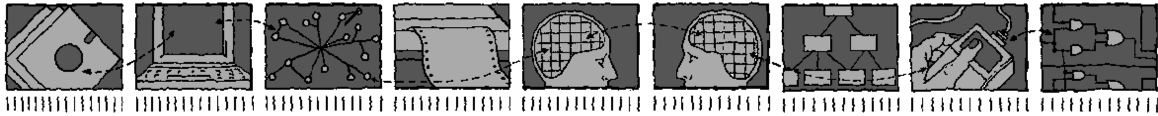
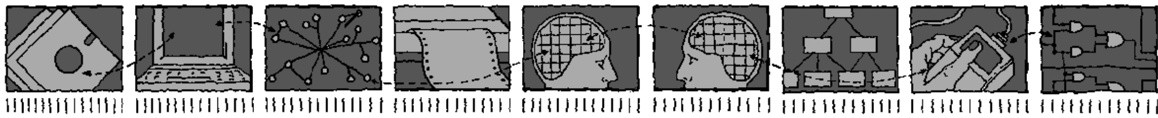


Department of Computing Science and Mathematics
University of Stirling



An Ontology Based Approach Towards A Universal Description Framework for Home Networks

Liam S. Docherty

Technical Report CSM-182

ISSN 1460-9673

February 2010

Department of Computing Science and Mathematics
University of Stirling

An Ontology Based Approach Towards A Universal Description Framework for Home Networks

Liam S. Docherty

Department of Computing Science and Mathematics
University of Stirling
Stirling FK9 4LA, Scotland

Telephone +44 1786 467 421, Facsimile +44 1786 464 551
Email lsd@cs.stir.ac.uk

Technical Report CSM-182

ISSN 1460-9673

February 2010

Abstract

Current home networks typically involve two or more machines sharing network resources. The vision for the home network has grown from a simple computer network, to every day appliances embedded with network capabilities. In this environment devices and services within the home can interoperate, regardless of protocol or platform. Network clients can discover required resources by performing network discovery over component descriptions. Common approaches to this discovery process involve simple matching of keywords or attribute/value pairings.

Interest emerging from the Semantic Web community has led to ontology languages being applied to network domains, providing a logical and semantically rich approach to both describing and discovering network components. In much of the existing work within this domain, developers have focused on defining new description frameworks in isolation from existing protocol frameworks and vocabularies.

This work proposes an ontology-based description framework which takes the ontology approach to the next step, where existing description frameworks are incorporated into the ontology-based framework, allowing discovery mechanisms to cover multiple existing domains. In this manner, existing protocols and networking approaches can participate in semantically-rich discovery processes. This framework also includes a system architecture developed for the purpose of reconciling existing home network solutions with the ontology-based discovery process.

This work also describes an implementation of the approach and is deployed within a home-network environment. This implementation involves existing home networking frameworks, protocols and components, allowing the claims of this work to be examined and evaluated from a 'real-world' perspective.

Acknowledgements

A number of people have had an impact on my time as a PhD student. I would like to acknowledge their contribution to my work and experience. Firstly, I would like to acknowledge the Scottish Funding Council who funded my PhD through the MATCH project. I would also like to thank the members of the MATCH project for their views and feedback on both the theory and implementation of my work.

I would also like to thank my supervisors Evan Magill and Mario Kolberg for the numerous conversations, thoughts, criticisms and suggestions which have shaped my thought process and approach towards this work. In particular, I would like to thank Evan for his understanding during the various circumstances I have encountered during the PhD process.

I would like to thank Paul Godley, Gavin Campbell, Lloyd Oteniya, Claire Maternaghan and Tony McBryan who each provided valuable contributions to my PhD experience, sharing thoughts, advising and testing out various aspects of my work. My experience has been, without doubt, a more fulfilling and enriching time because of them.

I would like to thank my family and my parents-in-law for the unmeasurable levels of support they have provided. There are no words eloquent enough to express how important my wife, Nicola, has been through my entire academic career. She is the overwhelming reason for my accomplishments to date. May this work be a testament to her, as it would not exist without her.

My final thanks are given to God. He is both my provider and my strong tower.

Contents

Abstract **i**

Acknowledgements **ii**

1 Introduction **1**
1.1 Issues 1
1.2 Ontologies 2
1.3 Aims 2
1.4 Contribution 2
1.5 Overview of the thesis 3

2 The Home Network **4**
2.1 Entertainment in the Home Network 4
2.2 The Progressing Vision 5
2.3 Comment on the Home Network Vision 5
2.3.1 The Single Vendor Environment 6
2.3.2 The Two Vendor Environment 6
2.3.3 The Multi-Vendor, Multi-Protocol Environment 6
2.3.4 Descriptions within the Home Network 7
2.4 Home Care in the Home Network 7
2.4.1 Motivation for the Home Care Network 7
2.4.2 The Emerging Home Care Network 8
2.5 Merging Aims of the Home Network 9

3 Home Network Solutions **10**
3.1 Home Network Terminology 10
3.2 Universal Plug and Play 11
3.2.1 Joining the Network 11
3.2.2 Device and Service Descriptions 11
3.2.3 Discovering other Devices 13
3.2.4 Joining a network 14
3.2.5 Leaving a network 14
3.2.6 Discovery in the Network 14
3.2.7 Replying to Searches 14
3.2.8 Controlling Devices and Services 15
3.2.9 Control Messages and SOAP 15
3.2.10 Events within the Network 15
3.2.11 Presentation 15
3.2.12 The Suitablilty of UPnP 16
3.3 Home Audio/Video Interoperability (HAVi) 16
3.3.1 Controllers, Devices, DCMs and FCMs 16
3.3.2 Device Classification 16
3.3.3 HAVi Networks 17
3.3.4 Descriptions and Discovery 17

3.3.5 Communication . . . 18
3.3.6 The Suitability of HAVi . . . 18
3.4 X.10 . . . 18
3.4.1 Plugging into X.10 . . . 19
3.4.2 Control . . . 19
3.4.3 Discovery . . . 19
3.4.4 Issues in X.10 . . . 20
3.5 Interoperability in the Home . . . 20
3.6 A Naive Home Network Scenario . . . 20
3.7 Protocol Bridges . . . 22
3.8 Middleware . . . 23
3.9 OSGi . . . 23
3.9.1 Life cycle Management . . . 23
3.9.2 Services And Bundles Within The OSGi Framework . . . 24
3.9.3 The Bundle Activator . . . 24
3.9.4 Registering Services . . . 25
3.9.5 Discovering Services . . . 25
3.9.6 Using Services . . . 25
3.9.7 Scope of OSGi . . . 26
3.10 Jini . . . 26
3.10.1 Jini Clients . . . 26
3.10.2 Registering Services . . . 27
3.10.3 Discovering Services . . . 27
3.10.4 Service Registry Groups . . . 27
3.10.5 Proxies within the Jini Network . . . 28
3.11 An OSGi Supported Home Network . . . 28
3.11.1 Service-Orientated Discovery . . . 30
3.11.2 Registrations . . . 30
3.11.3 Environment Evaluation . . . 31
3.11.4 X.10 Registration . . . 31
3.11.5 UPnP Registration . . . 31
3.11.6 Jini Registration . . . 33
3.11.7 Evaluation of the OSGi Discovery Process . . . 34
3.11.8 OSGi Evaluation Conclusion . . . 36

4 Service Discovery Approaches in the Home Network **37**
4.1 System Architecture . . . 37
4.1.1 Decentralised Architecture . . . 37
4.1.2 Centralised Architecture . . . 38
4.2 Component Description . . . 38
4.2.1 No Description . . . 39
4.2.2 Attribute/Value Pairs . . . 40
4.2.3 Description Schemas . . . 41
4.2.4 Comparison of approaches . . . 41

5 Ontology Languages **43**
5.1 Meta-Data . . . 43
5.2 The Resource Description Framework . . . 44
5.2.1 Expanding the Boundaries . . . 44
5.2.2 Using Resources . . . 44
5.2.3 Transporting and Exchanging RDF . . . 45
5.2.4 RDF/XML . . . 46
5.2.5 Issues in RDF . . . 47
5.3 RDF Schema . . . 48
5.3.1 Resources in RDFS . . . 48
5.3.2 Properties in RDFS . . . 49

