML corresponding to an event in the OWL ontology used for context model, while a

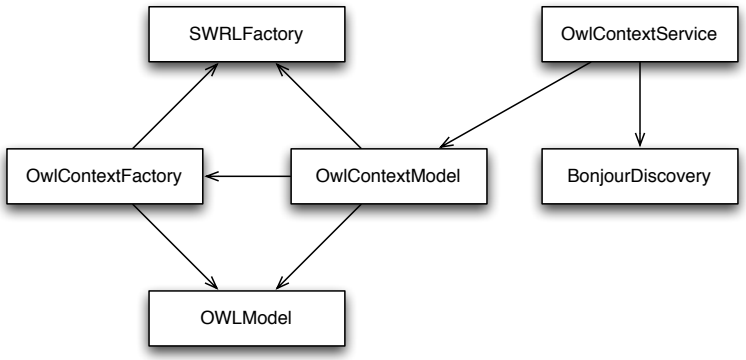

Figure 5.11: Simplified UML model of the **Context Service**, showing the main classes involved.

Listing 5.1: Creating a new individual of type `Event` named "ContextEvent" and setting its data property to point to the "ContextData" individual which is already present in the ontology..

```
OWLNamedClass eventCls = this.owlModel.getOWLNamedClass("
    Event"));
OWLIndividual ind = eventCls.createOWLIndividual("");

OWLIndividual data = this.owlModel.getOWLIndividual("
    ContextData");
OWLProperty subscribesTo = this.owlModel.getOWLProperty("
    data");
ind.setPropertyValue(subscribesTo, imp);
```

subscription is the name of the subscriber and an SWRL rule in the format used by Protege-OWL. When subscriptions match an event, a notification containing the event is sent to the subscriber. The actual subscription works like ontology based publish/subscribe, which is explained in section 7.1.

For our purpose, we have only created a simple context model with a few concepts necessary for the application (see section 4.2), and focused on the tools necessary to manipulate it. To create a "full" context model for agriculture, domain expertise is needed. Such a model would describe all the concepts used in provided services, allowing for using the same model for describing data that is used for input and output in services since this is needed for semantic service matching (as described in section 7.3). This poses a number of problems, like contradictory definitions of concepts in different areas of agriculture. As a simple example, take the "farm" concept in Denmark. In relation to reporting used fertiliser, a farm is a unit which produces a number of animal units and has a certain area on which to spread manure. This unit may be co-owned by several farmers.

Listing 5.2: The interface of the **Context** service as Java-like pseudo code. The WSDLinterface is provided in appendix B.2.

```
public interface ContextService {
    public void publish(XML event);
    public void subscribe(String subscriber, SWRLRule
        subscription);
}
```

5.7 SUMMARY

By now, context awareness has already had a long history in ubiquitous computing. In this chapter, we have analysed ubiquitous computing and middleware to support it, as well as how context has been defined and used. We looked into various previous context aware middleware systems, and discussed their properties and described the context model used in the Kilo project.

Our context model, while being intended for a specific domain, contains concepts that are of general purpose. However, the use of OWL allows for splitting the model into appropriate subsets, which can then be used in context models for other domains. In fact, parts of our model originate in another project. Another aspect of ontologies for context is that the model itself can integrate domain knowledge, and existing ontologies containing domain knowledge can be integrated into the model. However, this is a job for domain experts.

Chapter 6

First Class Connectors

In traditional programming languages, whether procedural, object oriented or functional, there is a schism between "pure" computation and side-effects, all of which can be related to input and output in some way, whether between applications and the operating system or between components. Note that we use input and output in a very general sense, including communication between entities. This is most notable in functional languages, where the core languages typically do not have any side effects at all, but is also obvious in other languages, where side effects can be difficult to handle. Traditional solutions have included using threads or timeouts to handle indeterminate delays when handling input and output. However, these methods can be seen as attempts to add constructs for *communication* into languages which are actually intended for *computation*. Instead, separate *coordination languages* which implement constructs for input and output have been suggested, as described in section 6.1.

While the idea of a separate coordination language, like Linda [54], is a good idea, the question remains on how to implement it in an existing language. The most obvious idea is to implement it as an API. However, if a language supports *first class connectors* it can instead be implemented as a connector. Conceptually, a connector embodies the coordination of a computer program, just like a coordination language does. figure 6.1 shows the relationship between the concepts.

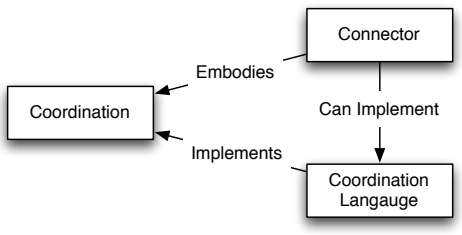

Figure 6.1: Ontology of Coordination Concepts.

In this chapter, we look into first class connectors and their relation to *coordina-*

tion languages and *architectural description*. We then explain why first class connectors are useful in middleware, and follow this up by describing a system we created for prototyping service oriented architectures based on web services using the first class connector abstraction available in ArchJava [4].

6.1 First Class Connectors

The seminal paper on first class connectors is due to Shaw [107]. The author argues that the then current practice in software architecture was insufficient for ensuring a correct implementation in code, identifying a number of problems with the current development practice:

- Inability to locate information about interactions,
- poor abstractions,
- lack of structure on interface definitions,
- mixed concerns in language specifications,
- problems with components with incompatible packaging,
- poor support for multi-language systems, and
- poor support for legacy systems.

The first problem is due to the use of import and export systems to link components located in different modules. This leads to problems with namespace clashes and forces an asymmetrical relation, where one component uses another, but not the other way around. The observation here is that although many relations are asymmetrical, some are inherently not. This is particularly true for peer-to-peer systems. Furthermore, structural information is spread out in many parts of the system, making it difficult to determine or review the structure of the system.

The second problems deals with how system architecture is described at design time. Traditionally, box-and-line diagrams have been used for describing the structure of a system, whether the diagram is formal, for instance UML, or informal. In practice, interactions are complex, with rules for instantiation and protocols for interaction, but this cannot be captured by such a diagram, and programming languages have no mechanisms for capturing design decisions. For this reason, component interfaces are ill-defined, and requirements between components are implicit, since there is no single logical place for them.

The lack of structure on interfaces give rise to two distinct problems. Firstly, there is no abstraction for connections, which could be used to aggregate primitive operations. Secondly, interfaces cannot be decomposed into roles with different functionality, abstracted into different connections.

The problem with the mixed concerns of language specifications is that most programming languages are designed for implementing algorithms and manipulating data structures, not for supporting implementation of complex inter-component structures.

Components might have incompatible packaging, event though they basically provide the same functionality. For instance, it might not be possible to reuse an algorithm from a library because the algorithm is interactive, while the need is for a non-interactive version of the algorithm. Furthermore, two implementations of the same algorithm may use different naming, requiring the programmer to write conversions.

While connections between components are independent of language, assumptions about interaction mechanisms and data representation often require the use of a specific language. If implementations in different languages have to interact, a method for describing and implementing interactions must be present.

When interaction and architecture is not immediately available in a system, creating changes or building new functionality is often difficult, since the architecture as well as the intention of the interactions in the system are not immediately visible.

Before Shaw, Gelernter and Carriero identified many of the same problems, but suggested a different solution in the form of Linda [54] . Linda is not a full programming language, but a *pure coordination language* . The idea is that a complete programming language consists of:

- A computational model, and
- a coordination model.

The computational model implements single-threaded algorithms, while the coordination model implements interaction for each algorithm with the environment. Like Shaw, they observe that traditional programming languages are mainly intended for implementing algorithms and usually implement coordination in an ad-hoc or unstructured way, and for this reason should only be used as a computation language . The authors specifically state, that the suggestion that traditional languages are incomplete is intentional.

Coordination includes everything that is not pure computation, including:

- Process creation,
- communication, and
- synchronisation.

This solves the problems with incompatibilities between components due to naming, packaging and languages.

Linda implements the notion of a *tuple space* (figure 6.2). A tuple space is a general purpose, persistent storage space, where applications can store and retrieve tuples containing any data. Data is stored by simple putting tuples into the space,

and retrieval is done using pattern matching on the contents of tuples. If more than one tuple matches a pattern, a single random tuple is returned. By default, retrieving a tuple removes it from the tuple space, although some implementations allow for non-destructive reading of tuples.

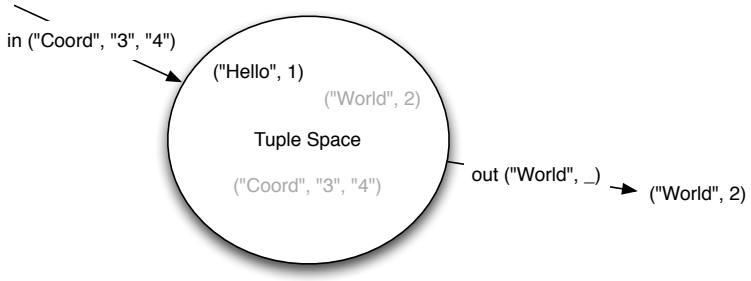

Figure 6.2: A Tuple Space is a persistent store where tuples can be written and retrieved.

The basic operations on a tuple space in Linda are:

- `out(t)`,
- `rd(t)`,
- `in(t)`, and
- `eval(t)`.

`out(t)` places the tuple `t` into the tuple space while `rd` and `in` respectively read and retrieve a tuple from the tuple space. When a tuple is retrieved, it is removed from the tuple space. `eval` inserts an *active* tuple into the tuple space. An active tuple is a process, which runs in the tuple space while the issuing process continues in parallel. When the process terminates, the tuple is replaced by the output of the process.

While Linda manages to separate computation and coordination, and clearly defines an interface for communication between components, it does not address the other issues identified by Shaw. Coordination is still interspersed with other concerns in code, and interactions are not clearly defined in terms of component interfaces and roles. On the other hand, it defines a concise language for expressing coordination.

The suggestion of Shaw is to define semantically rich connectors in architecture descriptions, and connector constructs in languages which support:

- Defining the semantics of connectors and their compositions.
- Generalise the rules for importing and exporting symbols to address asymmetry, multiplicity, locality, abstraction, and naming.
- Establish type structures for components and connectors, including taxonomies.

- Define rules for appropriate architectural abstractions.

While components essentially correspond to traditional units of compilation, possibly composite, connectors have no representation in traditional programming languages. However, connectors could be defined using structures similar to the way components are defined, as illustrated in figure 6.3.

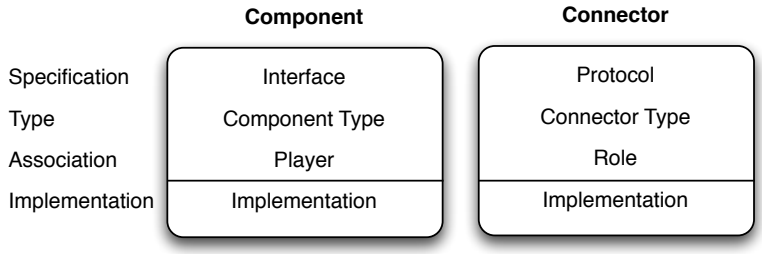

Figure 6.3: Structure of an architectural language, consisting of a specification part with units of association for system composition and an implementation part. Adapted from Shaw [107].

While components have interfaces in this model, connectors have a protocol defining interactions between components. Both connectors and components have some implementation, which in principle can be in any languages. For composition, components acts as players in a composition, playing a certain role in a connector. Both components and connectors are typed, but use two distinct type hierarchies.

The proposed model is only a coarse grained suggestion. In actual implementations, roles will also usually have an interface, defining how components interact with them.

A way of combining the best of both approaches is to implement the coordination language as a connector. This gives a well-defined construct for accessing coordination. Furthermore, such a relatively simple connector can be used to implement more complex connectors which can then implement the communication protocol. This is described in more detail in section 6.2.

Dashofy et al. have described methods for using off-the-shelf middleware for implementing connectors [38]. The work is in the context of the C2 architectural style [113]. They evaluate a number of existing middleware systems according to a number of requirements:

- inter- and intra-process communication support,
- features of software connectors,
- platform and language support,
- communication method,
- ease of integration and use,

- multiple instances in an application,
- support for dynamic change, and
- performance.

Interprocess communication is a necessity for building distributed systems, so middleware should support connectors which span isolated processes transparently. Their suggestion for adding transparent connectors to C2 involves encapsulating access to the middleware in C2 ports, and then sending messages using the middleware. However, they observe that this does not isolate communication to the connector, and instead suggest a method where a connector is instantiated in each process, and they then encapsulate communication between them.

Medvidovic observes that this method can be used for ensuring architecture conformance in distributed systems [85]. This is basically the same arguments as for non-distributed systems, with middleware being used for implementing distributed connectors, whereas objects may be used to implement them in non-distributed environments.

6.1.1 ARCHITECTURAL DESCRIPTION

While architectural description has been used for design-time descriptions of middleware, it can also be made useful for developing applications in a more direct manner by utilising architectural descriptions when programming for a specific middleware, and for observing the run-time behaviour of applications.

For applications, we can use architectural descriptions to setup applications on load-time. To do this, a loader takes as input an architectural description and loads the necessary components and sets up the connections between them.

The idea of first class connectors originate in the work of Shaw [107], but formal architecture description have used it early on. In architecture descriptions, connectors describe interaction between modules, usually described in a separate notation. In the simplest case, a connector is a statement of the fact that one component uses another component, but to be really useful, connectors should be able to describe the interaction in more detail, and some architecture description languages provide mechanisms for describing protocols and other complex aspects of interaction.

A number of different architecture description languages with first class connectors exist with different emphasis, strengths, and weaknesses. Among them are Acme [53], Wright [5], and xADL [70]. While all three of these systems provide support for connectors, they vary in how they are supported.

What is common for all of these systems is the use of first class connectors in some way. Acme provides description of components and connectors in terms of the ports of the components, the roles of connectors, and how roles and ports are attached. It also allows to express higher-level architectural properties, like invariants and heuristics as part of the description. What it does not provide is describing the protocol of the connectors, However, this is one of the main benefits of Wright. Like

Acme, Wright defines software architectures in terms of components with ports, connectors with roles, and their attachment. It also supports defining computations of components and protocols of connectors formally. Furthermore, systems are defined in terms of instances of components and connectors. xADL is more like a meta ADL language, in that the main goal of xADL is to provide an XML based language for using ADLs in IDEs. For this reason, xADL provides a basic language which is extensible. The basic language includes components, connectors, and architectures defining the links between components and connectors. It does not provide formal descriptions of protocols, ports, and roles. However, since the language is extendible, this can be added if needed.

6.2 First Class Connectors in Programming Languages

While many, notably Shaw [107], have argued that connectors should have first class status in programming languages, only a few, experimental, programming languages have implemented it. For instance, Chen et al. have described a component based programming language with first class connectors for interaction [30]. The language, SAJ , is based on a subset of Java called ReIJ which has first class relationships between classes [16]. Another example is ArchJava [4] , which is implemented on top of a full version of Java. ArchJava programs are compiled into regular Java, which can then be processed by the Java compiler (figure 6.4).