5.3.3 RDFS Issues . . . 50
5.3.4 The Semantic Web . . . 50
5.4 The Web Ontology Language . . . 51
5.4.1 OWL Resources . . . 51
5.4.2 OWL Properties . . . 52
5.4.3 OWL Restriction Classes . . . 54
5.5 Domain Descriptions . . . 55
5.5.1 Using owl:imports . . . 55
5.5.2 Describing Small Domains . . . 55
5.5.3 Describing Large Domains . . . 56
5.6 Ontology Conclusion . . . 58

6 Ontologies Within Web Services **60**
6.1 Web Services . . . 60
6.1.1 Web Service Description Language . . . 60
6.1.2 UDDI . . . 61
6.2 OWL-S . . . 62
6.2.1 The Service Profile . . . 63
6.2.2 The Service Model . . . 63
6.2.3 The Service Grounding . . . 63
6.2.4 The Feasible Impact of OWL-S . . . 64
6.3 Other Ontology-based Web Service Description Approaches . . . 66
6.3.1 SWSF . . . 66
6.3.2 WSMO . . . 67
6.4 Web Services Conclusion . . . 68

7 Ontology Related Work **69**
7.1 Home Network Ontology Projects . . . 69
7.1.1 The Networked Appliance Service Utilisation Framework . . . 69
7.1.2 The Gadgetware Architectural Style Ontology . . . 70
7.1.3 Other Home Network Projects . . . 70
7.1.4 Service Discovery within Home Networks . . . 70
7.2 Context Aware Systems . . . 71
7.3 Upper Ontologies . . . 73
7.3.1 Developing and Using Upper Ontologies . . . 74
7.3.2 Cyc and OpenCyc . . . 74
7.3.3 The Suggested Upper Merged Ontology (SUMO) . . . 75
7.3.4 DOLCE . . . 75
7.4 Comment on Existing Approaches . . . 75

8 Approach **77**
8.1 Purpose and Scope of the Approach . . . 77
8.2 Approach Method . . . 78
8.2.1 The Base Layer . . . 78
8.2.2 The Core Layer . . . 79
8.2.3 The Generic Layer . . . 80
8.2.4 The Protocol Layer . . . 82
8.2.5 Converting Between Protocol and Ontology Descriptions . . . 84
8.2.6 Developer Defined Ontologies . . . 85
8.3 HNOS Conclusion . . . 85

9 System Architecture and Implementation **87**
9.1 Implementation Overview 87
9.2 The Ontology Registry 88
9.3 The Simple Query Interface 90
9.3.1 Simple Queries 90
9.3.2 Complex Queries 90
9.3.3 Meta Queries 91
9.4 Summary of the Ontology Registry Architecture 92
9.5 Protocol Translators 92
9.5.1 The Translation Process 92
9.5.2 The UPnP Translation Bundle 93
9.5.3 The Jini Translation Bundle 94
9.5.4 The X.10 Translation Bundle 95
9.5.5 Summary of the Translation Approach 95

10 Deploying the Approach **97**
10.1 Applying the Approach to a Home Network Environment 97
10.1.1 Modifications and Assumptions 97
10.1.2 Ontology-Based Descriptions 98
10.1.3 Discovery with an Ontology Vocabulary 100
10.1.4 Adding New Protocols 102
10.1.5 Conclusion of the Ontology Approach 103

11 The MATCH Project **104**
11.1 System Architecture 104
11.1.1 User Interaction 104
11.1.2 System Communication 105
11.1.3 Core System Components 105
11.1.4 The Policy Server 105
11.1.5 The Task Manager 106
11.1.6 The Interaction Manager 106
11.1.7 The Resource Registry 106
11.1.8 System Review 106
11.2 The Role of Ontologies within MATCH 107
11.2.1 Describing Components 107
11.2.2 Evolving Vocabulary 107
11.2.3 Describing Interaction Details 108
11.2.4 Logic Based Discovery 108

12 Evaluation **110**
12.1 The Protocol/Vocabulary Relationship 110
12.2 The Reasoning Approach of the Registry 111
12.3 The Responsiveness of the Registry 111
12.3.1 Evaluation of the HNOS 112
12.3.2 Evaluation of the MATCH System Environment 112
12.3.3 Evaluation of the Registry with Additional Ontologies 112
12.3.4 Response Evaluation 113
12.4 The Range of the Vocabulary and Approach 114
12.4.1 Web Service Deployment 114
12.4.2 Peer to Peer Domains 115
12.5 The Versatility of the Approach 115
12.5.1 The Home Network Ontology Stack 115
12.6 The System Architecture and Implementation 116
12.7 Limitations of the Approach 116
12.7.1 Logical Metadata 116
12.7.2 Agreement on Standards 117

12.7.3 Component View of the Network . . . 117
12.7.4 Translators . . . 117

13 Conclusions and Future Work 119
13.1 Home Network Conclusions . . . 119
13.2 Review of Work . . . 120
13.3 Future Work . . . 121
13.3.1 Deploying the Approach Within Other Middleware and Environments . . . 121
13.3.2 Abstract Actions . . . 122
13.4 Final Remarks . . . 123

List of Figures

2.1 Initial Home Networks 4

3.1 CyberGarage Air Conditioner 12
3.2 Services Provided by the Air Con 12
3.3 Services of the Air Conditioner 13
3.4 A Discovery Message Template. 15
3.5 Relationship of the Device, DCM and FCM 17
3.6 An Example HAVi Network of Clusters 18
3.7 An X.10 Appliance Module 19
3.8 Muti-protocol Home Network 20
3.9 A Naive Home Network 21
3.10 Naive Discovery 21
3.11 Using a UPnP-to-HAVi Bridge 22
3.12 Middleware in the Home Network 23
3.13 An Example Bundle Manifest 24
3.14 The OSGi Service Registry 25
3.15 Relation Between Services and Users 26
3.16 Jini Registry Providing a Registrar 27
3.17 Copying The Service Object Into The Registry 28
3.18 Interaction Between the Registry and User 29
3.19 Service Proxies and Service Controller 29
3.20 Registering and Retrieving the Service Proxy 29
3.21 An X.10 Registration in OSGi 31
3.22 UPnP Description of the Light Device 32
3.23 UPnP Description Converted for OSGi 32
3.24 UPnP Description of the Power Service 33
3.25 UPnP Description of the Power Service Converted for OSGi 33
3.26 Jini Description of a Lamp Controller 33
3.27 Jini Description of the Lamp Controller converted for OSGi 34
3.28 Discovering the X.10 Lamp Service 34
3.29 Discovering the UPnP Power Service 35
3.30 Discovering the Jini Service 35
3.31 The Modified Jini Query 35

4.1 Discovery in the Decentralised Network 38
4.2 Discovery in the Centralised Network 39

5.1 RDF Graph about www.stir.ac.uk/ lsd 45
5.2 Expressing RDF Information in XML 46
5.3 RDFS Relations 48
5.4 Description of a Customer class 56
5.5 Description of a DetailedCustomer class 57
5.6 Relationships through Ontology Reuse 58

6.1 Overview of an OWL-S Description 62

6.2 The SearchMusicByArtist Service . . . 64
6.3 Service User Knowledge . . . 64
6.4 Specifying the relationship between class instances . . . 66

7.1 OWL representation of an OccupiedRoom . . . 72
7.2 OWL representation of the MeetingRoom . . . 72
7.3 Customising an Upper Ontology for Use . . . 74
7.4 Boundaries Between Protocols . . . 75
7.5 Common Communication Protocol . . . 76

8.1 The Base Level of the Stack . . . 78
8.2 The Base Ontology . . . 78
8.3 The Core Level of the Stack . . . 79
8.4 The Generic Layer of the Stack . . . 81
8.5 Description of an Audio Speaker . . . 81
8.6 The Protocol Layer of the Stack . . . 82
8.7 Architecture of Protocol Devices . . . 83
8.8 Architecture of Lamp Devices . . . 83
8.9 Using an Existing Television_Device Description . . . 85
8.10 The Development Layer of the Stack . . . 86