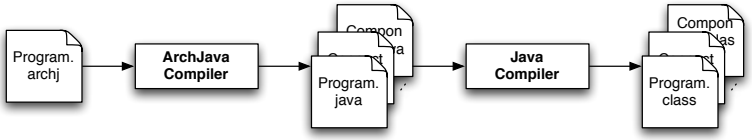

Figure 6.4: ArchJava programs compiles into Java which is then compiled to Java bytecode using the standard compiler.

The constructs for components and connectors are based on regular Java constructs for classes, but as suggested by Shaw, they fall into two distinct type systems. Each component defines a set of ports, which can either *provide* or *require* methods. Listing 6.1 and 6.2 shows the components of a simple echo system, where an **EchoService** provides three different methods required by an **EchoClient**. To actually use the system, the architecture has to be defined by expressing connections between the constituting components. This is done using the ***connect*** keyword, as illustrated in listing 6.6, where an **EchoSystem** is defined as consisting of an **EchoService** and an **EchoClient**, with the single ports of the two components being connected. The resulting system is shown in figure 6.5.

As such, a component is simply a special kind of object, which only communicates with other components in a structured way, as defined by connecting the ports of the components using the ***connect*** keyword. The only other kind of communication between components allowed, is for a component to call public methods on

Listing 6.1: Simple Echo component in ArchJava. The component offers a port with three methods.

```
public component class EchoService {
    public port service {
        provides String echo(String s);
        provides String concat(String s, String s1);
        provides int echoInt(int i);
    }
    String echo(String s) {...}
    String concat(String s, String s1) {...}
    int echoInt(int i) {...}
}
```

Listing 6.2: Simple Echo client in ArchJava. The client requires three different methods.

```
public component class EchoClient {
    public port client {
        requires String echo(String s);
        requires int echoInt(int i);
        requires String concat(String s, String s1);
    }

    public EchoClient() {
        ....
    }
}
```

the components it has instantiated, called *subcomponents*. In this way, composite components can be created in a hierarchical way, since components are allowed to invoke methods in subcomponents.

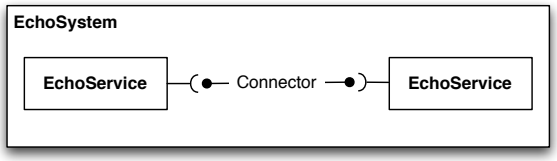

Figure 6.5: The result of instantiating and EchoSystem component.

The semantic of ***connect*** is to connect provided methods with required methods, using their name and signature in two or more ports. At compile time, it is checked that each required method is connected to a unique provided method. In the previous example, provided methods are implemented in their component, but they may also be implemented directly in the port or delegated to sub-components.

Listing 6.3: Defining relationship between components in ArchJava.

```
public component class EchoSystem {
    private final EchoClient echoClient = ...
    private final EchoService echoService = ...
    connect echoClient.client, echoService.service;

    public static void main() {
        EchoSystem.system = new EchoSystem();
        ...
    }
    ...
}
```

Connectors can be either simple or complex in ArchJava. In the simple case, there is only a single, built-in connector, which can connect two components in the same JVM. More complex connectors can be implemented by creating subclasses of the **Connector** class. Furthermore, connections between components can be defined dynamically using port interfaces and connect patterns. An example of this is the use of ArchJava to prototype service-oriented architectures, described in section 6.4.

A similar language which provides abstractions for components and connectors has been presented by Chen et al. [30]. The core of the language is based on a subset of Java called ReIJ which is formally defined [16], allowing a formal definition of SAJ. ReIJ has extensions for first class relationships, which are used for defining first class connectors. The goal is essentially the same as in ArchJava, namely providing enforcement of conformance between the architectural specification and its implementation. In contrast to ArchJava, the definition of a connector is limited to an entity which mediates interactions between components, consisting of a group of roles and interaction protocols. Like ArchJava, components have public interfaces defined by ports.

SAJ has two distinct connector types: *active* and *passive* . The active connectors describes collaboration between components and initiates control. That is, they encapsulate control, while passive connectors are short lived and encapsulate communication (figure 6.6). Passive connectors only provide direct communication, so they are always in a binary relation between two ports. When using passive connections, control is instead the responsibility of components. For this reason, active and passive connectors are intended for different scenarios, with the active connectors being suitable for centralised architectures, and the passive connectors for decentralised architectures.

Contrary to ArchJava, SAJ uses connectors internally in composite components to compose sub-components. Composite connectors on the other hand, are composed of both components and connectors.

The most common primitive connectors are pre-defined, and include:

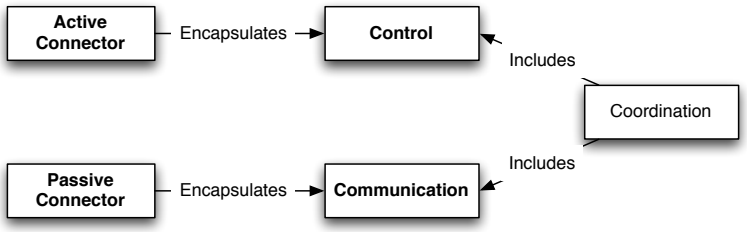

Figure 6.6: Relation between coordination and connector types in SAJ.

- Procedure call,
- pipeline, and
- data stream.

User defined connectors can be created by inheriting the primitive connectors, and will thus belong to one of the same groups in a connector taxonomy as the primitive connector.

Ports are defined in terms of Java interfaces instead of simply methods. As such, entire interfaces are either required or provided. Systems are in turn defined using a static and a dynamic part. The static part defines the structure of an interaction, in terms of components and connectors. The **EchoSystem** could be described as shown in listing 6.4.

The most notable difference from ArchJava is that instead of defining interactions between components in a separate **EchoSystem** component, they are defined in the **EchoConnector** connector. This implies that the client will actually have to implement methods the connector can access, since the client component is completely passive in the interaction. On the other hand, every part of an interaction is completely encapsulated by the connector, whereas in ArchJava, a composite component will contain the interaction protocol, and the connectors only encapsulate the communication protocol. For behaviour more similar to that of ArchJava, a passive connector can be used instead, making SAJ a more flexible language.

6.3 First Class Connectors in Middleware

In middleware, first class connectors allow for easier implementation of reuse of components as well as easier implementation of adaptability. While some systems, like OSGi or publish/subscribe based systems, rely on well defined interfaces for allowing flexibility in adding and removing components at run-time, using connectors for encapsulating coordination issues can be used to support this flexibility instead. Specifically, systems like OSGi can be seen as a component and connector system that only supports a single connector. Furthermore, connectors can also be exchanged at run time, allowing for changes in the environment or for mobile applications moving between environments.

Listing 6.4: Static model of a simple echo system in SAJ.

```
Interface IEchoService {
        String echo(String s);
        int echoInt(int i);
    String concat(String s, String s1);
}
Component class EchoService extends Component {
        Public Port service {
                Provides IEchoService;
                Requires IEchoClient;
        }
        Public String echo(String s) {
                ...
        }
        ...
}
Interface IEchoClient { ... }
Component class EchoClient extends Component {
        Public Port client {
                Provides IEchoClient;
                Requires IEchoService;
        }
}
Connector class EchoConnector extends Connector {
        void EchoConnector(IEchoService echoService,
           IEchoClient echoClient) {
                this.echoService = echoService;
                this.echoClient = echoClient;
        }
        public void DoEcho {
                ...
        }
}
```

More generalised connectors in middleware provide additional flexibility in terms of reusability and flexibility. While systems like OSGi provide flexibility in terms of the components of the system, they do not provide flexibility for the communication paradigms used. They either rely on a single paradigm, or requires the communication to be embedded in components. Using first class connectors, the communication is abstracted from the computation, separating concerns and allowing for easier distinction at deployment and run-time. In the next section, we show how that can be utilised for prototyping systems.

6.4 PROTOTYPING SERVICE ORIENTED SYSTEMS

When designing distributed systems it is useful to be able to rapidly create architectural prototypes of designs. Simple prototypes exploring components and the relations between them may simply be created in any programming language. This allows for exploring a number of architectural qualities like modifiability and implementability, but not others like performance.

For the purpose of rapidly prototyping service oriented systems based on web services, we have created a framework in ArchJava [4]. The framework allows for service-enabling components defined in ArchJava or connect them to existing web services without changing them. In this way, developers can start with a simple prototype using direct method invocation on components, and then change connections between components to use web service invocations.

Central to the framework is a custom ArchJava connector, **WSConnector**, which has two primary uses: Connecting two ArchJava components where one is instantiated as a web service, and connecting an ArchJava component to an existing Web Service. When connecting two ArchJava components, one of the components will actually be run as a web service, while the other will connect to it. The framework assists in this, be supplying the necessary building blocks. This implies, that the client component can only have required methods in the port that connects to the service, while the service component can only have provided methods in its service port. However, it is possible for a service to utilise other services by calling them through other ports during invocation of the service. The framework uses the Axis2 application server to run the web service [6], and the service object is only instantiated upon the reception of a message.

As mentioned earlier, the ArchJava connector is a single role in the conceptual connector. The other role is represented by the **ServerRole** class, which acts as a message receiver in an Axis2 service. The architecture of a system with one client connected to a single service is illustrated in figure 6.7. **WSConnector** uses Axis2 to call the remote web service, while the service is instantiated by Axis2, with **ServerRole** receiving the message and delegating it to the service component.

WSConnector uses external classes for marshaling and demarshaling, which allows for changing the encoding of messages.

Invocation through the connector in Axis2 is accomplished by creating a `ser-`

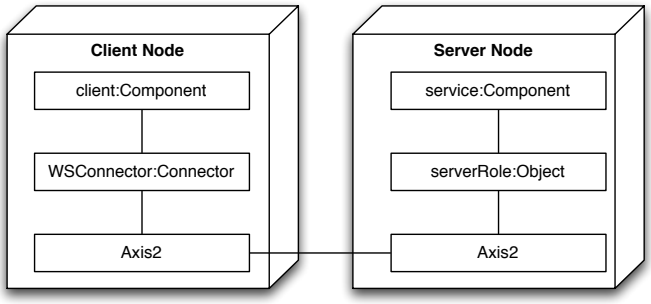

Figure 6.7: Connecting a client and a service through an ArchJava connector using Axis2. Axis2 instantiates the **ServerRole** object, when a message for the service is received. **ServerRole** then instantiates the actual component.

Listing 6.5: `services.xml` defining the Echo operation implemented by the class **EchoService** with **ServerRole** as message receiver.

```
<service name="Echo">
<description>
        Simple Echo Service for testing purposes.
    </description>
    <parameter name="ServiceClass" locked="false">kilo.
       prototype.axis.echo.EchoService
    </parameter>

    <operation name="echo">
        <messageReceiver class="kilo.prototype.axis.
           connector.ServerRole"/>
    </operation>
</service>
```

`vices.xml` file which specifies **ServerRole** as the message receiver for the service, as illustrated in listing 6.5. When Axis2 receives a request for the **Echo** service, it will instantiate **ServerRole**, which can in turn instantiate the service object, the class of which is provided by the Axis2 framework.

Marshalling

To avoid reliance on WSDL-files, the ArchJava connector uses reflection over method signatures to implement marshaling and demarshaling. This is an important part, since it is what allows services to be used without having to define the exact marshalling used. ArchJava provides its own reflection implementation in the **archjava.reflect** package while Java uses **java.lang.reflect**. Since ArchJava assumes the first interface, while Axis2 expects types to belong to the second interface, the connector uses **archjava.reflect** on the client side, and **java.lang.reflect** on the server

Table 6.1: Marshalling between Java and XML types are according to this table. It is intended to use the same marshalling as JAX-RPC and Axis2, although they support additional types [82].

Java Type	**XML Type**
boolean	boolean
byte	byte
char	String (of length 1)
short	short
int	int
long	long
float	float
double	double
void	no value represented
java.lang.String	string
array	soapenc:Array

side, in **ServerRole**. This gives rise to some limitations, since they have slightly different expressiveness. For instance, ArchJava does not support reflection over arrays of simple types.

The marshaling attempts to follow the same conversions as Axis2 and JAX-RPC but since documentation for those two packages are incomplete, some variations may occur. For example, the only type of array supported is arrays of **java.lang.String** and arrays of complex types. This is due to a limitation in ArchJava's support for reflection over arrays, since arrays of simple types are not supported.

Complex Types

Since the framework is intended for prototyping, using simple types is often sufficient. However, when connecting to existing services, it might be necessary to use complex types. Luckily, WSDL-files will be available for existing services, and classes implementing complex types can be generated from these files, although most generators have limitations in which types can be generated. Another possibility is to create such types by hand.

The framework uses Axis2 for web service support. Currently, Axis2 only supports the Doc/Lit style [25], and not all complex types can be successfully generated.

6.4.1 Example Systems

To use custom connectors, connect patterns for the connector must be defined. As an example, a pattern for connecting the **Echo** port of an **EchoClient** component with the **Echo** port of an **EchoService** component is shown in listing 6.6. The sig-

Listing 6.6: Using **WSConnector** to connect an **EchoClient** component and an **EchoService** component. The code defines a connect pattern, and will be called to instantiate a new **WSConnector** when **EchoClient** creates a new EchoPort. The Echo port of **EchoClient**. `connect` matches the constructor of the connector used, while the rest of the methods defines which methods the port require from the port is should be connected to.

```
connect pattern EchoClient.Echo, EchoService.Echo with
   WSConnector {
    connect(EchoClient sender, URL service) throws
        IOException {
         return connect(sender.Echo, EchoService.Echo)
             with new
              WSConnector(connection, service, "echo", "
                 echo", 1);
    }
}

public component class EchoClient {
        ...
        public port interface Echo {
                requires connect(URL service) throws
                    IOException;
                requires String echo(String s);
        }
        ...
}
```

nature of the pattern correspod to the required `connect` method of the **Echo** port, also shown in listing 6.6. The method corresponds to the constructor of components, and connections can be instantiated dynamically by using the keyword *new* on an **EchoPort**.

6.4.2 Usage

To change a regular ArchJava program into a program using **WSConnector**, it is necessary to rewrite connect expressions to use the custom connector. This is done by introducing connect patterns as illustrated in listing 6.6. The connect pattern defines *how* connections are instantiated. In this case, the patterns takes a URL to the service, while the other arguments to the constructor of the connector are predefined by the pattern. If the service URL is always the same, this could also be defined by the pattern.

Listing 6.7: The Port of the Google Client. The required methods match the WSDLdescription of the Google API.

```
public port Google {
    requires connect(URL service, String prefix, String
       namespace, int type) throws IOException;
    requires GoogleSearchResult doGoogleSearch(String key
       , String q, int start, int maxResults,
        boolean filter, String restrict, boolean
           safeSearch, String lr, String ie, String oe);
    requires String doGetCachedPage(String key, String
       url);
    requires String doSpellingSuggestion(String key,
       String phrase);
}
```

CONNECTING TO REAL WEB SERVICES

Since service components are actual web services , it is possible to connect clients to existing, "real" web services (as opposed to components exposed as web services for testing). As an example of this, we have created a simple client which connects to the Google SOAP API [56]. To access an existing service, the interface must be translated to a Java interface. While it is certainly possible to do this automatically, currently it must be done by hand. listing 6.7 shows the **Google** port of the client. Due to type checking in ArchJava, a corresponding port which provides the methods must also be created, although it will not be used.