9.1 The Middleware Architecture . . . 88
9.2 Algorithm for converting Complex Queries . . . 91
9.3 Algorithm behind Meta Queries . . . 91
9.4 The Ontology Registry Architecture . . . 92
9.5 Translation within the UPnP Translator . . . 93
9.6 A UPnP Alert . . . 94
9.7 Translation within the Jini Translator . . . 94
9.8 The X.10 User Wizard . . . 95
9.9 Translation within the X.10 Translator . . . 96

10.1 Ontology Description of the X.10 Lamp . . . 98
10.2 Ontology Description of the UPnP Light . . . 99
10.3 Ontology Description of the UPnP Service . . . 99
10.4 Ontology Description of the Jini Light Interface . . . 99
10.5 Simple Ontology Query . . . 100
10.6 A Complex Ontology Query . . . 101
10.7 A Logic Based Query . . . 102
10.8 Unsupported Excite Protocol . . . 102

11.1 The MATCH System . . . 107
11.2 A Sample MATCH Query . . . 108
11.3 Description of TomsPersonalTracker Held by the Registry . . . 108
11.4 Relationships within the MATCH Ontology . . . 109

12.1 Response times of the Registry using the HNOS . . . 112
12.2 Response Times of the Registry Within the MATCH system . . . 112
12.3 Response Times of the Registry using Imported Ontologies . . . 113
12.4 Comparisons of Response Times . . . 113
12.5 Comparison of Secondary Response Times . . . 114

List of Tables

4.1 Table of Comparisons 42

Chapter 1

Introduction

The idea of the home network has changed in recent years, due to the increasing availability of both network devices and high speed internet connections and the low costs of computing hardware. Typical existing home networks consist of desktop computers, laptops and mobile phones all sharing resources, such as documents, printers or internet access. In recent years, the boundaries of the home network have changed. Substantial research and development has been given to redefining simple household appliances, allowing these to become resources within the network [58, 38]. Pervasive computing involves computational components being embedded in everyday appliances and objects, becoming 'invisible' to the home user. Taking advantage of this new discipline promises exciting possibilities for the home network, where the services offered by appliances are now available to the network. The mobile phone can be used to switch on the central heating. A power monitoring program running on the desktop computer can maintain levels of power consumption without direct intervention of the user. Films can be streamed directly from the laptop DVD drive to the family television. The network acts like a marketplace, offering services and resources to interested parties [76] . Transactions are initiated and concluded in a predefined manner.

Viewing the possibilities from such a high level obscures the issues facing this vision of the home network. The main issues, and that which also draws the majority of research, is the locating and use of the devices and services on offer. Part of this process requires discovering the availability of desired resources within the network. Descriptions have a decisive role within the discovery process, especially within the home network, where resources may differ in terms of protocol, platform, developer and vendor [36, 11].

1.1 Issues

In networked environments which share a common protocol, discovery can be a relatively straight-forward process. Each network component would know exactly what resource or service it required. The description schemas and vocabularies used to describe the various components could allow rich representations of the attributes and capabilities of components within the network. As the syntax would be shared by all components, discovery could involve a predefined set of desired attributes of a component, which could be fixed before the client component joined the network. As all components *'know'* about the set of vocabularies used, they are able to *'understand'* descriptions within their own network. For example, components using the Universal Plug and Play protocol (covered in section 3.2) understand the terms used in the description process.

The emerging home network environment cannot afford this luxury approach to component description and discovery. This is a result of the dynamic nature of the environment. The home network is a collection of ad hoc devices and services, which may include a wide range of differing protocols [10, 65]. The term protocol need not only refer to how components communicate, but also how components discover and describe themselves. In such an environment, the discovery process can be hindered by the possible number of protocols existing. Each protocol may have its own internal classification system, or its own way of referring to a particular component.

This issue can be illustrated with an example in the natural language world. In the English language, the desktop PC is classified as *'the computer'*. In the French language, the same desktop PC is classified as *'de ordinateur'*. This example highlights a situation where the *terms* describing the *same object* are syntactically different. This issue can exist in the home network environment. The protocol-specific vocabularies used to describe the classification of the same component may be syntactically different from one another. To resolve this

issue, a more adaptive and robust approach towards supporting discovery is required.

It is not enough to simply create a new protocol, as this would alienate any existing home network protocols from participation. This approach must refrain from excluding existing home network protocols, and be extensive enough to include emerging protocols. To fulfill this requirement, the approach must be independent from any specific protocol or existing discovery method. In essence, the approach would require to project the illustration of a marketplace, where services and resources may be offered by many vendors, but there is a common understanding of the 'goods' required.

1.2 Ontologies

An emerging solution for the issues described can be found within computing-orientated Ontologies [82, 62]. Ontology languages can be used to provide a common understanding between clients and providers. Ontology languages come in many flavours, but are designed for a single purpose: To define data in a structured, logical and *machine understandable* way. These languages can be used to unify description vocabularies or define common knowledge bases. Chapter 5 provides an in depth overview of ontologies.

1.3 Aims

The primary aim of this work is to provide a description approach capable of addressing the home network issues identified. This work aims to provide a description framework which:

- is concerned with the Home Network, but abstract from any particular protocol
- allows inclusion of existing protocols and their descriptive vocabularies
- is extensive and scalable to suit new and developing protocols, their services and devices, and new environments

This work also aims to define an approach for adapting current home network environments, allowing existing components to benefit from the description framework. In particular, the aims of this approach are to:

- allow existing network services and devices to 'plug in' without need to conform to the new description framework
- allow the description framework to be applied across different home network environments, not being tied to a single solution

1.4 Contribution

The contribution of this work is novel in intention and in approach. While efforts exist which provide new description frameworks for the home network [82, 70, 19], no work exists which attempts to reconcile this work with *existing* protocols. This can lead to existing frameworks being isolated and rendered obsolete. In an environment where multi-protocols may exist, there is little to be gained from simply introducing a new framework which ignores previous work. Similarly, no work exists which allows the new description framework to retain the *protocol specific* information within the description. Assumptions of what data is important and what is not are difficult to make in environments where parties or agents may be interested in different aspects of a description.

This approach satisfies the aims stated previously by applying ontology languages to the domain. Rather than simply creating a new stand-alone vocabulary or schema-based framework, this work proposes a two-stage approach towards supporting cross-domain discovery.

The first stage of this approach is designed to support network clients in the discovery process. By viewing the home network as a collection of multiple description domains, ontologies are used to unify these domains. Relationships and associations are created between protocol specific attributes and generic ontology concepts. Semantic intentions behind protocol specific terminologies are unified with a view towards removing syntactical differences within the home network. In this manner distinctions between protocols can be removed at the descriptive level. Providing a single vocabulary which can be applied to multiple domains allows a logical and

intelligent approach to discovery in the home. By using the vocabulary, network clients can use a common set of terms to discover network components, regardless of their operating protocol or platform. The approach is scalable to incorporate new protocols and components, allowing the generic vocabulary to remain relevant over time, evolving as the home network evolves.

The second stage of this approach is aimed at supporting protocols within the home network. Deploying an abstract description framework within a home network would require all existing and emerging protocols to conform. By not doing so, protocols risk being alienated from the discovery process. This work proposes a system architecture designed to address this issue. By providing interpretation between protocols specific terminology and the generic ontology vocabulary, existing protocols can take part in the discovery process. The architecture described in this work supports protocol domains by interpreting descriptions on behalf of the protocol. In this manner, low level heterogeneous descriptions are transformed into high-level generic descriptions without requiring existing protocols to conform. In much the same way as the ontology vocabulary is designed to hide the protocol distinctness at the descriptive level, the system architecture is designed to obscure the generic discovery process from all existing protocols within the network.