As a simplification, `doGetCachedPage` returns a String, which will be the base64 encoding of the cached page, while `doGoogleSearch` returns an object of type **GoogleSearchResult**, which has been made especially for handling the complex return type. **GoogleSearchResult** cannot be auto generated by Axis2, since the Google SOAP API uses RPC/Enc style, and Axis2 can currently only generate types for the Doc/Lit style.

6.4.3 PERFORMANCE

Although it is possible to use the framework for evaluating many different architecture qualities, one interesting property is its usefulness for measuring performance. This is only possible if the performance of the framework is similar to the performance of real systems, or vary in performance in a way that is comparable to the way real services vary.

A simple measure of performance is the total invocation time of operations. We have used a simple echo service, which just returns its string argument, and as a baseline, we use implementations for Axis1 and Axis2. The results of 1000 invocations are shown in table 6.2, with comparison of the average invocation times shown in figure 6.8.

Table 6.2: Average, minimal, and maximal invocation time of a call to an operation for Axis, Axis2, and **WSConnector** between a client and a remote web service.

System	Average	Minimal	Maximal
Axis	57	41	438
Axis2	66	36	3157
WSConnector	112	42	1835

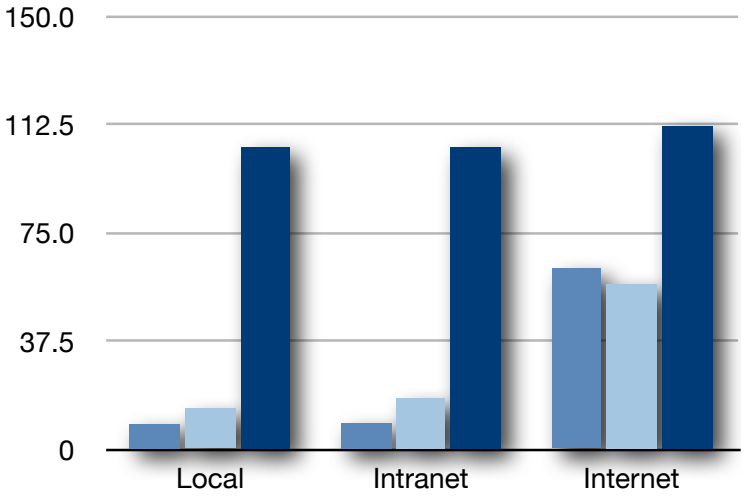

Figure 6.8: Comparison of Axis1, Axis2, and WSConnector. WSConnector is somewhat slower, mostly due to two layers of reflection in each call, whereas both versions of Axis use auto generated, specialised code for marshaling.

As can be seen from the table, there is a significant difference between minimal, maximal, and average invocation time. This is most likely partly due to limitations of the machines running the tests.

WSConnector is considerable slower than both Axis and Axis2. This is most likely due to the cost of reflection. However, for Internet-based communication, the connector provides a performance that is similar (a factor two compared to a factor 10) to the performance of the generated code, making prototypes created usable for performance comparison in this case.

6.5 Summary

First class connectors have many interesting aspects and have been used in many different forms, ranging from design time entities in architecture description lan-

guages to object-like constructs in middleware and frameworks. The idea of first class connectors is in many ways related to the idea for coordination languages, in that they too embody the coordination aspects of a system. In architecture description languages, this is purely a design time description, but may include a semantic description of ports and roles, and even a formal specification of protocol. When first class connectors are reified into compile-time constructs in programming languages, it allows for the reification of the software architecture into the source code of a program. For middleware and frameworks, first class connectors is also a way to structure programs, but are not necessarily distinguishable from other objects at run-time.

We have also shown how first class connectors can be utilised for creating frameworks for testing service oriented architectures, by encapsulating the coordination aspect in the connectors, and letting components act as services.

Chapter 7

Semantic Web Technology

To improve the usefulness of data, it has been suggested to annotate it with semantic meta-data, thus allowing programs to reason about the data. The basis for this is the use of *ontologies*, describing the structure and the content of data. Semantic web technology builds on existing technologies, noticeably XML [20], to create ontologies that are machine readable, and use these to describe meta-data for services and data.

Besides our use of ontologies to describe context (section 5.6), we have used these technologies for implementing a publish/subscribe mechanism using existing tools for manipulating ontologies. The publish/subscribe mechanism, while useful in itself, forms the basis for accessing the context service of our middleware. We also use OWL to describe the deployment architecture of applications and OWL-S for describing services, allowing us to utilise existing research in semantic matching of services for service discovery.

7.1 Semantic Publish Subscribe

Event-based communication is becoming standard in widely different systems: In enterprise systems it is often used in conjunction with message queues such as the Java Message Service [112] and in ubiquitous computing [102], it is often used when processing sensor data for, e.g., context awareness [42]. The promise of event-based technology is *decoupling* of the agents in a distributed interaction: the sender does not need to know, e.g., the address of the receiver and vice versa. A particularly versatile paradigm for event-based communication is *publish/subscribe* in which events are routed from a sender to potentially multiple receivers based on receivers' definitions of interesting events [46].

Even if senders and receivers are decoupled, they need to share a common schema for events and they need to ideally be able to describe both structure and contents of events that they either create or are interested in. Ontologies, and in particular Description Logic-based [7] ontologies such as the Web Ontology Language (OWL; [65]) provide a convenient and analysable way of expressing such schemas.

Eugster et al. define publish/subscribe systems in terms of a basic interaction scheme in which *subscribers* express interest in certain patterns of *events* through *subscriptions* and are subsequently *notified* of the publication of events by *publishers* if the events match their interests [46]. Furthermore, an *event service* provides storage and management of subscriptions and delivers notifications. In this general framework, several types of publish/subscribe systems can be discerned including topic-based, content-based, and type-based publish/subscribe. In topic-based publish/subscribe, subscriptions are essentially named (possibly hierarchical) subjects to which publishers may publish. Content-based publish/subscribe allows subscriptions based on the runtime contents of events and type-based publish/-subscribe in a sense unifies topic-based and content-based publish/subscribe in that it subscriptions are specified using (object-oriented) types that unify hierarchical topic filtering through the type hierarchy and contents-based filtering through (public) members. Ontology-based publish/subscribe provides the expressiveness of type-based publish/subscribe, but adds the ability to create types and abstractions through ontology descriptions. Furthermore, we describe how this can be achieved using available ontology specifications and tools.

Among related work, WS-Notification is an OASIS standard defining a notification infrastructure for use with the WS-* family of standards [57], notably web services and the WS-Resources framework. The framework is a topic based publish subscribe system, with the possibility of filtering notifications based on the contents of the notification or the state of the publisher. For larger systems, a broker may be used to route messages. To publishers, the broker acts as a subscriber, and to subscribers it acts as a publisher, so it can be dropped into existing systems without changing publishers and subscribers. The main drawback of WS-Notification is that the filtering language is not standardised. Instead, each use of the standard in a system will have to decide on a filtering language and implement it for use by providers. This also implies that it is impossible to define the expressiveness of WS-Notification in general, but of course possible for any implementation. As an example, the Apache Servicemix implementation provides no filtering language and instead uses flat topics defined by strings [92], where the ontology-based approach provides increased expressiveness. In contrast, IBM WebSphere uses an implementation of WS-Topics [115], which provides XML based topic descriptions using XML namespaces to avoid collisions, provides for hierarchical topics, and define 4 dialects of a filtering language with increasing expressiveness.

Wang et al. [116] have created an ontology-based publish/subscribe mechanism with a strong emphasis on effective evaluation of filters on the ontology. Their systems represents events as RDF graphs and subscriptions as graph patterns and includes an efficient subgraph isomorphism algorithm. However, they only achieve the high efficiency by introducing some constraints on graphs and graph patterns. Furthermore, there solution is based on DAML+OIL, which has been superseded by OWL.

We have created an ontology-based publish/subscribe mechanism, named **Kilo-.PS**, which we describe in the following sections in terms of design, implementation,

and expressiveness. The approach is evaluated in three ways:

1. by implementing it using OWL and SWRL libraries,
2. by measuring performance of the notification mechanism, and
3. by using it in the Kilo.Two context model.

7.1.1 Conceptual Design

To describe **Kilo.PS**, we take our outset in the formalisation of publish/subscribe by Mühl [90] in which a set of operations on a publish/subscribe system is described. A publish/subscribe system may be defined as:

> *A* publish/subscribe system *is a tuple* $PS = (C, E, N, S)$*, where*
>
> - C *is a non-empty, finite set of* clients *of the system,*
> - E *is a set of* events*, and*
> - F *is a set of* filter *functions from E to a Boolean.*

Informally, then, the runtime semantics of a publish/subscribe system may be described by a *trace of states and operations* on the system. The state of the system is the sum of states of the clients and the state of a client is the filters installed at a client, the events sent by the client, and the notifications received by the client. The operations are one of:

1. $subscribe(c, f), c \in C, f \in F$: The client c subscribes to receive notifications of events, $e \in E$, for which $f(e) = true$. We say that the client subscribes to the filter, f.
2. $unsubscribe(c, f), c \in C, f \in F$: The client c no longer subscribes to f.
3. $publish(c, e), c \in C, e \in E$: The client c publishes an event e. Eventually, clients that are subscribed to e, where $f(e) = true$ should receive the event.
4. $notify(c, e), c \in C, e \in E$: The client c is notified of e. This should only apply to clients that are subscribed to a filter, f, for which $f(e) = true$.

We now describe how this can be realized using OWL-DL and SWRL

> *A semantic publish/subscribe system is a publish/subscribe system described by a tuple* $SPS = (C, O)$ *where*
>
> - C *is a non-empty, finite set of* clients *of the system;*
> - O *is an* ontology *that defines event and filters*

In the present work, ontologies are described using OWL-DL. We represent filters by SWRL rules in which the antecedent constrains the Events for which Notifications are made. The general form of these filters are:

$$Event(?x) \wedge Subscriber(c) \wedge [Filter\ using\ x] \Rightarrow notify(c, x)$$

When this rule is evaluated and an Event exists in the ontology for which the specific filter evaluates to true, then a Notification is created. This leads to the following realization of operations:

1. Subscribe: A client subscribes by adding a Subscriber individual and a rule on the form described above to the ontology.
2. Unsubscribe: A client unsubscribes by removing the added Subscriber individual and rule.
3. Publish: A client creates an Event individual in the ontology the result being that all subscription rules are evaluated. This will create notify properties on each Subscriber individual defining the events they must be notified of.
4. Notify: For each notify property, the corresponding client is notified of the event.

Given this conceptual description, we now turn to implementation and detailed design of ontology-based publish/subscribe.

7.1.2 IMPLEMENTATION AND DESIGN

Kilo.PS is based on standard semantic web technology. The core component of the system is an ontology on which all actual computation is carried out. This implies that the system is inherently centralised, with publishers sending events to a broker, and subscribers subscribing at the same broker (figure 7.1), although it is in principle possible to have more than one broker, even one for each publisher.

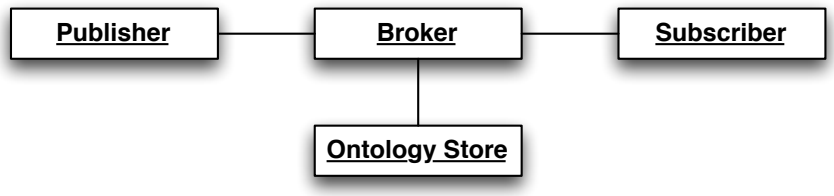

Figure 7.1: High-level system overview.

The ontology is described in OWL-DL, and consists of classes and properties defining the structure of events. Events can have any structure, and contain any data, as long as it is defined by the ontology. Figure 7.2 shows a small example of an ontology in the system, having only a single sub-class of Event, and which doesn't

define any structure on the data contained in the event, except for declaring that it is an individual in the ontology. Further types of events can be defined by sub-classing Event as illustrated by the LocationEvent subclass of event, and the structure of the data of event can be defined by refining the hasData property.

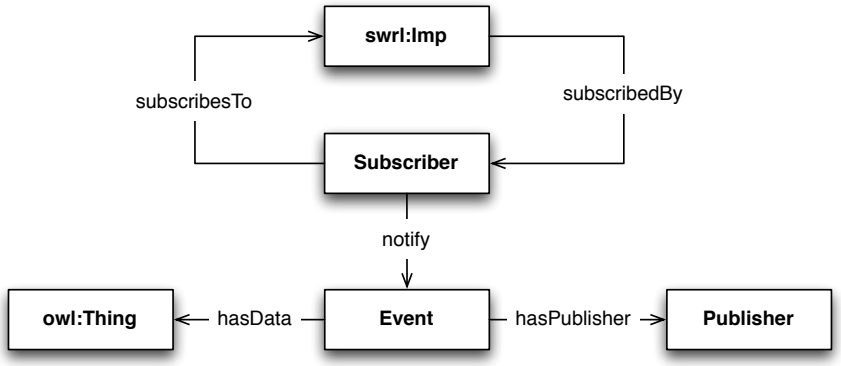

Figure 7.2: The publish-subscribe ontology with sample subclasses of event. *swrl:Imp* is the main concept for SWRL rules which represents the filters.

While publishers publish data by creating new individuals of the Event class or a sub-class in the ontology, subscribers define their subscriptions using SWRL rules. For instance, a subscriber named "Subscriber1" can declare interest in *all* events by using the following rule:

$$\begin{aligned} & \mathit{Event}(?x) \wedge \mathit{Subscriber}(\mathit{Subscriber1}) \\ \Rightarrow \quad & \mathit{notify}(\mathit{Subscriber1}, ?x) \end{aligned}$$

In this example, the filter does not define any specific type or structure on the data of the event. If data of an event is added to the ontology, for instance in the form of a string, it would correspond to a pure topic-based publish subscribe. However, we can do more than that, by using the ontology to define the structure of the data. If this is the case, we can use SWRL rules which match on the contents of the data instead of merely the topic.

To retrieve notifications from the ontology the broker uses SQWRL to retrieve a list of events and the corresponding subscribers [94]. The rule used is:

$$\begin{aligned} & \mathit{Event}(?x) \wedge \mathit{Subscriber}(?y) \wedge \mathit{notify}(?y, ?x) \\ \Rightarrow \quad & \mathit{sqwrl} : \mathit{select}(?y, ?x) \end{aligned}$$

which will return a list of (Event,Subscriber) pairs. The broker can then send each Event to the subscriber. The actual implementation of the Broker uses the Protege-OWL API for storing the ontology [77], and the Protege Jess bridge and Jess for executing the SWRL rules.

There are other implementation issues in regard to publish/subscribe systems which we are not part of the mechanism itself, as discussed by Eugster et al. [46], namely

- *event delivery*,
- *media*, and
- *quality of service*.