This work describes a working implementation of both stages, specifying a set of ontologies to unify existing protocols and deploying an implementation of the system architecture within an existing middleware framework. The implementation demonstrates that this approach is not only successful in satisfying the issues identified, but is both relevant and viable within existing and emerging environments

1.5 Overview of the thesis

The thesis is arranged in the following order:

- Chapter 2 introduces the home network. It discusses the present state of the network concept and introduces the progressing vision for this environment. With the introduction of this vision, initial issues are highlighted and discussed.
- Chapter 3 is concerned with reviewing existing popular protocols and technologies developed so support home network environments. This chapter discusses the various strength and weakness of existing technology, and introduces scenarios and experiments which provide the motivation for this work.
- Chapter 4 reviews common approaches towards service and device discovery within home and other relevant network technologies. The chapter compares approaches and evaluates the usefulness of each approach in satisfying the needs of emerging home network scenarios. Based upon the conclusions drawn, this chapter proposes the use of ontology languages to address weaknesses identified.
- Chapter 5 discusses the ontology languages used by this work, including their strengths and weakness in addressing the issues identified.
- Chapters 6 and 7 discuss existing approaches and applications of ontology languages within networked domains. These approaches are evaluated to both show the relevance of the technology and show that this work is unique in its approach and intention.
- Chapters 8 and 9 provide the main body of the thesis. These chapters are concerned with describing this work, highlighting the advantages of this work and providing evidence to support these claims. These chapters also discuss a working implementation of this approach using existing technologies and frameworks.
- Chapter 10 applies the implementation described in Chapters 8 and 9 to a real-world home network scenario.
- Chapter 11 introduces the MATCH project, and describes how this work was integrated into a home care system.
- Chapter 12 evaluates this work, in accordance with existing work and against the claims made. This chapter also describes the weaknesses and limitations of this work, and provides discussion as to how these could be addressed.
- Chapter 13 concludes this work with a discussion of the contribution of this work to the research and development community. This chapter also discusses possible areas of future study.

Chapter 2

The Home Network

The home network environment can be defined in a number of ways. At the simplest level, a home network consists of a few computers sharing files between each other. This domain has grown to include the sharing of peripherals (such as printers) and other resources (such as external storage and internet connections - See Figure 2.1). Currently, the typical home network would be at this level, with computers and games consoles sharing a single internet access point within the home. On top of the relatively static machines, mobile computers (such as PDAs and mobile phones) can rapidly join and leave the network. In this manner, the home network can take on the shape of a Local Area Network (LAN).

While existing implementations of the home network are at this level, a great deal of interest and research is being investing into expanding the boundaries of the network, by integrating existing technology and providing more functionality. This chapter is concerned with emerging configurations of the home network, examining the possibilities offered and issues which require to be addressed.

2.1 Entertainment in the Home Network

From an entertainment perspective, the existence of digital television content heralds a new level of control for the user [15]. With the introduction of digital transmission, the user can now record their favourite programs, download 'on-demand' programs and rent out films directly from their digital box. The transmission can be wired (cable) or wireless. In many homes, the home digital box would be connected to a television, and the media would be streamed directly via a connected cable.

At this point, an intervention by the home network could allow the media from the digital box to be streamed wirelessly to any device in the home capable to displaying the visual data, and producing the sound output. For example, after renting out a film, the presentation data could be streamed to whichever room the user was

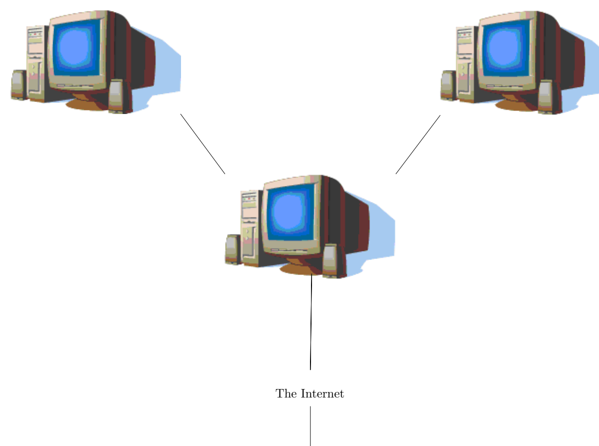


Figure 2.1: Initial Home Networks

occupying, provided it contained a television. If the user needed a drink while watching the film, the media could be switched from the lounge television to the small TFT television in the kitchen. To take the example one stage further, suppose the user then moved through to the study. With a television lacking in this room, the home networked could stream the visual data to the computer monitor, and re-route the sound data through the study stereo system. The changing location of the user could be derived from sensor readings which convey user movement.

While this may be a relatively far-fetched example in practical terms, it highlights the current opinion on what the next stage for the home network could be. The above example dealt with a simple, entertainment based scenario which involved a specific subset of home entertainment appliances. This example is concerned with a single-point solution. A single-point solution may include many different components, and require interaction between these components, but is ultimately concerned with a single outcome. In this case, the outcome is to allow the user to watch a film, regardless of their location.

The expanding vision of the home network may be driven by such small scenarios, but the realisation of this vision can have further reaching possibilities. If the home network is not only concerned with supporting single point solutions, but also with supporting large-scale, multi-goal solutions, then the domains which could benefit from the home network become more numerous.

2.2 The Progressing Vision

Rather than having a collection of single point solutions, offering these solutions as a collection of 'services' within the network would allow a flexible, dynamic and responsive home network environment. Using the above example, if the televisions and monitors of the home could communicate with, not only the digital box, but any device capable of producing visual output, the boundaries of the network could be pushed beyond simple scenarios. For example, it would be possible to participate in a webcam chat where the video feed of the other webcam could be streamed from the laptop to the lounge television, without the need to expressly set up the communication channels.

In this example, the ability of the lounge television to accept video data would be offered to the network as a service, allowing any interested clients to use the feature as needed. Should a client make use of the television, the service would be withdrawn from the network until the client had finished. What this example highlights is that future home network environment may become increasingly ad hoc, where services can be offered and withdrawn at a moment's notice. While having such a dynamic environment may seem to reduce the reliability given by fixed communication channels (e.g. the webcam always uses the lounge television), the network can be seen to be more reliable, as there are more alternatives should the first choice selection not be available. If the lounge television is unavailable, the user can move through to the study and use the computer monitor.

In realising this vision, the home network becomes a 'marketplace' of services, where providers can advertise their services, and clients can discover and browse desired services [25]. Like a marketplace transaction, service usage can take place when the service client can provide what the service provider requires. For example, suppose the lounge television requires DVD-format video input. The laptop webcam can only then use the service if it can provide the television with DVD-format input. In the marketplace, this transaction would be akin to the seller only accepting VISA. The buyer can only carry out the transaction if they have the means to pay with VISA. There can also be the possibility of services accepting multiple types of input (DVD, AVI, MPEG, WMV) similar to a market seller accepting multiple forms of payment.

Services may also produce output, and so the service client needs to be able to understand the data which is generated by the provider.

2.3 Comment on the Home Network Vision

As has been discussed, the purpose of the home network is a continually changing notion. While initially driven by simple goals of sharing files and internet connections, the vision has now grown to accommodate more complex goals, and, as a side effect, may have a more profound impact on everyday life. While the entertainment and simple home appliance usage provides an attraction, the home network can play a major role in enhancing and supporting lifestyles by providing various levels of care for those at home. Several research projects are concerned with this area, ranging from small scale monitoring, such as *busyness* and location monitoring [40, 85], to large scale systems [41, 39]. Home care networks are discussed in Section 2.4.

Regardless of the role which the home network may play, providing an infrastructure to support a dynamic and evolving environment provides compatibility issues amongst existing home network devices and protocols. For clients to be able to make use of services available within the network, they must be able to discover them. Using the example given in section 2.2 the laptop needs to stream the video data using the DVD-format. This input format may be particular to the category of television, or the laptop may only be able to output the video data in this format. If the lounge television accepts only AVI-format video data, but the laptop cannot provide this input, the laptop cannot make use of the television. This may be as a result of the laptop manufacturer overseeing the need for the laptop to provide this output, or the television manufacturer only allowing the television to accept this particular format.

In order for successful interaction in the home network, clients require to discover services and devices which they know they can interact with. This requires that the devices and service describe themselves, in order to allow clients to discover them. Clients may know in advance what services they wish to use, and the discovery process is a simple matter of finding out where the services are. This information would be found within the description of the service. On discovering the desired service, the client can then interact with the provider. Discovery and description processes become more complex as the number of standards, protocols and vendors within the network increase. This complexity can be highlighted using three examples: One where a single protocol and vendor exists within a network, one where two vendors exists (but share the same description protocol) and one where multiple vendors and protocols exist within a single network. These examples will reuse that given in section 2.2.