Kilo.PS as such does not define how events should be delivered, but the use of OWL based ontologies suggest using XML for serialising events for transport, indicating a message based approach. There is no built-in support for supporting invocations in the subscribers. From the subscribers point of view, to subscribe to events they have to know the structure of the events they are interested in, or at least the type of the event. To subscribe to an event, a subscribe passes its name and a SWRL rule to the broker, and will then receive events as they are processed.

Likewise, the middleware media responsible for transmitting events could be one of several. We have used **Kilo.PS** in a middleware based on web services which is a decentralised architecture using point-to-point communication. However, if there is a need to support for instance temporal decoupling this could be combined with a message queue system resulting in a centralised architecture.

Finally, **Kilo.PS** does not in itself provide any quality of service guarantees, which mostly depend on the middleware used for transmission of events. When combined with web services the only real guarantee is the use of reliable communication. The current implementation does not store events that cannot be delivered.

7.1.3 EXPRESSIVENESS

Eugster et al. [46] describe three types of publish/subscribe systems: topic-based, content-based and type-based publish/subscribe, where type-based essentially subsume both topic-based and content-based considering the expressiveness of subscriptions. In **Kilo.PS**, topic-based publish/subscribe can be modelled by creating subclasses of the Event class. An example of this is shown in figure 7.3. In this

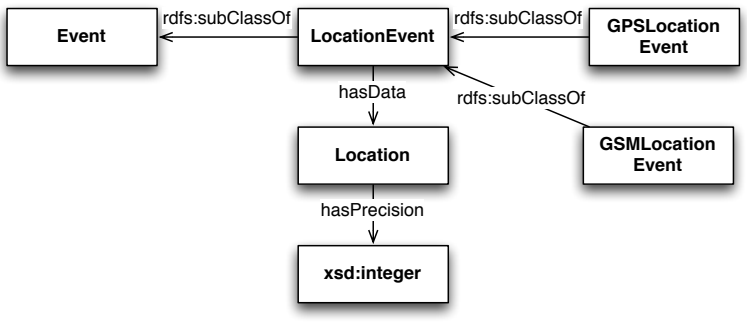

Figure 7.3: Example of a Topic Hierarchy.

case, a subscriber to LocationEvents would also receive GPSLocationEvents and

GSMLocationEvents if they were published. The following shows such a subscription:

$$LocationEvent(?e) \wedge Subscriber(c) \Rightarrow notify(c, e)$$

In content-based publish/subscribe systems, subscriptions can filter on the contents of events. An example of this is:

$$\begin{aligned} & LocationEvent(?e) \wedge Subscriber(c) \wedge hasData(e, d) \wedge \\ & hasPrecision(d, p) \wedge swrlb : lessThan(p, 100) \\ \Rightarrow \quad & notify(c, e) \end{aligned}$$

Assuming that the precision of Locations are encoded as meters, this will result in c being notified of LocationEvents where the precision is better than 100 meters. Combining the possibility for subscribing based on topics, subscribing based on data contents, and the ability to send data as objects in events essentially gives us type-based publish/subscribe. The only part which cannot be handled in ontology-based publish/subscribe is behaviour of event objects.

Furthermore, event correlation (in which, e.g., a notification is based on the sending of two events), can also be modelled using SWRL rules. Finally, in addition to traditional publish/subscribe functionality, it is also possible to, e.g., change data as well as structure for events at runtime. In the current implementation, it is up to the discipline of the user of the publish/subscribe system to ensure that standard behaviour of publish/subscribe systems is preserved.

7.1.4 Performance

The performance of **Kilo.PS** has been tested using a test bed consisting of an implementation of the broker and interfaces for publishing and subscribing. Thus, the tests are performed on the pure performance of the broker, without taking network input and output into account. Furthermore, no optimisations where implemented.

The tests where performed on a 2.93 GHz Intel Core 2 Duo iMac with 4GB 1067 MHz DDR2 memory running Mac OS X 10.6 and Java 1.6.0_15. Every test was run 100 times and the average and standard deviation calculated.

For the sake of the tests, we define the size of an ontology as the total number of elements in the ontology. That is, the total number of classes, properties, and individuals.

Of some interest, is the time it takes to initialise the system by reading in the ontology. This depends on the size of the ontology. In most cases, the intitialization time is of minor importance, but in the case of a highly dynamic system where brokers might come and go, it might be important. Figure 7.4 shows the average load times and deviation for ontologies of varying size.

The time it takes for publishing an Event is negligible, taking less than 1 ms. If the data associated of the event is large, this will increase, but will still be low. Thus, in actual use this will be bounded by the transmission time of the network, not the time spent by the broker.

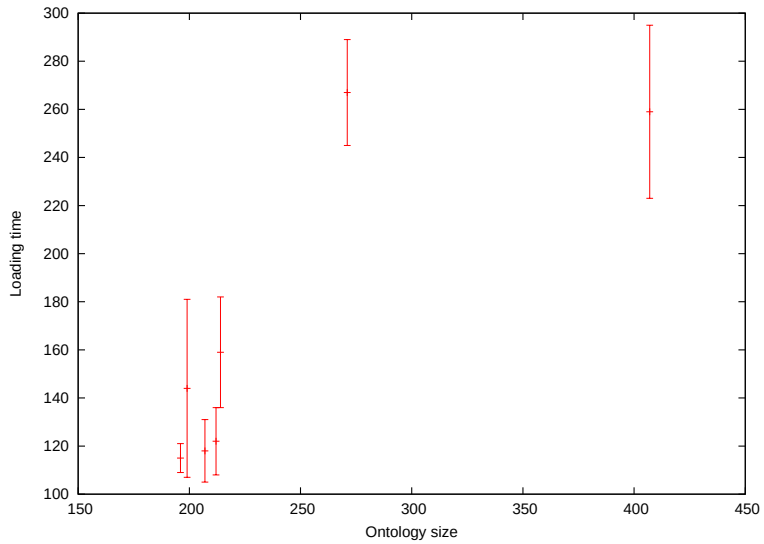

Figure 7.4: Initialisation time in ms as a function of ontology size

Of more interest is the time it takes to evaluate subscriptions. Potentially, this can depend on the size of the ontology and the number of subscriptions. It turns out that the most important of these is the number of subscriptions. While the evaluation time of subscriptions appears nearly independent of the size of the ontology (figure 7.5), the number of subscriptions influence the evaluation time heavily (figure 7.6).

Wang et al. provide performance statistics of their ontology-based publish/-subscribe system [116] based on graph-matching. Although theirs as well as our system are both based on ontologies, they a very different in model and implementation. For the purpose of performance testing, the authors use randomised events, while we use a number of tailored test-cases. The main implication is that they do not control their matching rate, which is unfortunate, since it has a high impact on the effectiveness of their algorithm, in that lowering the matching rate decreases the evaluation time significantly. However, they do report an evaluation time for 10.000 subscriptions, 10 classes, and 10 properties with a matching rate of approximately 3% as 1200 ms. Unfortunately, they do not provide an evaluation showing the effect of increasing the size of the ontology while maintaining the matching rate, resulting in tests showing that the evaluation time *decrease* as the number of classes grow. However, we feel that we can safely say that their implementation is orders of magnitude faster than ours. The reason for this is mostly that their implementation of the matching algorithm is benefits from being optimised for publish/subscribe, while our implementation is intended to be used for other, though similar purposes,

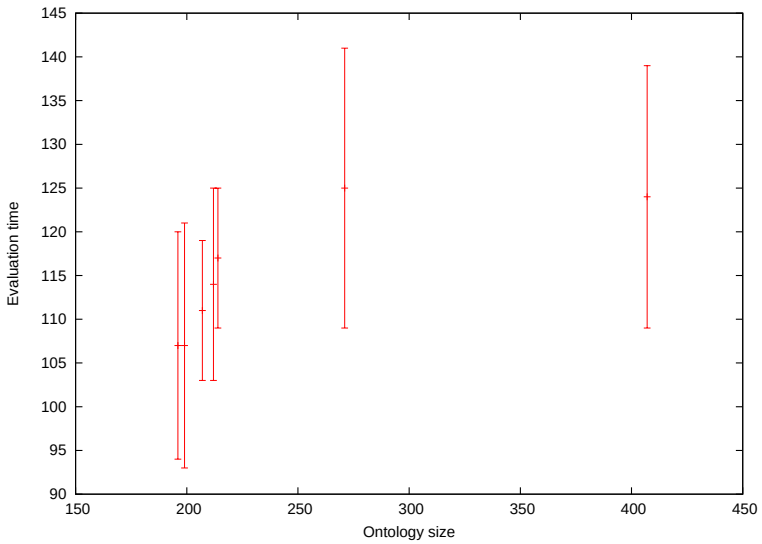

Figure 7.5: Evaluation time in ms of a single subscription as a function of the ontology size.

like accessing a context model. However, our implementation offers increased functionality in choosing how to use the system, and is not yet optimised.

While there are several different WS-Notification implementations, Humphrey et al. have provided a comparison of four different implementations [67]. For their tests, they provide a shared resource, a counter, using WS-Resource [35], and let interested parties access a notification of changes using WN-Notification. For a client and a server co-located on the same machine using no security, which is the closest we get to our setup, they report the time for notifications to arrive when the counter is updated between 8.78 ms and 46.52 ms. However, this includes not only matching subscriptions, but also marshalling and demarshalling XML, indicating that the actual matching times are somewhat smaller. However, WN-Notification is topic-based, so we expect a more effective matching algorithm, and their tests only use a single event and a single subscriber, making direct comparisons difficult.

7.1.5 Example of Use

The ideas in the publish/subscribe system have been used for implementing a system for context-sharing using an ontology-based context-model. In this case, the context-model replaces the publish/subscribe terminology of events carrying data, but the principle is the same. Consumers of context subscribe to changes in context using SWRL rules, and will receive notifications when updates matching the rules happen. This results in a highly expressive mechanism for declaring interest in spe-

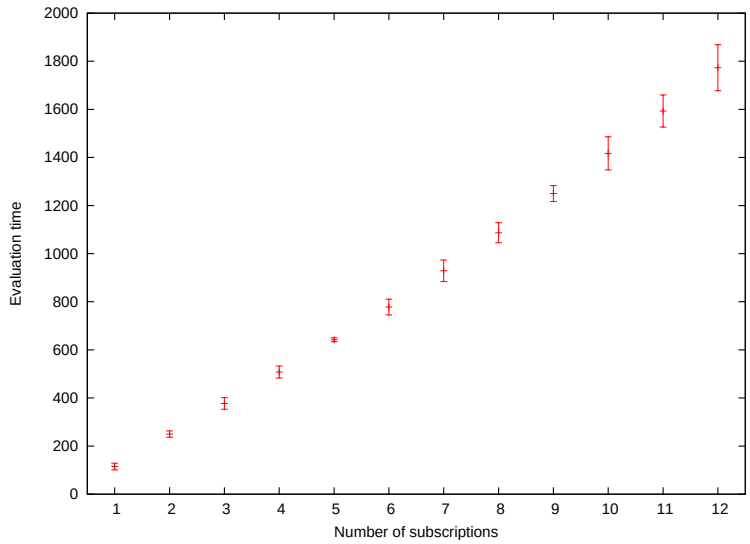

Figure 7.6: Time in ms for Evaluating subscriptions as a function of the number of subscriptions for an ontology of size 411.

cific changes and has the added value that the rules provided by consumers can themselves aggregate sensed context, providing higher-order context not already present in the model. The system is envisioned to be used in applications with a relatively small number of contextual events, where consumers of context act on behalf of users, and not for machine-to-machine interaction.

7.1.6 FUTURE WORK

One obvious area of future work is performance improvements. One way is to improve performance by optimising rules. For instance, if two subscribers ("c" and "d') both want all Events, there will be two rules in the ontology:

$$Event(?x) \wedge Subscriber(c) \Rightarrow notify(c, ?x)$$
$$Event(?x) \wedge Subscriber(d) \Rightarrow notify(d, ?x)$$

which can be rewritten as:

$$\begin{array}{ll} & Event(?x) \wedge Subscriber(c) \wedge Subscriber(d) \\ \Rightarrow & notify(c, ?x) \wedge notify(d, ?x) \end{array}$$

thereby reducing the evaluation time.

Another possibility for optimising rules is parallelisation of rule-invocation. In principle, more than one model could be maintained and rules could be distributed

among them to be evaluated in parallel. While feasible, this would imply considerable analysis of rules, since rules may themselves create new individuals in the ontology. Another possibility would be to limit rules to only notify subscribers, and not create new individuals, but this would limit the expressibility of the subscription mechanism.

Tool support in the form of integration into Protege or Eclipse, allowing for easier creation of publishers and subscribers would also be useful. This should include a simulation environment allowing for testing the results of a set of subscriptions when a series of events are produced.

7.2 Semantic Architecture Description

As described in section 6.1.1, architectural description is useful at design- and runtime. For the Kilo middleware, we use it to describe the deployment of applications, in terms of which components and connectors to load as well as for describing required and provided services of components. While several architectural description languages useful for this exist, it is interesting to investigate semantic technologies for use in such descriptions. The motivations for using ontologies of architectures and standards like OWL is the ability to use existing tools to perform transformations and reasoning on architecture models. Modelling the architecture can be supported by existing software tools, like Protege-OWL [77], and reasoners can be used, with appropriate rules, to investigate properties of the architecture.

For loading components and connectors in the Kilo middleware, we have a system where an ontology describes the initial, local setup. Our ontology is designed to plug into an existing ontology from the Hydra project for describing the architecture of applications [120]. The existing ontology is created for use with services implemented in Java as OSGi bundles, so we need to extend it to support both the Kilo component model and the fact that Kilo is implemented in C#. To support this, we added the new classes shown in figure 7.7. Besides adding Kilo services and the C# language to the ontology, we also need a new component-type, to support a property that describes that Kilo components might require other components or services.

Additional parts of the ontology, not shown in the figure, include a **kilo:Implementation** class for defining from where a service or component can be loaded. This includes defining the full name of the C# class implementing the component and the dll from which it can be loaded.

To actually describe on load-time configuration of an application, we describe the components as individuals in the ontology, as shown in figure 7.8. To actually load the configuration, a loader reads the XML serialisation of the ontology, and loads the described components. It should be noted that this particular example assumes the loader knows how to locate the components. If this is not the case, a property defining this can be added. However, the components may have requirements defined as C# properties, which the loader then examines and tries to ful-

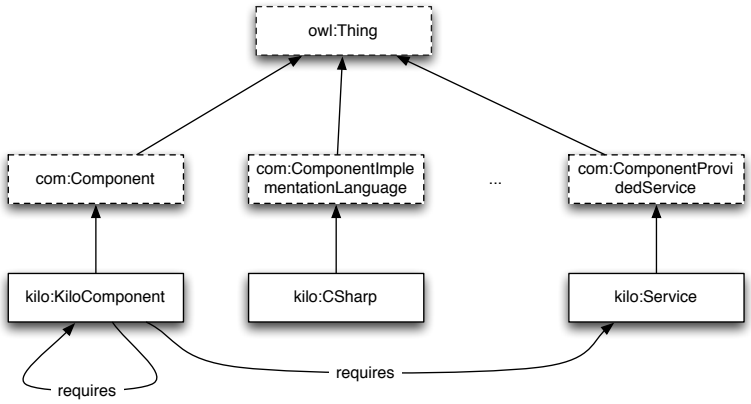

Figure 7.7: Part of the Kilo additions for the existing Hydra architecture ontology. Classes with solid borders are part of the Kilo ontology and arrows without label are rdfs:subClassOf properties which are left out for readability.

fill. As an example, the **ContextStore** component might need a **Context** service. In this case, that will be defined by a `require` property containing a URL to a *partial* OWL-S description. This describes the minimum requirements of service fulfilling the needs of the component, and it is then the role of the Kilo **ServiceDiscovery** service,to find a suitable service to fulfill these requirements. This is described in more detail in section 7.3.