2.3.1 The Single Vendor Environment

On being instructed to play the video data, the laptop searches through the network for a 'Sony Television', in order to stream the DVD-format data to. On discovering that the television exists within the network, the laptop extracts the network location of the TV from the description and then begins to stream the data. The laptop can know beforehand that the television accepts DVD-format data as input, as both devices may have been developed within the same environment, given that they are provided by the same vendor.

2.3.2 The Two Vendor Environment

On searching for a Sony television, the laptop discovers that there are none in the home, but there is a Panasonic television. The laptop could naively follow the same pattern and extract the location from the description and begin to stream the data, but what if this model of television does not accept DVD-format video as input? At this point, the laptop could also check the description and extract the types of video input this television accepts. The Panasonic television accepts AVI-format video as input, which the laptop is able to provide. At this point, the laptop can now begin to stream the video data to the television.

2.3.3 The Multi-Vendor, Multi-Protocol Environment

On searching for a Sony television, the laptop discovers again discovers that there are none available. The laptop then searches for any other brand of devices which categorise themselves as a television, which again returns no results. The user then instructs the laptop to search for any computer monitors within the home and discovers one in the study. On discovering this monitor (of a different vendor), the laptop needs to work out how it can interact with the computer monitor, as it has only explicit knowledge of interacting with Sony televisions. A recent software update on the laptop added a component capable of utilising computer monitors for video output. The laptop passes the reference to the monitor to this component, and receives the relevant information back. This component then works on behalf of the laptop, interrogating the description of the monitor to extract relevant details (such as location and accepted inputs) and passes the information back to the laptop in a format it can understand. The laptop can now begin to stream the video data to the monitor.

In this environment, there are a few observations which can be made. Firstly, instead of seeing a monitor as a completely separate device from the television, it would be useful if the laptop knew about the similarities of the devices. A television produces video output, much like a monitor. The monitor and television are related in their purpose. If this were known about, it would not require intervention from the user to discover an appropriate device to use. The laptop could have searched for all instances of televisions *and* monitors, as it would know they provide the same function.

The second observation to be made is that the laptop required another component to assist it in carrying out the transaction. This was because the laptop did not know how to interrogate and extract information from the monitor

description (as it only knows about televisions!). At the description protocol level, the monitor was indeed distinct from the television at both the discovery stage, and at the descriptive stage. It would be useful if both devices share a common description protocol, as the laptop could have carried on the transaction without the need for the help of an additional component. (This is assuming that the communication stage was identical for both devices).

2.3.4 Descriptions within the Home Network

What becomes clear is that the descriptions of devices and services become more important as the heterogeniality within the network increases [76]. In an environment which is expected to be multi-vendor, as well as multi-protocol, continuity between the description protocols is essential in allowing services and devices to interoperate within the home network. There is a need of collaboration between developing protocols, as well as a reconciliation of existing ones, if such a vision is to be realised.

This need is not a small matter. There are varying approaches to providing descriptions which typically are determined by the platform, resources and level of intelligence available to the device or service. Some protocols are tailored for specific categories of network components (such as assistive technology, which can support those at home) while others are related to the vendor or developer (Apple devices use Apple-specified protocols). Through the variety of the potential home network environment, it may not be suitable to require all description protocols to adhere to a single specification. If this were the case, information relative to the specific domain of the component may be lost. Rather than specifying a rigid description schema, it is more useful to locate common features of component descriptions and attempt to unify these attributes.

As component descriptions move towards a common vocabulary or description approach, components can be described at a level higher than that of their protocol or vendor. Sony and Panasonic Televisions can simply be described as Televisions, with any vendor or protocol specific information merely becoming attributes of the description - rather than information which renders the devices logically distinct from each other. In a similar manner, network services can be composed at a higher level, being able to discover and use components based upon their function, rather than their protocol or vendor. For example, a DVD-playback service could perform discovery using 'Television' as the desired component, rather than 'Sony Television' or 'Bluetooth Video Playback Device'.

High-level services may make use of low level, single point devices or existing services. By combining these primitive network components, the home network can offer advantages greater than automated entertainment services or functions which reduce daily repetitive tasks. As mentioned in Section 2.3, home networks which can support care functions for those at home may provide a catalyst to realising the potential of the home network.

2.4 Home Care in the Home Network

Home Care Networks are attracting a substantial amount of both research and financial investment [69, 68, 96]. There are multiple definitions of a home care network, ranging from social to technical. From a social interpretation, a home care network refers to individual or groups of personnel involved in caring for someone at home. The network may involve doctors, nurses, health professionals, home help staff, wardens, first response persons, family members, neighbors and the home occupant. The idea behind this 'network' is that each person, or group, can communicate in some way with other persons or groups. For the purposes of this work, it is satisfactory to leave this definition of the social home care network as it is.

From a technical interpretation of the home care network, a user's home would include a number of networked devices and services which assist the user, the carers and other care personnel, in supporting the user within their own home [96]. This home care network could support many aspects of the health care service, from symptom management and patient monitoring [91] to community care and rehabilitation [104].

2.4.1 Motivation for the Home Care Network

Within the UK, the proportion of retired persons within the population is expected to rise. With the increase of retirees, the proportion of those who are working will fall [74]. An aging population, coupled with a proportional decrease in medical personnel, will result in more pressure being put on the health service resources [12]. As an alternative to living in care homes, many people are making use of assistive technology to allow them to remain independent, and in their own homes, for longer [60, 87]. Assistive technology typically consists of single point services and devices, such as a flood detector, an alarmed pull cord or neck pendant.

These single point devices can be integrated into a rudimentary network, allowing then to share a single phone or internet connection to a care-providing call centre. If a device is triggered, an alarm is sent to the call centre and a response person is on hand to deal with the situation. An alarm signal may include a device id, which corresponds to a description within the care providers database, allowing them to determine what kind of alarm has been received.

Home Care technology can be categorised into three *Generations*:

- First Generation: This group consists of simple single point devices, such as pull cords and neck pendant alarms, which respond to a direct action from a user (e.g. a pull, or press). Alarms from these devices are directed to response staff, such as a housing warden or social care worker for immediate response.
- Second Generation: This generation uses groups of simple sensors for ambient monitoring of the patient. Data gather from these sensors is used to detect potentially dangerous situations in the home where the user may not be able to raise an alarm (such as the user suffering a fall, or loss of consciousness). In the event of such a situation, the system generates the alarm on behalf of the user.
- Third Generation: This emerging generation of Home Care systems is responsible for a higher level of monitoring and data processing[39, 41], which may be more intrusive than that of previous generations. Third Generation systems attempt to predict problems and dangerous scenarios by means of monitoring many aspects of the user's life [72], such as lifestyle, exercise and diet. These systems can also offer assistance to users, supporting them in every day tasks. This generation is not confined to the home, as wearable sensors can be taken outside the home, with data being downloaded on return.

First Generation systems are currently installed in many sheltered housing complexes and residential care homes. The simplicity and non-intrusiveness of the devices removes many of the barriers to Home Care systems. Simple implementations of Second Generation systems can also be found in homes, as passive monitoring devices, such as flood and smoke detectors, can alleviate some of the risks faced by those who require care at home. More complex devices within the Second Generation tier, such as fall detectors, require the user to take a more active role in their care, by wearing sensors, or understanding meanings behind alarms (such as an inactivity alarm) [94]. Third Generation systems are still largely in the research stages.

2.4.2 The Emerging Home Care Network

Existing home care networks are largely alarm-based, that is to say, the network reacts to alarm events, such as an alarm cord being pulled, or a neck pendant being pressed. The need for a more comprehensive network is an emerging issue, which researchers believe can be supported by the home network. As discussed in Section 2.2, home networks are moving towards a collection of services within the home which can be used by a variety of clients. Within the emerging home care network, a similar environment can exist.

For example, a home owner has a security system installed which consists of a set of simple PIR sensors and a monitoring system which detects intrusions in the home. In order to monitor the mobility of the home user, an activity monitoring system is installed. This system detects and records activity taking place within the home, and provides a high level overview of this data to the community nurse. Rather than install new devices to detect motion activity, the monitoring system can simply use the outputs from the currently installed PIR sensors. Using the marketplace simile, the PIR sensors offer a 'motion detection' service to the network (of which, the security system is the priority client), and the activity monitoring system agrees to the transaction of data from the sensors, to its own system.