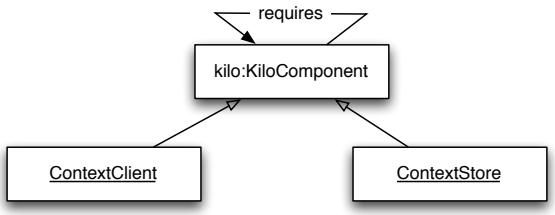

Figure 7.8: Defining a, very simple, architecture of an application using the Kilo architecture ontology. Individuals are underlined.

7.3 SEMANTIC SERVICE DISCOVERY

To use semantic descriptions for service discovery, we have analysed existing implementations of OWL-S based service matchers. For our purpose, we have used OWLS-MX [76], which is a hybrid service matcher that matches partial service descriptions with a database of service descriptions. While this implies a centralised service store, this is actually useful for our purpose, since it can be pre-loaded with descriptions of remote services.

OWLS-MX is *hybrid* in the sense that is uses both logic based and approximate semantic matching to match services. This improves upon both approaches, since

Listing 7.1: XML serialisation of a simple Kilo application consisting of two components, **ContextClient** and **ContextStore**.

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:xsp="http://www.owl-ontologies.com/2005/08/07/
       xsp.owl#"
    xmlns:com="http://localhost:9999/ontology/
       GenericComponent.owl#"
    xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
    xmlns="http://localhost:9999/ontology/kilotest.owl#"
    xmlns:kilo="http://localhost:9999/ontologies/
       kilocomponent.owl#"
    xmlns:swrl="http://www.w3.org/2003/11/swrl#"
    xmlns:protege="http://protege.stanford.edu/plugins/
       owl/protege#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns
       #"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
  xml:base="http://localhost:9999/ontology/kilotest.owl">
  <owl:Ontology rdf:about=""/>
  <owl:AllDifferent>
    <owl:distinctMembers rdf:parseType="Collection">
      <kilo:KiloComponent rdf:ID="Client">
        <com:componentName rdf:datatype="http://www.w3.
           org/2001/XMLSchema#string"
        >ContextClient</com:componentName>
      </kilo:KiloComponent>
      <kilo:KiloComponent rdf:ID="Store">
        <com:componentName rdf:datatype="http://www.w3.
           org/2001/XMLSchema#string"
        >ContextStore</com:componentName>
      </kilo:KiloComponent>
    </owl:distinctMembers>
  </owl:AllDifferent>
</rdf:RDF>
```

Listing 7.2: The interface of a simple **echo** service.

```
public interface EchoService {
    public String echo(String echo);
}
```

neither of the methods always succeed in finding a match. Logic based matching can use a number of different matching strategies including exact matching and subsumed match, i.e. output of the service is more than requested. The approximate service matching is based on a nearest neighbour approach, where a matching service may require more input than requested, and provide more output than requested. OWLS-MX first tries logic matching and, if it fails, tries nearest-neighbour. For logic matching, it has 4 different matching filters, which are applied in order of most-exact match. The approximate match is the least exact. That is, OWLS-MX will always return the most exact match to a request. As an example of service matching, we look at a very simple setup involving an **echo** service, with a simple interface as shown in listing 7.2. To register the service with the Kilo discovery service, the device offering the service will first have to find the discovery service using Bonjour [68]. This returns a URL specifying the location of the service, which may then be invoked. In turn, the client can discover the necessary service by sending a partial profile of the service it needs to the discovery service, which will then reply with a URI pointing to the service if it is available. This bootstrapping process is shown in figure 7.9. For exact matching, the partial profile could contain a description of an operation that takes a string as input, and returns the same string.

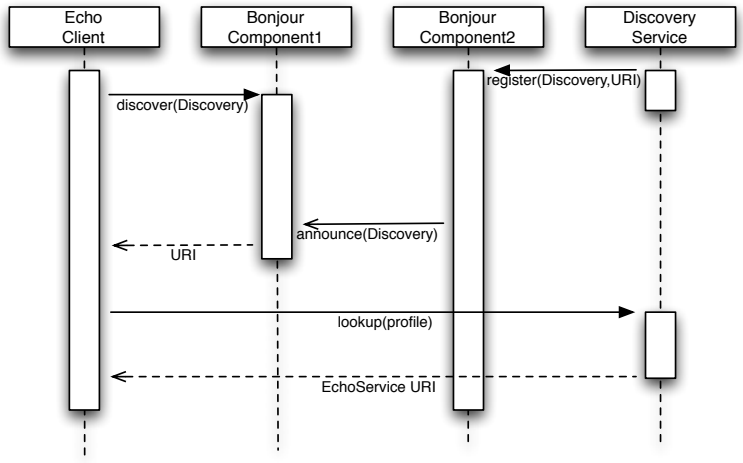

Figure 7.9: Sequence diagram showing the registration process of the Echo service.

As described in section 2.3, a profile consists of four parts:

- a service description (partly shown in listing 7.3),

- a profile,
- a grounding,
- and a model.

The complete description is included in appendix B.3. The client can send either a complete or a partial profile to the **Discovery** service and will in turn get a URL pointing to a service that corresponds to the required service.

Using semantic service matching for allows the client of the discovery process to focus on exactly what is needed from the service, and not a specific service type with a pre-known interface. For instance, the partial profile provided by the client could define a needed operation, and will then match all services that provide that particular operation. It also allows the client to simply define what data it needs, and that what discover all services that can provide the data.

7.4 Summary

We have used semantic web technology for a variety of purposes. Describing web services using semantic descriptions allows for new ways of describing services, which in turn allows for service matching based on semantic matching. Furthermore, ontologies can be used for implementing publish/subscribe allowing for a very expressive subscription language, in our case using by using SWRL. Furthermore, the same technology can be used for implementing context models, allowing for related technologies to be used for service matching, messaging, and context.

Listing 7.3: OWL-S service description of the Echo Service.

```
<rdf:RDF ... >
  <owl:Ontology rdf:about="">
    <owl:versionInfo>
      $Id: EchoService.owl,v 1.0 2009/01/14 argo $
    </owl:versionInfo>
    <rdfs:comment>
      ...
    </rdfs:comment>
    <owl:imports rdf:resource="&service;" />
      ...
    </owl:Ontology>

  <service:Service rdf:ID="EchoService">
    <service:presents rdf:resource="&echo_profile;#
       EchoServiceProfile"/>
    <service:describedBy rdf:resource="&echo_process;#
       EchoServiceProcess"/>
    <service:supports rdf:resource="&echo_grounding;#
       EchoServiceGrounding"/>
  </service:Service>

  <profile:Profile rdf:about="&echo_profile;#
     Profile_EchoService">
    <service:presentedBy rdf:resource="#EchoService"/>
  </profile:Profile>

  <process:AtomicProcess rdf:about="&echo_process;#
     EchoServiceEcho">
    <service:describes rdf:resource="#EchoService"/>
  </process:AtomicProcess>

  <grounding:WsdlGrounding rdf:about="&echo_grounding;#
     EchoServiceGrounding">
    <service:supportedBy rdf:resource="#EchoService"/>
  </grounding:WsdlGrounding>
</rdf:RDF>
```

Chapter 8

Summary and Conclusion

This thesis has described work carried out in the Kilo project in regard to context-awareness and middleware. The goal of the middleware was to suport integration of information from heterogeneous sources and support filtering of the information in regard to the current context of the user. The main body of the work concerned designing and evaluating the middleware using techniques from participatory design.

With insight into the domain and requirements originating in the participatory design process, a number of additional research areas where identified, and work where carrried out in each of the research areas.

For participatory design, we explored the use of a number of techniques for the development of middleware. Of these, field studies and interview proved useful for determining requirements of the middleware, although the requirements were usually expressed indirectly, for instance as requirements of an application supporting a scenario discussed during an interview. During field studies, when discovering how workers carried out their daily tasks, we also analysed the existing systems used at the farms, and some requirements arose from the need to potentially support interaction with these systems.

For evaluation, deployment of prototypes in actual work environments provided feedback about the prototype application as well as the middleware. Again, end-users of software cannot be expected to explain problems or shortcomings of the middleware, but can explain what they have experienced. It is then up to the developer to "translate" these observations into implications for the further development of the middleware.

To goal of the project was the development of a middleware design for integration and context awareness in agriculture. As such, the results in middleware are of importance. In this area, we have mostly looked into using semantic web technologies in middleware for service oriented systems. Semantic descriptions have been used for the context model, the service model, architecture description, and publish/subscribe.

8.1 RESULTS IN RESEARCH AREAS

As described in section 1.1, the research has focused on three areas of research:

- Software Architecture,
- context Awareness, and
- integration.

These research areas were analysed indirectly, by covering five different research subjects:

- Participatory design,
- context awareness,
- architectural models,
- first class connectors, and
- semantic technology.

A number of tasks where carried out in relation to each research subject. Participatory design techniques where used for designing and evaluating systems. For context, we analysed what kind of data a context model should support, and designed a context model. Architectural models have been used in middleware, by using architectural descriptions of applications to provide a declarative method for describing requirements between components. To implement integration of heterogeneous services, we employed first class connectors, which where also part of the declarative architectural description. Finally, we used semantic technology for describing the architectural model, as well as integration in the form of semantic annotations of services used for service discovery, and created a semantic context model, which used tools for querying ontologies for implementing subscriptions.

If we look at the research areas, for software architecture, we have used the process of designing and architecture for middleware for exploring the use of ontologies the architecture of applications built with the middleware. For context awareness, we have analysed how it can be useful in the agriculture domain and created a model and methods for querying the model. Finally, for integration, we have explored the use of semantic annotations and first class connectors for providing integration of heterogeneous services and devices.

8.2 RESEARCH QUESTIONS

In chapter 1, we described four research questions as the basis of the work in this dissertation. Restated, the research questions are:

Research Question 1 *How well do techniques from participatory design work for designing context aware middleware?*

Research Question 2 *Is a reified software architecture, including first class connectors, useful for deployment and integration for context aware middleware?*

Research Question 3 *Can the use of applications developed for a specific domain be used to design a general purpose middleware?*

Research Question 4 *Are semantic web technologies suitable for describing context, services, and architecture?*

For research question (1), we applied various techniques from participatory design for gathering requirements and evaluating results. Field studies, interviews, and focus groups proved successful in obtaining requirements and evaluating early designs, and the result was a viable middleware design. An important part is choosing the right artifact for each activity, but this is to be expected, as this is also the case for participatory design for applications. Our conclusion is, that participatory design techniques lend themselves well to the design of middleware, but have the same pitfalls as participatory design for end-users applications. Mainly, most participatory design techniques are qualitative, and requires the developers to ensure that the results supports generalisation.

For question (2), we have explored the use of first class connectors for prototyping service oriented systems, and found that they provide a good abstraction which allows for experimenting with service-enabling different components in an architecture without changing the components, since the communication is encapsulated in the connectors. Furthermore, our use of first class connectors in the middleware not only encapsulates communication, but allows for defining the architecture of the running systems as a component and connector view and supports defining the requirements of components in terms of abstract service descriptions which can then be accessed though connectors. Potentially they can also perform necessary translation and adaptation needed for interacting with the service without the involvement of the components. Thus, they provide a useful abstraction for service oriented architectures, and the reified software architecture, in terms of deployment descriptions and required and provided services provides flexibility in choosing suitable services through semantic service matching.

Question (3) is concerned with generalising development for a specific domain to general purpose middleware. We have not fully verified this, since that would require implementation of applications from other domains on top of the middleware. However, we argue that for most parts, the middleware will generalise to other domains. While our context model, defined as an ontology in OWL, clearly has some domain specific parts, this does not preclude using the context service with other models. Furthermore, the ontology model uses parts of an existing context model from the Hydra project which is itself used in more than one domain. This is an example of structuring the context model so that parts of it are general purpose, with

domain specific parts being added when needed. Other services, like the administration service, is clearly domain specific and is, in fact, an existing service used in Danish agriculture. As such, the middleware resembles the model by Schmidt [106], in that the middleware consists of common middleware services as well as domain specific middleware services.

For the use of semantic web technology, as expressed by question (4), we have explored the use of OWL-S for service description and discovery and ontologies written in OWL for modelling context and deployment architecture. Our conclusion is that the use of related technologies provides benefits to the middleware as well as the developer, although the last conclusion would require more research to fully verify. Since OWL-S describes input and output in terms of concepts in an ontology, we can potentially use an ontology for describing this as part of a context ontology, thus increasing the value of creating the ontology. For software architecture, the use of an OWL ontology for describing the architecture means that developers can use the same tools, e.g. Protege-OWL, for describing context, data, and architecture. Finally, using the same mechanism for publish/subscribe and accessing the context models means that data can be shared using the same ontological description as used for context and services.

8.3 HYPOTHESISES

The hypothesises of the project are:

Hypothesis 1 *Participatory design provides useful techniques for designing and evaluating middleware.*

Hypothesis 2 *A service oriented middleware using first class connectors provides abstractions that are useful for the implementation of applications that access information and services from heterogeneous administrative domains.*

Hypothesis 3 *Semantic web technologies can be used for a variety of purposes in a context aware middleware, including architecture description, integration, and context modelling.*

To evaluate hypothesis (1) we have used participatory design techniques throughout the project with good results. The techniques proved useful for the design process but have a some pitfalls, noticeably that they are mostly qualitative by nature. This means that the designers will have to ensure that they are still designing general purpose middleware, while supporting the requirements gathered in the participatory design process. Furthermore, participatory design techniques work well with developers as participants when the right artifacts are chosen.

For hypothesis (2) we have built applications using a service oriented middleware that interacts with web services using first class connectors. The use of web services provides us with suitability for heterogeneous administrative domains, and

the use of first class connectors provide us with useful abstractions for building the applications. To fully evaluate the usefulness though, it would be necessary to have external developers use the middleware to create applications.

We have shown that semantic web technologies can be used for a variety of purposes, thus validating hypothesis (3). We have used ontologies and XML Schema for describing the deployment architecture of applications. For describing service interfaces and service matching, we have used OWL-S and OWL-S based semantic matching, the latter of which is used for service discovery. Finally, we have used OWL and SWRL for describing context and for publish/subscribe.

8.4 Future Work

One of the areas left out of the project to some degree was direct involvement of developers. While this was compensated by using focus groups ideally, to determine the usability of the middleware towards developers, field studies of developers using the middleware to create applications should be carried out. This should be followed up by evaluation of the resulting applications in terms of development time and suitability.

The second application prototype was not completed, and therefore not evaluated by real-world deployment. Therefore, the usefulness of the final middleware design is partly theoretical. A future evaluation would involve not only deploying an application using the middleware, but deploying it on more than on farm, to perform a quantitative evaluation.

While our use of ontologies for describing context is promising for the domain, the problem of clearly defining concepts in the domain remains. Conflicting understanding of concepts in the domain remain a problem that has to be solved to fully benefit from a context model, as well as service oriented integration. This is a problem that can only be solved by experts from different sub-domains working together to create a common understanding of the concepts involved.

We have analysed different semantic web technologies for different purposes, and shown how they can be used, but there remain a number of outstanding research questions. In the case of our publish/subscribe mechanism, optimisations would make it more useful, since the current implementation is relatively slow. For the context model we have used it as a centralised model, but the possibility of a distributed OWL based context model seems an interesting direction of research. This would expand the range of domains where the context model could be used.

8.5 Conclusion

This dissertation has provided on overview of the parts of the Kilo project carried out at Aarhus University. The focus of the project was the development of a context aware middleware for service oriented integration in agriculture. After two iterations, we have designed a middleware using semantic web technologies through-

out the middleware. This has the benefit that the technologies are related resulting in a less steep learning curve, and tie well into existing infrastructure based on web services. For instance, our service model is based on OWL-S, which in turn use WSDLfor describing actual interfaces, providing us with the benefit that WSDLinterfaces are already available for the existing web services. Similarly, our use of OWL to describe context means that concepts defined in the context ontology can also be used for describing the data of input and output of web services, potentially allowing for applications to simply require services that can provide them with the needed context. This is promising for using the middleware for context awareness and service oriented integration in both the examined domain and other domains.

Appendix A

Requirements

This appendix provides some of the initial requirements gathered in the beginning of the project for the first prototype, although some extend to the second prototype. This is intended to give examples of requirement, and does not represent requirements that where eventually incorporated into the project.

Requirement 1	
name:	Disconnected operation
description:	The middleware must not expect continuous connectivity, and should work even though the connection to central systems is broken at times.
Priority:	High
Origin:	Field studies for case 1.

Requirement 2	
name:	Use knowledge of data coverage
description:	The middleware should take advantage of previous measurements of coverage in its caching strategy, and deliver high-priority data and retrieve necessary information before a device is moved into an area that is known to have limited or no mobile data coverage.
Priority:	Low
Origin:	Internal discussions at DAIMI

Requirement 3	
name:	Integration with existing back-end systems
description:	The middleware must support integration with existing middleware systems, to enable transmission of data either way.
Priority:	High
Origin:	Talks with engineers at LC.

Requirement 4	
name:	Location awareness
description:	The system must be able to determine the location of a device carried by an individual.
Priority:	High
Origin:	Field Studies at Skovgård and Nygård

Requirement 5	
name:	Partial Task Identification
description:	The system should be able to partially identify the current task of the user. If the system is unable to determine the task, a number of likely possibilities should be presented to the user.
Priority:	High
Origin:	Case 2, developed at LC

Requirement 6	
name:	Implementation language
description:	As much as possible of the implementation should be done in C#, to support existing developers at LC.
Priority:	High
Origin:	LC current practice.

Requirement 7	
name:	Service Discovery
description:	The middleware must support discovery of available services. Service discovery should be fine-grained, in the sense that applications should be able to ask for a specific service and QoS, and not for a particular device.
Priority:	High
Origin:	Research goal

Requirement 8	
name:	Representation of Context
description:	Context information is stored and queried using ontologies.
Priority:	High
Origin:	Design decision

Requirement 9	
name:	Situation awareness
description:	The middleware must be situation-aware. I.e. instead of merely determining the context, it must determine what *situation* the user is in, e.g. using a tractor.
Priority:	High
Origin:	DAIMI internal. Necessity to determine e.g. current task.

Appendix B

Code

This appendix provides the source of the XSD configuration schema for **Kilo.Two** applications (section B.1) mentioned in section 4.2.2, the WSDLinterface of the **Context** service (section B.2) from section 5.6.1, and the full OWL-S profile of the **Echo** service (section B.3) used in section 7.3.

B.1 Configuration Schema

Listing B.1: "XSD Schema for Kilo.Two Component Configuration."

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="Configuration"
    targetNamespace="http://kilo.daimi.au.dk/schemas/
       Configuration.xsd"
    elementFormDefault="qualified"
    xmlns="http://www.daimi.au.dk/~kilo/schemas/
       Configuration.xsd"
    xmlns:mstns="http://tempuri.org/Configuration.xsd"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
>
  <xs:annotation>
    <xs:documentation>
      XML Schema describing the format of configuration
         files for the Kilo component model.

      Author: Kristian Ellebaek Kjaer  -- argo@cs.au.dk
    </xs:documentation>
  </xs:annotation>

  <xs:element name="Configuration">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Component" minOccurs="0"
           maxOccurs="unbounded"/>
```

```
      <xs:element ref="Connector" minOccurs="0"
         maxOccurs="unbounded"/>
      <xs:element ref="Architecture" minOccurs="0"
         maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="Component">
  <xs:complexType>
    <xs:sequence>
      <!-- Name of the component-->
      <xs:element name="name" type="xs:string"
         minOccurs="1" maxOccurs="1"/>
      <!-- Assembly where component can be found -->
      <xs:element name="assembly" type="xs:string"
         minOccurs="0" maxOccurs="1"/>
      <!-- How many should be instantiated -->
      <xs:element name="cardinality" type="xs:integer"
         minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="Connector">
  <xs:complexType>
    <xs:sequence>
      <!-- Name of the component-->
      <xs:element name="name" type="xs:string"
         minOccurs="1" maxOccurs="1"/>
      <!-- Assembly where connector can be found. If
         not present, the assembly must already be
         linked -->
      <xs:element name="assembly" type="xs:string"
         minOccurs="0" maxOccurs="1"/>
      <!-- The class implementing the connector -->
      <xs:element name="class" type="xs:string"
         minOccurs="1" maxOccurs="1"/>ss
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="Connection">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="component" type="xs:anyURI"
         minOccurs="2" maxOccurs="unbounded"/>
      <xs:element name="connector" type="xs:anyURI"
         minOccurs="1" maxOccurs="1"/>
```

```
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Architecture">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Connection" minOccurs="1"
           maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

B.2 Context Service WSDL interface

Listing B.2: WSDL Interface of the **Context** service

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:context="http://kilo.daimi.au.dk/
   context/" xmlns:soap="http://schemas.xmlsoap.org/wsdl
   /soap/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="
   contextservice" targetNamespace="http://kilo.daimi.au
   .dk/context/">
  <wsdl:types>
    <xsd:schema targetNamespace="http://kilo.daimi.au.dk/
       context/">
      <xsd:element name="Publish">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="event" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="Subscribe">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="subscriber" type="
                   xsd:string"/>
            <xsd:element name="subscription" type="
               xsd:string"/>
            </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="PublishRequest">
```

```
    <wsdl:part element="context:Publish" name="parameters
       "/>
  </wsdl:message>
  <wsdl:message name="SubscribeRequest">
    <wsdl:part name="parameters" element="
       context:Subscribe"></wsdl:part>
  </wsdl:message>
  <wsdl:portType name="contextservice">
    <wsdl:operation name="Publish">
      <wsdl:input message="context:PublishRequest"/>
    </wsdl:operation>
    <wsdl:operation name="Subscribe">
        <wsdl:input message="context:SubscribeRequest"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="contextservice" type="
     context:contextservice">
    <soap:binding style="document" transport="http://
       schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="Publish">
      <soap:operation soapAction="http://kilo.daimi.au.dk
         /context/publish"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
    </wsdl:operation>
    <soap:binding style="document" transport="http://
       schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="Subscribe">
      <soap:operation soapAction="http://kilo.daimi.au.dk
         /context/subscribe"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="context">
    <wsdl:port binding="context:contextservice" name="
       contextPort">
      <soap:address location="http://kilo.daimi.au.dk/
         context/"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

B.3 ECHO SERVICE OWL-S PROFILE

Listing B.3: **Echo** service OWL-S description.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE rdf:RDF [
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-
     ns">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
  <!ENTITY owl "http://www.w3.org/2002/07/owl">
  <!ENTITY service "http://www.daml.org/services/owl-s
     /1.1/Service.owl">
  <!ENTITY profile "http://www.daml.org/services/owl-s
     /1.1/Profile.owl">
  <!ENTITY process "http://www.daml.org/services/owl-s
     /1.1/Process.owl">
  <!ENTITY grounding "http://www.daml.org/services/owl-s
     /1.1/Grounding.owl">
  <!ENTITY echo_profile "http://www.daimi.au.dk/~kilo/owl
     -s/EchoServiceProfile.owl">
  <!ENTITY echo_process "http://www.daimi.au.dk/~kilo/owl
     -s/EchoServiceProcess.owl">
  <!ENTITY echo_grounding "http://www.daimi.au.dk/~kilo/
     owl-s/EchoServiceGrounding.owl">
  <!ENTITY DEFAULT "http://www.daimi.au.dk/~kilo/owl-s/
     EchoService.owl">
]>

<rdf:RDF
  xmlns:rdf = "&rdf;#"
  xmlns:rdfs    ="&rdfs;#"
  xmlns:owl = "&owl;#"
  xmlns:service= "&service;#"
  xmlns:profile= "&profile;#"
  xmlns:process= "&process;#"
  xmlns:grounding= "&grounding;#"
  xmlns         ="&DEFAULT;#"
 >

     <owl:Ontology rdf:about="">
       <owl:versionInfo>
         $Id: EchoService.owl,v 1.0 2009/01/14 argo $
       </owl:versionInfo>
       <rdfs:comment>
         This ontology represents the OWL-S service
            description for the
         EchoService web service example.
       </rdfs:comment>
       <owl:imports rdf:resource="&service;" />
       <owl:imports rdf:resource="&profile;" />
       <owl:imports rdf:resource="&process;" />
       <owl:imports rdf:resource="&grounding;" />
```

```
      <owl:imports rdf:resource="&echo_profile;" />
      <owl:imports rdf:resource="&echo_process;" />
      <owl:imports rdf:resource="&echo_grounding;" />
    </owl:Ontology>

    <service:Service rdf:ID="EchoService">
        <service:presents rdf:resource="&echo_profile;#
           EchoServiceProfile"/>
        <service:describedBy rdf:resource="&echo_process
           ;#EchoServiceProcess"/>
        <service:supports rdf:resource="&echo_grounding;#
           EchoServiceGrounding"/>
    </service:Service>

    <profile:Profile rdf:about="&echo_profile;#
       Profile_EchoService">
        <service:presentedBy rdf:resource="#EchoService"/
           >
    </profile:Profile>

    <process:AtomicProcess rdf:about="&echo_process;#
       EchoServiceEcho">
        <service:describes rdf:resource="#EchoService"/>
    </process:AtomicProcess>

    <grounding:WsdlGrounding rdf:about="&echo_grounding;#
       EchoServiceGrounding">
        <service:supportedBy rdf:resource="#EchoService"/
           >
    </grounding:WsdlGrounding>
</rdf:RDF>
```

Listing B.4: **Echo** service OWL-S profile.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-
     ns">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
  <!ENTITY owl "http://www.w3.org/2002/07/owl">
  <!ENTITY process "http://www.daml.org/services/owl-s
     /1.1/Process.owl">
  <!ENTITY service "http://www.daml.org/services/owl-s
     /1.1/Service.owl">
  <!ENTITY actor "http://www.daml.org/services/owl-s/1.1/
     ActorDefault.owl">
  <!ENTITY profile "http://www.daml.org/services/owl-s
     /1.1/Profile.owl">
  <!ENTITY profileHierarchy "http://www.daml.org/services
     /owl-s/1.1/ProfileHierarchy.owl">
```

```
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY echo_process "http://www.daimi.au.dk/~kilo/owl
     -s/EchoServiceProcess.owl">
  <!ENTITY DEFAULT "http://www.daimi.au.dk/~kilo/owl-s/
     EchoService.owl">
]>

<rdf:RDF
  xmlns:rdf=     "&rdf;#"
  xmlns:rdfs=    "&rdfs;#"
  xmlns:owl = "&owl;#"
  xmlns:actor= "&actor;#"
  xmlns:service= "&service;#"
  xmlns:process= "&process;#"
  xmlns:profile= "&profile;#"
  xmlns:profileHierarchy= "&profileHierarchy;#"
  xmlns:xsd=     "&xsd;#"
  xmlns=         "&DEFAULT;#">

  <owl:Ontology rdf:about="">
    <owl:imports rdf:resource="&service;" />
    <owl:imports rdf:resource="&process;" />
    <owl:imports rdf:resource="&profile;" />
    <owl:imports rdf:resource="&profileHierarchy;" />
    <owl:imports rdf:resource="&echo_process;"/>
  </owl:Ontology>

  <profile:Profile
    rdf:ID="EchoServiceProfile">

    <service:presentedBy rdf:resource="&service;#
       EchoService"/>
    <profile:has_process rdf:resource="&process;#
       EchoServiceEcho"/>

    <profile:serviceName>EchoService</profile:serviceName
       >
    <profile:textDescription>
      Sample Echo Service for testing
    </profile:textDescription>

    <profile:contactInformation>
        <actor:Actor rdf:ID="EchoContacts">
            <actor:webURL>
                http://www.daimi.au.dk/kilo/owl-s/
                   EchoService.html
            </actor:webURL>
        </actor:Actor>
    </profile:contactInformation>
```

```
    <!-- Add input/preconditions/effects here -->
    <profile:hasOutput rdf:resource="&echo_process;#
       EchoServiceEchoMessageOutput"/>
    <profile:hasInput rdf:resource="&echo_process;#
       EchoServiceEchoMessage"/>
  </profile:Profile>