The integration of various existing assistive technologies, cumulating with the emerging home network provides possibilities for a more comprehensive network of support for all those involved in home care [86, 87]. Using webcams and home computers, doctors can hold remote consultations from their own offices, reducing the need to travel, and allowing for quicker detection of issues. In a similar fashion, community care nurses can handle a greater number of patients due to the reduced need to visit the geographical location of each patient.

It should be stated that the aim of the home care network is not to reduce the face-to-face contact between the user and health care personnel, but rather that entire care infrastructure is supported more efficiently. The support offered by the home care network would assist in correctly identifying those who require intervention quicker, reducing the number of unnecessary hospital referrals and supporting the independent living of those who are still able to remain at home.

2.5 Merging Aims of the Home Network

While the entertainment aspect of the home network may attract typical home network users to buy the technology (and therefore continue to drive research and development), the home care network may be the area where the most impact is felt. Approaches and technical developments aimed at providing the interactions discussed in Section 2.2 could be reused to support the provision of home care technology.

For example, suppose some services are developed which detects when the user arrives home, and switches on the hall light and re-routes all calls to the user's mobile phone to their home line. These services could be seen as luxuries which simply remove some of the burdens from the user. Suppose another user owns a home care network. These services can be modified to provide real benefit to the user. If the user gets up in the middle of the night, a service may switch on their bedroom light and a hall light, assuming the user is going to the bathroom. Similarly, if the user requires to make an emergency call, a service may route the call to the nearest location which contains a microphone and speaker.

While entertainment may drive the home network vision forward, the home care network has the potential to bring the home network into the everyday lives of users.

Chapter 3

Home Network Solutions

Various protocols and approaches have emerged as candidates for use in the Home Network. Some have been purpose built specifically for the home, while others have been leveraged from other environments. This section will briefly introduce some of the terminology used within the home network domain and within this work. This section will then discuss some of the main home network solutions, highlighting those used within this work, and providing discussion on the suitability of each, focusing on the service and device discovery techniques offered by these solutions, if this feature exists.

To conclude, this section will present two discussions of existing home network scenarios, highlighting the issues which require to be addressed within the home network domain.

3.1 Home Network Terminology

A **Service** is a function, action or operation offered to network clients. For example, a power service may be switched on or off. A data-storage service may save data in a certain location. An alarm service may notify listeners to interesting events. A **Service** is mainly software based, capable of being accessed by other software clients or agents. **Services** are in general offered by **Devices** and **Components**

A **Device** is an item of hardware that may offer or provides one or more **Services** to the network. For example, a Lamp device would offer a power service. An external hard-disk may offer a data storage service. A smoke alarm may offer an alarm service. A **Device** may also contain software agents which are designed to use **Services** within the network. A **Device** is networked, in that its services can be offered to the network, and any agents within the **Device** can act within the network.

A **Component** represents a less tangible concept within the home network domain. A **Component** may be a software container for one or more **Services**. For example, a Mathematical **Component** may offer computation **Services** to the network. The term **Component** can also be used to represent a logical grouping of a **Device** or software container and the services offered. For example, a network management **Component** may refer to a software module which offers network management services to a user through a physical user interface. In this manner, a **Component** may also refer to a **Device** or **Service**. It can be used as an abstract term to describe any instance of networkable item.

As mentioned above, this Chapter is concerned with describing home networking approaches, issues and solutions. A common term used in this discussion is **Protocol**. **Protocol** is a diverse term. It can represent the method by which network components communicate (i.e. the transport layer). For example, components may use the Bluetooth protocol or Firewire protocol to communicate. **Protocol** can also refer to the manner in which components describe themselves. In this instance, **Protocol** may refer to the vocabulary used by components, or an industrial standard which dictates the presence of attributes. Finally, **Protocol** can be used as a collective term to describe all aspects of component behaviour, including communication and description. This work refers to this definition of **Protocol** when the term is used.

Using these definitions, the rest of this chapter is concerned with discussing popular home network solutions and scenarios.

3.2 Universal Plug and Play

Universal Plug and Play, or UPnP as it is commonly known, is a set of standards for component addressing, information exchanging and service usage within a networked environment [90]. This environment may be small and local, such as the Home Network, or large and wide, such as a company network, or even the Internet. The architecture is built upon common internet technologies, utilising TCP/IP, HTTP and XML. This allows UPnP to offer a driver-free environment where transactions are carried out using common protocols rather than device or vendor specific protocols. In this manner, existing devices can also 'plug into' the UPnP architecture, as long as they adhere to the UPnP Device Architecture (UDA). The UDA specifies how devices can join and leave UPnP networks, as well as how they can communicate with other enabled devices. The UPnP architecture is suitable for both managed and unmanaged networks.

UPnP networks consist of Service Providers and Control Points. Service Providers are devices which offer services to the network, e.g. a Printer. Control Points are devices which can use services, such as a Digital Camera (which can use a Printer). UPnP devices may contain both Service Providers and Control Points. The UPnP Architecture also allows for UPnP Devices to contain embedded UPnP Devices. For example, a UPnP Stereo may contain a CD Player and a Radio. The UPnP Architecture can be broken down into six main points concerning a UPnP Device:

- Joining the Network
- Device and Service Descriptions
- Discovering other Devices
- Controlling Devices and Services
- Eventing in the Network
- Presentation

This work shall concentrate on the discovery and descriptive parts of the architecture.

3.2.1 Joining the Network

For a device to participate in an UPnP network, it requires an IP address. The UPnP Specification utilises the Dynamic Host Configuration Protocol (DHCP) for assigning addresses to the network. If the network is managed, the local DHCP server assigns an address to the client. If the network is unmanaged, i.e. there is no DHCP server, the client retrieves an address through Auto IP. Auto IP is a process where DHCP clients can gain a network address by a series of communications with the network.

3.2.2 Device and Service Descriptions

Descriptions within a UPnP network take the form of an XML document which captures important features of the device. A typical device description document takes the form of first specifying the version of the UPnP protocol used, and then stating attributes of the device, such as device type, device name, manufacturer, reference and unique identifications and finally a list of services offered by the device. An excerpt describing attributes of an UPnP Air Conditioner is shown in Figure 3.1

Descriptions of services contained within the device description contain only a few details: A service type, service ID and URLs to where interested clients can learn more about the service capabilities (from the SCPDURL), possible subscriptions (from the eventSubURL) and the service endpoints where control points can invoke actions (from the controlURL). Services offered by the UPnP Air Conditioner are described in Figure 3.2

This level of detail is enough for prospective clients to decide if this UPnP device contains the functionality they require. If a UPnP device does not contain the services desired, it can be ignored. If the device does contain the correct services, a client can then examine a more detailed service description at the location specified by the SCPDURL element.

```
...
<deviceType>urn:schemas-upnp-org:device:aircon:1</deviceType>
<manufacturer>CyberGarage</manufacturer>
<manufacturerURL>http://www.cybergarage.org</manufacturerURL>
<modelDescription>CyberUPnP AirCon Device</modelDescription>
<modelName>AirCon</modelName>
<modelNumber>1.0</modelNumber>
<modelURL>http://www.cybergarage.org</modelURL>
<serialNumber>1234567890</serialNumber>
<UDN>uuid:cybergarageAirConDevice</UDN>
<UPC>123456789012</UPC>
...
<presentationURL>http://www.cybergarage.org</presentationURL>
...
```

Figure 3.1: CyberGarage Air Conditioner

```
...
<serviceList>
  <service>
    <serviceType>urn:schemas-upnp-org:service:power:1</serviceType>
    <serviceId>urn:schemas-upnp-org:serviceId:power:1</serviceId>
    <SCPDURL>/service/power/description.xml</SCPDURL>
    <controlURL>/service/power/control</controlURL>
    <eventSubURL>/service/power/eventSub</eventSubURL>
  </service>
  <service>
      <serviceType>urn:schemas-upnp-org:service:temp:1</serviceType>
      <serviceId>urn:schemas-upnp-org:serviceId:temp:1</serviceId>
      <SCPDURL>/service/temp/description.xml</SCPDURL>
      <controlURL>/service/temp/control</controlURL>
      <eventSubURL>/service/temp/eventSub</eventSubURL>
  </service>
</serviceList>
...
```