</rdf:RDF>
```

Listing B.5: **Echo** service OWL-S grounding.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE rdf:RDF [
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-
     ns">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
  <!ENTITY owl "http://www.w3.org/2002/07/owl">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY service "http://www.daml.org/services/owl-s
     /1.1/Service.owl">
  <!ENTITY process "http://www.daml.org/services/owl-s
     /1.1/Process.owl">
  <!ENTITY grounding "http://www.daml.org/services/owl-s
     /1.1/Grounding.owl">
  <!ENTITY echo_service "http://www.daimi.au.dk/~kilo/owl
     -s/EchoService.owl">
  <!ENTITY echo_profile "http://www.daimi.au.dk/~kilo/owl
     -s/EchoServiceProfile.owl">
  <!ENTITY echo_process "http://www.daimi.au.dk/~kilo/owl
     -s/EchoServiceProcess.owl">
  <!ENTITY echo_grounding "http://www.daimi.au.dk/~kilo/
     owl-s/EchoServiceGrounding.owl">
  <!ENTITY echo_wsdl_grounding "http://www.daimi.au.dk/~
     kilo/owl-s/echoservice.wsdl">
  <!ENTITY DEFAULT "http://www.daimi.au.dk/~kilo/owl-s/
     EchoServiceGrounding.owl">
]>

<rdf:RDF
  xmlns:rdf = "&rdf;#"
  xmlns:rdfs ="&rdfs;#"
  xmlns:owl = "&owl;#"
  xmlns:xsd ="&xsd;#"
  xmlns:service  = "&service;#"
  xmlns:grounding= "&grounding;#"
  xmlns:echo_service= "&echo_service;#"
  xmlns="&DEFAULT;#"
  >
```

```
<owl:Ontology rdf:about="">
  <owl:versionInfo>
    $Id: EchoServiceGrounding.owl,v 1.0 2009/01/14 argo
       Exp $
  </owl:versionInfo>
  <rdfs:comment>
    This ontology represents the OWL-S service grounding
       for the
    EchoService web service example.
  </rdfs:comment>
  <owl:imports rdf:resource="&service;"/>
  <owl:imports rdf:resource="&process;"/>
  <owl:imports rdf:resource="&grounding;"/>
  <owl:imports rdf:resource="&echo_process;"/>
  <owl:imports rdf:resource="&echo_service;"/>
</owl:Ontology>

<grounding:WsdlGrounding rdf:ID="EchoServiceGrounding">
    <service:supportedBy rdf:resource="&echo_service;#
       EchoService"/>
    <rdfs:comment>
        This service has a single atomic process
    </rdfs:comment>
    <grounding:hasAtomicProcessGrounding rdf:resource="#
       EchoGrounding"/>
</grounding:WsdlGrounding>

<grounding:WsdlAtomicProcessGrounding rdf:ID="
   EchoServiceGroundingProcess">
    <grounding:owlsProcess rdf:resource="&echo_process;
       EchoServiceEcho"/>
    <grounding:wsdlOperation>
        <grounding:WsdlOperationRef>
            <grounding:portType>
                <xsd:anyURI rdf:value="&
                   echo_wsdl_grounding;#EchoPort"/>
            </grounding:portType>
        </grounding:WsdlOperationRef>
    </grounding:wsdlOperation>

    <grounding:wsdlInputMessage>
        <xsd:anyURI rdf:value="&echo_wsdl_grounding;#
           echoRequest"/>
    </grounding:wsdlInputMessage>

    <grounding:wsdlInputs rdf:parseType="Collection">
        <grounding:wsdlInputMessageMap>
            <grounding:owlsParameter
```

```
                rdf:resource="&echo_process;#
                    EchoServiceInput"/>
            <grounding:wsdlMessagePart>
                <xsd:anyURI rdf:value="&
                    echo_wsdl_grounding;#echo"/>
            </grounding:wsdlMessagePart>
        </grounding:wsdlInputMessageMap>
    </grounding:wsdlInputs>

    <grounding:wsdlOutputMessage>
        <xsd:anyURI rdf:value="&echo_wsdl_grounding;#
            echoResponse"/>
    </grounding:wsdlOutputMessage>

    <grounding:wsdlOutputs rdf:parseType="Collection">
        <grounding:WsdlOutputMessageMap>
            <grounding:owlsParameter
                rdf:resource="&echo_process;#
                    EchoServiceOutput"/>
            <grounding:wsdlMessagePart>
                <xsd:anyURI rdf:value="&
                    echo_wsdl_grounding;#result"/>
            </grounding:wsdlMessagePart>
        </grounding:WsdlOutputMessageMap>
    </grounding:wsdlOutputs>

    <grounding:wsdlReference>
        <xsd:anyURI rdf:value="http://www.w3.org/TR/2001/
            NOTE-wsdl-20010315"/>
    </grounding:wsdlReference>
    <grounding:otherReference>
        <xsd:anyURI rdf:value="http://www.w3.org/TR/2001/
            NOTE-wsdl-20010315"/>
    </grounding:otherReference>
    <grounding:otherReference>
        <xsd:anyURI rdf:value="http://schemas.xmlsoap.org
            /wsdl/soap/"/>
    </grounding:otherReference>
    <grounding:otherReference>
        <xsd:anyURI rdf:value="http://schemas.xmlsoap.org
            /soap/http/"/>
    </grounding:otherReference>
    <grounding:wsdlDocument>
        <xsd:anyURI rdf:value="&echo_wsdl_grounding;"/>
    </grounding:wsdlDocument>

</grounding:WsdlAtomicProcessGrounding>
</rdf:RDF>
```

Listing B.6: **Echo** service OWL-S process.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE rdf:RDF [
  <!ENTITY rdf     "http://www.w3.org/1999/02/22-rdf-
     syntax-ns">
  <!ENTITY rdfs    "http://www.w3.org/2000/01/rdf-schema"
     >
  <!ENTITY xsd     "http://www.w3.org/2001/XMLSchema">
  <!ENTITY owl     "http://www.w3.org/2002/07/owl">
  <!ENTITY service "http://www.daml.org/services/owl-s
     /1.1/Service.owl">
  <!ENTITY process "http://www.daml.org/services/owl-s
     /1.1/Process.owl">
  <!ENTITY echo   "http://www.daimi.au.dk/~kilo/owl-s/
     EchoService.owl">
  <!ENTITY DEFAULT "http://www.daimi.au.dk/~kilo/owl-s/
     EchoServiceProcess.owl">
  <!ENTITY THIS     "http://www.daimi.au.dk/~kilo/owl-s/
     EchoServiceProcess.owl">
]>


<rdf:RDF
  xmlns:rdf=     "&rdf;#"
  xmlns:rdfs=    "&rdfs;#"
  xmlns:xsd =     "&xsd;#"
  xmlns:owl = "&owl;#"
  xmlns:service = "&service;#"
  xmlns:process = "&process;#"
  xmlns =         "&DEFAULT;#"
>

<owl:Ontology rdf:about="">
  <owl:imports rdf:resource="&service;"/>
  <owl:imports rdf:resource="&process;"/>
</owl:Ontology>

<!-- Basic data types -->
<owl:Class rdf:ID="Message"/>

<owl:DatatypeProperty rdf:ID="messageContent">
    <rdfs:domain rdf:resource="#Message"/>
    <rdfs:range rdf:resource="&xsd;#string"/>
</owl:DatatypeProperty>

<!-- ProcessModel instance -->
<process:ProcessModel rdf:ID="EchoServiceProcessModel">
    <service:describes rdf:resource="&echo;#EchoService"/
       >
```

```
    <process:hasProcess rdf:resource="#EchoServiceEcho"/>
</process:ProcessModel>

<!-- The Atomic process EchoServiceEcho
     There are no preconditions or effects - only output.
    -->
<process:AtomicProcess rdf:ID="EchoServiceEcho">

    <process:hasOutput>
        <process:Output rdf:ID="
           EchoServiceEchoMessageOutput">
            <process:parameterType rdf:datatype="&xsd;#
               anyURI">&xsd;#string</
               process:parameterType>
        </process:Output>
    </process:hasOutput>

    <process:hasInput>
        <process:Input rdf:ID="EchoServiceEchoMessage">
            <process:parameterType rdf:datatype="&xsd;#
               anyURI">&xsd;#string</
               process:parameterType>
        </process:Input>
    </process:hasInput>

</process:AtomicProcess>

</rdf:RDF>
```

Listing B.7: **Echo** service OWL-S WSDL file.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/
       encoding/"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    xmlns:echo="http://ellebaek.net"
    targetNamespace="http://ellebaek.net">

    <message name="echoRequest">
        <part name="echo" type="xs:string"/>
    </message>
    <message name="echoResponse">
        <part name="result" type="xs:string"/>
    </message>

    <portType name="EchoPort">
```

```
        <operation name="echo">
            <input message="echo:echoRequest"/>
            <output message="echo:echoResponse"/>
        </operation>
    </portType>

    <binding name="EchoBinding" type="echo:EchoPort">
        <soap:binding style="rpc"
            transport="http://schemas.xmlsoap.org/soap/
               http"/>
        <operation name="echo">
            <soap:operation soapAction="http://ellebaek.
               net/echo/echo" style="rpc"/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>

    <service name="EchoService">
        <port name="EchoPort" binding="echo:EchoBinding">
            <soap:address location="http://ellebaek.net/
               echo"/>
        </port>
    </service>
</definitions>
```

Appendix C

GUI Mockups

This appendix provides the original mockups used as artifacts for participatory design (chapter 3). Due to the participants of the projects, the language of the mockups is Danish.

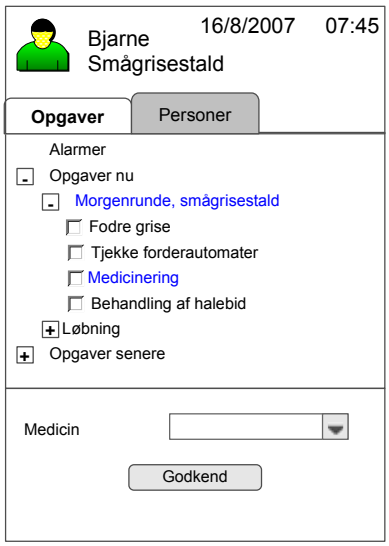

Figure C.1: GUI mockup used in initial design process (1).

Figure C.2: GUI mockup used in initial design process (2).

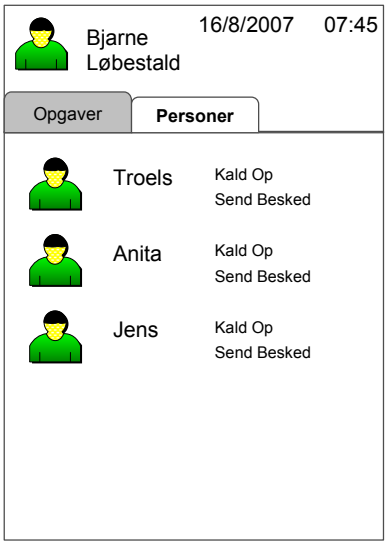

Figure C.3: GUI mockup used in initial design process (3).

Bibliography

[1] Aarhus University, Faculty of Science and Danish Agricultural Advisory Services. Crop protection online - demo version. Website `http://pvo.planteinfo.dk/cp/menu/menu.asp?id=demo\&subjectid=1\&language=en`, August 2009.

[2] R. Akkiraju, J. Farell, J. A. Miller, M. Nagarajan, A. Sheth, and K. Verma. Web service semantics – WSDL-S. In *W3C Workshop on Frameworks for Semantics in Web Services*, 2005.

[3] Rama Akkiraju, Joel Farrell, John Miller, Meenakshi Nagarajan, Marc-Thomas Schmidt, Amit Sheth, and Kunal Verma. Web service semantics - wsdl-s. Technical report, University of Georgia Research Foundation, Inc., International Business Machines Corporation, November 2005.

[4] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: connecting software architecture to implementation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, New York, NY, USA, 2002. ACM.

[5] Robert J. Allen. *A formal approach to software architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1997.

[6] Apache Software Foundation. Apache axis2/java - next generation web services. Website `http://ws.apache.org/axis2/`, July 2009.

[7] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and Patel F. P. Schneider. *The description logic handbook*. Cambridge University Press New York, NY, USA, 2007.

[8] J. E. Bardram. Hospitals of the future – ubiquitous computing support for medical work in hospitals. In J. E. Bardram, I. Korhonen, A. Mihailidis, and D. Wan, editors, *UbiHealth 2003: The 2nd International Workshop on Ubiquitous Computing for Pervasive Healthcare Applications.*, 2003.

[9] Jakob E. Bardram. The java context awareness framework (JCAF) – a service infrastructure and programming framework for context-aware applications.

In *Proceedings of the Third International Conference, PERVASIVE 2005*, pages 98–115, 2005.

[10] Louise Barkhuus and Anind Dey. Is context-aware computing taking control away from the user? three levels of interactivity examined. In *UbiComp 2003: Ubiquitous Computing*, pages 149–156, 2003.

[11] John Barton and Tim Kindberg. The cooltown user experience. Technical report, Hewlett-Packard Development Company, 2001.

[12] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture In Practice*. Person Education, Inc., 2nd edition, 2003.

[13] Paolo Bellavista, Antonio Corradi, Rebecca Montanari, and Cesare Stefanelli. Context-aware middleware for resource management in the wireless internet. *IEEE Transactions on Software Engineering*, 29(12):1086–1099, 2003.

[14] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284:28–37, 2001.

[15] Philip A. Bernstein. Middleware: a model for distributed system services. *Communications of the ACM*, 39(2):86–98, February 1996.

[16] Gavin Bierman and Alisdair Wren. First-class relationships in an object-oriented language. In *ECOOP 2005 - Object-Oriented Programming*, pages 262–286, 2005.

[17] G. Bjerknes, P. Ehn, and M. Kyng. *Computers and Democracy*. Avebury, 1987.

[18] Jeanette Blomberg, Lucy Suchman, and Randall H. Trigg. Reflections on a work-orinted design project. *Human-Computer Interaction*, 11:237–265, 1996.

[19] David Booth and Canyang K. Liu. Web services description language (WSDL) version 2.0 part 0: Primer. Technical report, World Wide Web Consortium, June 2007.

[20] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML) 1.0 (fifth edition). Technical report, World Wide Web Consortium, November 2008.

[21] Anne Bruseberg and Deana M. Philp. Focus groups to support the industrial/product designer: a review based on current literature and designers' feedback. *Applied Ergonomics*, 33(1), 2002.

[22] Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, and Naveen Srinivasan. OWL-S: Semantic markup for web services. Technical report, W3C, November 2004.

[23] M. Büscher, M. Christensen, K. M. Hansen, P. Mogensen, and D. Shapiro. Bottom-up, top-down? connecting software architecture design with use. In *Configuring User-Designer Relations: Interdisciplinary Perspectives*. Springer Publishing Company, Incorporated, 2008.

[24] Monika Büscher, Mette A. Eriksen, Jannie F. Kristensen, and Preben H. Mogensen. Ways of grounding imagination. In *PDC 04: Proceedings of the eighth conference on Participatory design*, pages 193–203. ACM Press, 2004.

[25] Russel Butek. Which style of WSDL should i use? Website `http://www-128.ibm.com/developerworks/webservices/library/ws-whichwsdl/`, October 2003.

[26] L. Capra, W. Emmerich, and C. Mascolo. Carisma: context-aware reflective middleware system for mobile applications. *Software Engineering, IEEE Transactions on*, 29(10):929–945, 2003.

[27] Licia Capra. Mobile computing middleware for context-aware applications. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 723–724, New York, NY, USA, 2002. ACM.

[28] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. Reflective middleware solutions for context-aware applications. In *Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 126–133, 2001.

[29] Alvin T. S. Chan and Siu N. Chuang. Mobipads: A reflective middleware for context-aware mobile computing. *IEEE Transactions on Software Engineering*, 29(12):1072–1085, 2003.

[30] Bo Chen, Zhoujun Li, and Huowang Chen. A new component-oriented programming language with the first-class connector. In *7th Joint Modular Language Conference*, pages 271–286, September 2006.

[31] Keith Cheverst, Keith Mitchell, and Nigel Davies. Investigating context-aware information push vs. information pull to tourists. In *In Proceedings of Mobile HCI 01*, volume 1, 2001.

[32] Aino Corry, Klaus Hansen, and David Svensson. Traveling architects – a new way of herding cats. In *Quality of Software Architectures*, pages 111–126, 2006.

[33] Fábio M. Costa. Meta-orb: A highly configurable and adaptable reflective middleware platform. Technical report, Instituto de Informática Universidade Federal de Goiás, 2002.

[34] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the web services web: an introduction to SOAP, WSDL, and UDDI. *Internet Computing, IEEE*, 6(2):86–93, March 2002.

[35] Karl Czajkowski, Donald F. Ferguson, Ian Foster, Jeffrey Frey, Steve Graham, Igor Sedukhin, David Snelling, Steve Tuecke, and William Vambenepe. The ws-resource framework version 1.0. Technical report, Computer Associates International, Inc., Fujitsu Limited, Hewlett- Packard Development Company, International Business Machines Corporation and The University of Chicago, 2004.

[36] Dansk Landbrugsrådgivning. *Kom godt i gang med DLBR Lommebedriften Online*, 2006.

[37] Dansk Landbrugsrådgivning. *Kom godt i gang med DLBR Minigrisen*, 2007.

[38] Eric M. Dashofy, Nenad Medvidovic, and Richard N. Taylor. Using off-the-shelf middleware to implement connectors in distributed software architectures. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 3–12, New York, NY, USA, 1999. ACM.

[39] Philippe Debaty, Patrick Goddi, and Alex Vorbau. Integrating the physical world with the web to enable context-enhanced services. Technical report, Hewlett-Packard Development Company, 2003.

[40] Anind K. Dey. Understanding and using context. *Personal Ubiquitous Comput.*, 5(1):4–7, February 2001.

[41] Anind K. Dey and Gregory D. Abowd. Towards a better understanding of context and context-awareness. Technical report, Graphics, Visualization and Usability Center and College of Computing, Georgia Institute of Technology, 1999.

[42] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Hum.-Comput. Interact.*, 16(2):97–166, 2001.

[43] Hector A. Duran-Limon, Gordon S. Blair, Adrian Friday, Paul Grace, George Samartzidis, Thirunavukkarasu Sivaharan, and Maomao Wu. Context-aware middleware for pervasive and ad hoc environments. Technical report, Computing Department, Lancaster University, 2003.

[44] Pelle Ehn and Morten Kyng. *Cardboard computers: mocking-it-up or hands-on the future*, pages 169–196. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1992.

[45] C. Escoffier, D. Donsez, and R. S. Hall. Developing an osgi-like service platform for .net. In *Consumer Communications and Networking Conference, 2006. CCNC 2006. 3rd IEEE*, volume 1, pages 213–217, 2006.

[46] Patrick T. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[47] David C. Fallside and Priscilla Walmsley. Xml schema part 0: Primer second edition. Technical report, World Wide Web Consortium, October 2004.

[48] FAO. Geopolitical ontology v. 0.9. Ontology `http://www.fao.org/aims/geopolitical.owl`, 2009.

[49] Joel Farrell and Holger Lausen. Semantic annotations for WSDL and XML schema. Technical report, W3C, 2007.

[50] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. Technical report, The Internet Society, June 1999.

[51] C. Floyd. *A Systematic Look at Prototyping*, pages 1–18. Springer-Verlag, Berlin, 1984.

[52] D. Garlan, D. P. Siewiorek, A. Smailagic, and P. Steenkiste. Project aura: toward distraction-free pervasive computing. *Pervasive Computing, IEEE*, 1(2):22–31, 2002.

[53] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[54] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, February 1992.

[55] Barnet G. Glaser and Anselm L. Strauss. *The Discovery of Grounded Theory – Strategies for Qualitative Research*. Aldine De Gruyter, New York, 1977.

[56] Google. Google soap search api (deprecated). Website `http://code.google.com/apis/soapsearch/`, 2006.

[57] Steve Graham, Peter Niblett, Dave Chappell, Amy Lewis, Nataraj Nagaratnam, Jay Parikh, Sanjay Patil, Shivajee Samdarshi, Steve Tuecke, William Vambenepe, and Bill Weihl. Web services notification (ws-notification) version 1.0. Technical report, International Business Machines Corporation, Sonic Software Corporation, SAP AG, Hewlett-Packard Development Company, Akamai Technologies Inc., Tibco Software Inc and The University of Chicago, 2004.

[58] Joan Greenbaum and Morten Kyng. *Design at work: cooperative design of computer systems*. Lawrence Erlbaum Associates, Inc., Mahwah, NJ, USA, 1992.

[59] Tao Gu, Hung K. Pung, and Da Q. Zhang. A middleware for building context-aware mobile services. In *Vehicular Technology Conference, 2004. VTC 2004-Spring. 2004 IEEE 59th*, volume 5, pages 2656–2660 Vol.5, 2004.

[60] Jeff Heflin. Owl web ontology language use cases and requirements. Technical report, W3C, 2004.

[61] Urs Hengartner and Peter Steenkiste. Protecting access to people location information. In *First International Conference on Security in Pervasive Computing*, pages 222–231, 2004.

[62] Hewlett-Packard. Jena - a semantic web framework for java. Website `http://jena.sourceforge.net/`, September 2009.

[63] Jason Hong, Gaetano Borriello, James Landay, David Mcdonald, Bill Schilit, and Doug Tygar. Privacy and security in the location-enhanced world wide web. In *Proceedings of Ubicomp 2003*, 2003.

[64] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. Swrl: A semantic web rule language combining owl and ruleml. Technical report, World Wide Web Consortium, May 2004.

[65] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From shiq and rdf to owl: the making of a web ontology language. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(1):7–26, December 2003.

[66] Richard Hull, Philip Neaves, and James Bedford-Roberts. Towards situated computing. In *ISWC '97: Proceedings of the 1st IEEE International Symposium on Wearable Computers*, Washington, DC, USA, 1997. IEEE Computer Society.

[67] M. Humphrey, G. Wasson, K. Jackson, J. Boverhof, M. Rodriguez, J. Gawor, J. Bester, S. Lang, I. Foster, S. Meder, S. Pickles, and Mc. State and events for web services: a comparison of five ws-resource framework and ws-notification implementations. In *High Performance Distributed Computing, 2005. HPDC-14. Proceedings. 14th IEEE International Symposium on*, pages 3–13, 2005.

[68] Apple Inc. Networking – bonjour. Website `http://developer.apple.com/networking/bonjour/`, September 2009.

[69] G. Judd and P. Steenkiste. Providing contextual information to pervasive computing applications. In *Pervasive Computing and Communications, 2003. (PerCom 2003). Proceedings of the First IEEE International Conference on*, pages 133–142, 2003.

[70] R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic, and R. N. Taylor. xadl: enabling architecture-centric tool integration with xml. In *System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on*, pages 9 pp.+, August 2001.

[71] Kristian E. Kjær. Ethnographic studies as a requirement gathering process for the design of context aware middleware. In *MC '07: Proceedings of the 2007 ACM/IFIP/USENIX international conference on Middleware companion*, pages 1–2, New York, NY, USA, 2007. ACM.

[72] Kristian E. Kjær. First class connectors for prototyping service oriented architectures. In *First European Conference on Software Architecture*, pages 171–178, 2007.

[73] Kristian E. Kjær. A survey of context-aware middleware. In *SE'07: Proceedings of the 25th conference on Software Engineering, IASTED International Multi-Conference*, pages 148–155, Anaheim, CA, USA, 2007. ACTA Press.

[74] Kristian E. Kjær. Designing middleware for context awareness in agriculture. In *MDS '08: Proceedings of the 5th Middleware doctoral symposium*, pages 19–24, New York, NY, USA, 2008. ACM.

[75] Kristian E. Kjær and Klaus M. Hansen. Modeling and implementing ontology-based publish/subscribe using semantic web technologies. In *25th Symposium On Applied Computing (submitted)*, 2010.

[76] Matthias Klusch, Benedikt Fries, and Katia Sycara. Automated semantic web service discovery with owls-mx. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 915–922, New York, NY, USA, 2006. ACM.

[77] Holger Knublauch, Ray W. Fergerson, Natalya F. Noy, and Mark A. Musen. The protégé owl plugin: An open development environment for semantic web applications. In *Proceedings of the Third International Semantic Web Conference*, pages 229–243, 2004.

[78] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering, 2007. FOSE '07*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[79] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1999.

[80] Salvatore T. March and Gerald F. Smith. Design and natural science research on information technology. *Decision Support Systems*, 15(4):251–266, December 1995.

[81] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *MI4*, pages 463–502, 1969.

[82] James McGovern, Sameer Tyagi, Michael Stevens, and Sunil Mathew. *Java Web Services Architecture*, chapter 10, JAX-RPC, pages 313–403. Elsevier Science, 2003.

[83] Deborah L. McGuinness and Frank van Harmelen. OWL web ontology language overview. Technical report, W3C, 2004.

[84] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, January 2000.

[85] Nenad Medvidovic. On the role of middleware in architecture-based software development. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 299–306, New York, NY, USA, 2002. ACM.

[86] Ulrich Meissen, Stefan Pfennigschmidt, Agnès Voisard, and Tjark Wahnfried. Context- and situation-awareness in information logistics. In *Current Trends in Database Technology - EDBT 2004 Workshops*, pages 335–344, 2005.

[87] Microsoft. Microsoft .NET framework. Website `http://www.microsoft.com/net/`, 2008.

[88] Nilo Mitra and Yves Lafon. Soap version 1.2 part 0: Primer (second edition). Technical report, World Wide Web Consortium, April 2007.

[89] P. Mogensen and R. Trigg. Using artefacts as triggers for participatory analysis. In M. Muller, S. Kuhn, and J. Meskill, editors, *Proceedings of the Participatory Design Conference (PDC) 1992*, pages 55–62, 1992.

[90] Gero Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt University of Technology, 2002.

[91] Ian Niles and Adam Pease. Towards a standard upper ontology. In *FOIS '01: Proceedings of the international conference on Formal Ontology in Information Systems*, pages 2–9, New York, NY, USA, 2001. ACM.

[92] Guillaume Nodet. The servicemix-wsn2005 JBI component. Website `http://servicemix.apache.org/servicemix-wsn2005.html`, 2008.

[93] OASIS. UDDI — online community for the universal description, discovery, and integration OASIS standard. Website `http://uddi.xml.org`, August 2009.

[94] Martin O'Connor, Csongor Nyulas, Ravi Shankar, Amar Das, and Mark Musen. The swrlapi: A development environment for working with swrl rules. In *Proceedings of the International Workshop on OWL: Experiences and Directions (OWLED 2008)*, 2008.

[95] Plato. *Republic*, chapter Book 7, pages 225+. Wordsworth Editions Limited, 1997.

[96] Anand Ranganathan and Roy H. Campbell. A middleware for context-aware agents in ubiquitous computing environments. In *Middleware '03: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 143–161, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[97] Anand Ranganathan, Jalal A. Muhtadi, Shiva Chetan, Roy Campbell, and M. Dennis Mickunas. Middlewhere: a middleware for location awareness in ubiquitous computing applications. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 397–416, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[98] Raymond Reiter. On closed world data bases. In Gallaire A. Minker, editor, *Logic and Data Bases*. Plenum Press, 1978.

[99] Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1(4):74–83, October 2002.

[100] Sandia National Laboratories. Jess, the rule engine for the Java platform. Website `http://www.jessrules.com/`, September 2009.

[101] Martin Sarnovsky, Peter Butka, Peter Kostelnik, and Dasa Lackova. Hydra – network embedded sys- tem middleware for ambient intelligent devices. In *ICCC'2007: Proceedings of 8th Inter- national Carpathian Control Conference*, 2007.

[102] M. Satyanarayanan. Pervasive computing: vision and challenges. *Personal Communications, IEEE*, 8(4):10–17, 2001.

[103] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on*, pages 85–90, 1994.

[104] B. N. Schilit and M. M. Theimer. Disseminating active map information to mobile hosts. *Network, IEEE*, 8(5):22–32, 1994.

[105] William N. Schilit. *A system architecture for context-aware mobile computing*. PhD thesis, Columbia University, New York, NY, USA, 1995.

[106] Douglas C. Schmidt. Middleware for real-time and embedded systems. *Commun. ACM*, 45(6):43–48, June 2002.

[107] Mary Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In *Studies of Software Design*, pages 17–32, 1996.

[108] Thirunavukkarasu Sivaharan, Gordon Blair, and Geoff Coulson. Green: A configurable and re-configurable publish-subscribe middleware for pervasive

computing. In *OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005*, pages 732–749, 2005.

[109] Carl F. Sorensen, Maomao Wu, Thirunavukkarasu Sivaharan, Gordon S. Blair, Paul Okanda, Adrian Friday, and Hector D. Limon. A context-aware middleware for applications in mobile ad hoc environments. In *MPAC '04: Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, pages 107–110, New York, NY, USA, 2004. ACM.

[110] João P. Sousa and David Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *WICSA 3: Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture*, pages 29–43, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[111] Thomas Strang and Claudia Linnhoff-Popien. A context modeling survey. In *Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004 - The Sixth International Conference on Ubiquitous Computing, Nottingham/England*, 2004.

[112] Sun Microsystems Inc. Java message service (JMS). Webaite `http://java.sun.com/products/jms/`, 2009.

[113] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component- and message-based architectural style for gui software. *Software Engineering, IEEE Transactions on*, 22(6):390–406, 1996.

[114] The OSGi Alliance. OSGi service platform – release 4. Technical report, 2005.

[115] William Vambenepe, Steve Graham, and Peter Niblett. Web services topics (ws-topics). Technical report, OASIS, 2006.

[116] Jinling Wang, Beihong Jin, and Jing Li. An ontology-based publish/subscribe system. In *Middleware 2004*, pages 232–253, 2004.

[117] Roy Want, Bill N. Schilit, Norman I. Adams, Rich Gold, Karin Petersen, David Goldberg, John R. Ellis, and Mark Weiser. *The Parctab Ubiquitous Computing Experiment*, volume 353, chapter Chapter 2, pages 45–101. Springer US, Boston, MA, 1996.

[118] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3), 1991.

[119] Ludwig Wittgenstein. *Philosophical Investigations*. Blackwell Publishing Ltd, 1953.

[120] Weishan Zhang, Klaus M. Hansen, and Thomas Kunz. Enhancing intelligence and dependability of a product line enabled pervasive middleware. *Pervasive and Mobile Computing*, July 2009.

[121] Andreas Zimmermann, Andreas Lorenz, and Reinhard Oppermann. An operational definition of context. In *6th International and Interdisciplinary Conference, CONTEXT 2007*, pages 558–571, 2007.

[122] H. Zimmermann. OSI reference model–the ISO model of architecture for open systems interconnection. *Communications, IEEE Transactions on*, 28(4):425–432, 1980.