Figure 3.2: Services Provided by the Air Con

```
...
<actionList>
    <action>
    <name>SetPower</name>
    <argumentList>
      <argument>
        <name>Power</name>
        <relatedStateVariable>Power</relatedStateVariable>
        <direction>in</direction>
      </argument>
      <argument>
        <name>Result</name>
        <relatedStateVariable>Result</relatedStateVariable>
        <direction>out</direction>
        </argument>
    </argumentList>
  </action>
...
</actionList>
...
```

Figure 3.3: Services of the Air Conditioner

Detailed Service Descriptions

A UPnP service description contains a list of actions, along with a set of variables associated with the actions. An action is an invocation of a particular service. For example, a service list may contain a 'SetPower' action which allows a control point to switch a device on or off. With each action comes a corresponding list of arguments used by the action. These arguments may be inputs (e.g. a power value) or an output (e.g. the success of the action), and a service may have one or more arguments for each action. A state table contains a list of state variables associated with the service. For example, a Power service may contain a variable, called Power, which refers to the current state of the Device. These variables may be private to the Service, such as the result of an action, while others may be available to interested clients.

An excerpt from the description of services offered by the UPnP Air Conditioner is shown in Figure 3.3.

3.2.3 Discovering other Devices

All discovery and advertising within a UPnP network uses the Simple Service Discovery Protocol (SSDP), which is a protocol specified within the UPnP Architecture.

SSDP

SSDP specifies headers for messages between providers and clients in the discovery process. SSDP messages are broadcast to the network, either by control points requesting services, or by services replying to requests. The broadcast address specified by SSDP is 239.255.255.250, and on port 1900. By broadcasting messages, all control points within the network are guaranteed to receive each advertisement of services. Similarly, ensuring all service providers receive every service request provides the maximum return of relevant services. As an aside, the UPnP specification also contains rules governing the volume and transmission of messages on the broadcast network. This set of rules ensure that the network is free from devices 'spamming' SSDP messages.

There are three main types of SSDP message used within the UPnP network:

- Notification messages - Used when joining and leaving the network
- Search messages - Used during discovery
- Response messages - Replies to search messages

3.2.4 Joining a network

Once a UPnP device has obtained a network address, it broadcasts a series of messages, advertising its capabilities to control points within the network. This takes the form of a number of SSDP discover messages designed to announce the availability of the device, and its services, to the network. An SSDP announcement message contains four main attributes:

- A notification type. This is usually the classification of the device or service being announced.
- A unique identifier for the message.
- A URL where interested control points can query for more information about the sending device.
- A time-out specifying how long the message is valid for.

This message is classified as a ssdp:alive message, due its nature. The message announces that the service or device is now available.

3.2.5 Leaving a network

When leaving a network, a UPnP Device must announce its removal to ensure that other components are aware of its unavailability. A leaving message has three main attributes:

- A target recipient.
- A notification type. When leaving a network, this attribute has the value **ssdp:byebye**.
- A Unique Service Name (or USN).

The USN value is the name of the service or device being removed (which is an instance of the type of service or device). The type of notification is always ssdp:byebye, which indicates that this is a leaving message. More specifically, a leaving message is a revoking of a ssdp:alive message. For each ssdp:alive message broadcast on joining the network, a corresponding message must be sent to revoke the announcement. A leaving message has no lifetime (once broadcast, it is assumed the device has left the network). Both joining and leaving messages contain 'NOTIFY * HTTP/1.1' as the initial message header. This indicates that this kind of message is a notification to the whole network.

3.2.6 Discovery in the Network

Discovery within the UPnP network also uses SSDP. A search message contains three important elements:

- A maximum time for the validity of the search message.
- A search target.
- A notification type

A Control Point provides a maximum time value to ensure that it can handle all responses to the search. Devices which respond to a search message choose a delay between 0 and the maximum time before responding to the message. In this manner, the Control Point is not flooded with responses instantaneously. The search target is the desired service or device type. This target can be generic (e.g. find all devices) or specific (e.g. find an Air Conditioner). This type of message is categorised as a ssdp:discover message. An example search template is shown in Figure 3.6.

3.2.7 Replying to Searches

Rather than replying to the Control Point performing the search, candidate devices simply re-broadcast their capabilities to the network. The capabilities which are re-advertised are those which match the search message. For example, a Digital Camera may search for a Printer device. A Device capable of printing, scanning and faxing can reply to the search message, but only announce its printing capabilities. A response message takes a similar format to that of a ssdp:alive message. The main difference is that instead of a notification type, a search target (ST) header is used instead, denoting the particular device or service witch is responding.

```
M-SEARCH * HTTP/1.1
HOST: 239.225.225.250:1900
MAN: "ssdp:discover"
MX: (seconds to delay response)
ST: (search target)
```

Figure 3.4: A Discovery Message Template.

3.2.8 Controlling Devices and Services

As previously mentioned in section 3.1, only UPnP Control Points can invoke services within the network. After a UPnP Control Point locates a desired UPnP component, it can control the device or invoke the service by sending action requests to the component. A Control Point can gather information about how to interact with the component by parsing the service descriptions provided. On finding a suitable service, and constructing a valid control message, the service user passes the control message to the provider. The provider then carries out the action associated with the service, and may return a similar message back to the user. This message may contain the result of the action (such as a current variable value) or any error states.

3.2.9 Control Messages and SOAP

Control messages are formed using Simple Object Access Protocol (SOAP) [101]. A SOAP message encapsulates all necessary information required by the service provider to carry out the action. For example, if a service description states that a power service requires a **POWER** variable as input, the SOAP message will contain such a variable, along with a desired value (e.g. POWER = 1). On receiving a SOAP message, the service provider carries out the action, and may return a different SOAP message which encapsulates any output from the service (such as variable values or errors).

Protocol-Free Interaction

Using SOAP allows service invocation to be independent of any particular platform or protocol which the service provider is using. This approach is in keeping with the UPnP philosophy of using common protocols for all aspects of discovery and control. It also provides scope for developers to enhance their existing products to be UPnP enabled.

3.2.10 Events within the Network

As discussed in Section 3.2.2, service descriptions contain a URL which Control Points may subscribe to in order to receive events from that service. The state table within a service description can contain a number of variables which may be of interest to other parties. When these variables change, an event message is sent out to all subscribers containing the new value of the variable. For example, a Power Monitoring device may wish to subscribe to any changes in the power state of the Air Conditioning. If a Control Point requested that the Air Conditioning be switched on, the Power Monitoring device would receive this new state of the Power variable (e.g. 1). A state table may contain more than one variable which can be observed. In such cases, any changes to any observable variables results in the current values of all such variables being sent to subscribers.

State messages, which contain the state table, have a SSDP-like header which contains information about the message type and target. The body of the message is an XML representation of the observable state variables. A state message does not contain state variables which are not observable. Such variables may be able to be retrieved through a simple variable value request.

3.2.11 Presentation

Device descriptions may contain an attribute while provides a presentation URL. The presentation URL points to a location where a graphical representation of the device can be found. This representation may provide a visual overview of the current state of the device. For example, an UPnP House Light Controller may provide a graphical representation of the current state of all lights within the home, allowing the user to quickly view which lights

were on and off. Any presentation provided by a UPnP Device is intended for human users, allowing a natural interface into the Device.

3.2.12 The Suitablilty of UPnP

UPnP provides an attractive option for home networks. Compliant devices need only implement the UPnP architecture in order to participate within an UPnP network. Devices are free to operate on any platform or protocol they wish, providing they adhere to the UPnP architecture while interacting within the network. For this reason, UPnP is attracting interest from research projects in a number of home network applicable domains [92, 66]

3.3 Home Audio/Video Interoperability (HAVi)

The Home Audio/Video Interoperability (HAVi) architecture is concerned with incorporating typical home entertainment devices, such as televisions, stereos, and digital television receivers, into a home network [50]. The HAVi specification can be applied to a range of existing and emerging devices, being designed to be both future proof and incorporate legacy devices. HAVi is independent of any platform or vendor specific implementations, instead defining a set of Application Programmable Interfaces (APIs) which devices may offer to other devices. Such APIs allows devices to interoperate with any resources available in the network, allowing late device binding (i.e. devices can use any available instance of a particular device category, rather than a specific implementation). This section will describe the descriptive and discovery approaches specified by the HAVi protocol.

3.3.1 Controllers, Devices, DCMs and FCMs

The HAVi architecture distinguishes between two main components within a Home Network, Controllers and Controlled Devices.

- A Controlled Device is a device which provides functionality, such as a Visual Display
- A Controller is an interface into a Controlled Device, for example a Control Panel on front of the Visual Display

Controllers and Controlled Devices may be part of the same logical device, or they may exist on separate devices (e.g. a Remote Control and a Television). A Controller may be responsible for one or more Controlled Devices. A Controller hosts a Device Control Module (DCM) for each Controlled Device. A DCM is a set of APIs which other devices or applications use to control the Controlled Device.

In order to be portable DCMs need to be independent of the platform of the using device or application. DCMs are generally sets of Java code complied into bytecode. The Java language is used because of its '*Write Once, Run Anywhere*' properties. The bytecode is able to be run on any platform which contains a Java Virtual Machine (JVM).

Devices and applications download DCMs from the Controller in order to discover the functions of the device. The DCM acts as a proxy between the user and the Controlled Device. DCMs may contain a standard set of functions, defined by the HAVi specification, or a mixed set of standard and specialised functions. Specialised functions may be particular to a set of devices, for example vendor-specific devices, which may limit the range of users of the particular device (as users require to know the interface of the DCM in advance).

Each DCM owns one or more Function Control Modules (FCMs), with one FCM for each function or service offered by the device. An FCM is an abstract representation of a service, and is used by service users to interact with the service. As the DCM acts as a proxy to a Controlled Device, a FCM acts as a proxy to a service on that Controlled Device. The relationship between a Controlled Device, DCM and FCM is shown in Figure 3.5.

3.3.2 Device Classification

HAVi specifications contain scope for four main groups of Audio/Visual (AV) devices:

- Full AV devices (FAV) - These devices are rich in terms of resources and capabilities. FAV devices contain a JVM, and so can control any device which adheres to the HAVi architecture (Those which offer a DCM). FAV devices can act as co-ordinators for a HAVi network, offering the functionality of other devices to the network (through use of DCMs).

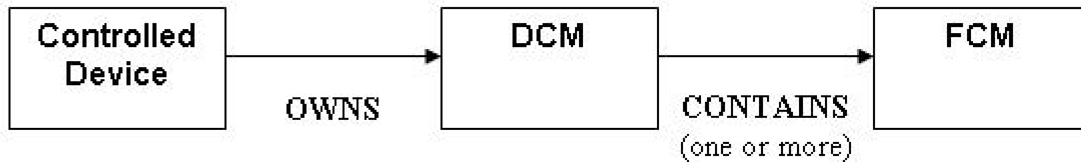


Figure 3.5: Relationship of the Device, DCM and FCM

- Intermediate AV devices (IAV) - These devices do not contain a JVM, and so cannot download and utilise DCM code. IAV devices may, however, contain predefined code which allows them to use a set range of devices. IAV devices may also offer the functionality of these devices to the network
- Base AV devices (BAV) - These devices are the most basic HAVi devices available. They have no capabilities for using other HAVi devices, but still offer a DCM, which allows them to be used by FAV devices
- Legacy AV devices (LAV) - These devices are not HAVi enabled, and are largely independent from the Home Network. FAV and IAV devices may work on behalf of LAV devices, bringing them into the HAVi network. In such cases, the HAVi component translates HAVi commands into those understood by the LAV device

3.3.3 HAVi Networks

As mention in section 3.3.2, FAV and IAV devices can offer the functionality of other BAVs and LAVs to the network. The HAVi specifications suggest that Home Networks will, in general, consist of clusters of devices, with one main device acting as co-ordinator for all other devices and services within the cluster. A cluster may represent devices in a single room, or per housing level. FAV and IAV devices act as the co-ordinator for clusters, corresponding with other co-ordinators within the network. (It should be mentioned that the HAVi architecture can also support cluster-free networks). An example cluster may be the Living Room cluster. In this cluster, the Digital Television Receiver is a FAV device, and offers its own services to the network. The co-ordinator of a cluster is responsible for providing the services within the cluster to the rest of the network. As coordinator for the Living Room, the Digital Television Receiver may also offer the services of the BAV Television and BAV stereo system to the network. The co-ordinator is responsible for managing the DCMs for all devices within its cluster. Figure 3.6 depicts a example HAVi network, containing two clusters of devices.

3.3.4 Descriptions and Discovery

Discovery with HAVi networks relies on Registry components. Each FAV or IAV device contains its own Registry, which maintains a list of services which the co-ordinating device can provide. Services within the registry may be explicitly owned by the device, or implicitly offered, on behalf of BAV or LAV devices in the cluster. In this manner, each cluster will have a registry, responsible for providing consistent and relevant information about the services within the cluster.

Each service in the cluster is represented by a Software Element within the Registry, along with a list of system attributes. A Software Element contains a well-defined interface for users to interact with the service, with the service being an implementation of that interface. The HAVi specification defines a set of system attributes which can be attached to a Software Element. Having a defined list increases the continuity between devices of differing developers and vendors. A Software Element contains a list of system attributes and a Software Element Identifier (SEID), which uniquely represents the element with the network.

Applications can query the Registry of a device to discover desired services, and then retrieve the associated DCM and FCM from the Controller. Querying requires the application to construct a template of a desired Software Element, complete with a set of required attributes. This template is then submitted to the Registry, with the SEID of any matching services being returned to the application. The application can then identify the relevant device and download the DCM from the device.

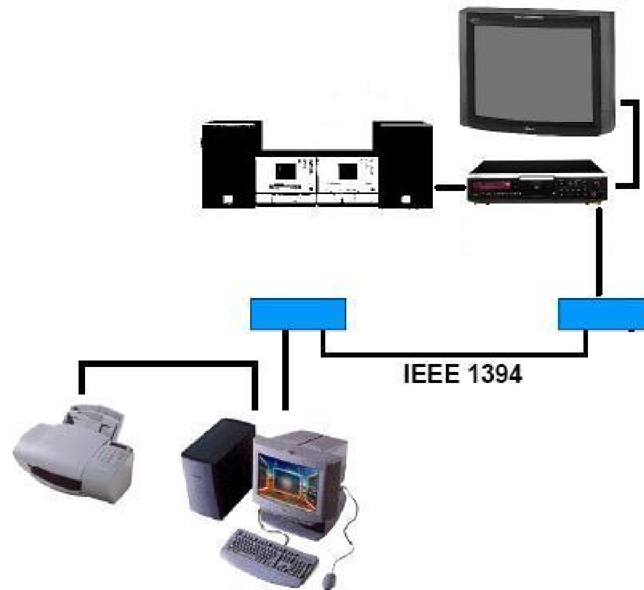


Figure 3.6: An Example HAVi Network of Clusters

3.3.5 Communication

The main transport protocol used within HAVi networks is the IEEE 1394 interface, commonly known as FireWire. This protocol is well suited to high speed communication and data transfer. Some clusters, such as those managed by IAVs, may use specific protocols for communicating with LAV devices, which may not support the IEEE 1394 interface.

All messages sent within the HAVi network contain a SEID, which acts as an recipient address for the message. Messaging within the network is managed by one or more Message Systems, a component present on all FAV and IAV devices. A FAV or IAV therefore also acts as the message co-ordinator for the cluster it is a member of. The Message System is responsible for assigning SEIDs to software elements within its domain (cluster). This SEID is the same attribute used within the registry process.

The Message System also contains functions for removing SEIDs from the network (i.e. unregistering the element from the network), validating elements for trust purposes, and co-ordinating 'call backs', which are responses to messages from Software Elements containing any success or error messages.

3.3.6 The Suitability of HAVi

HAVi presents an interesting choice for a home network environment. Like UPnP, the HAVi architecture contains properties which can be applied to network components which are not native to the protocol [71]. FAV devices are able to act as proxies for other devices which do not offer the architecture. In this manner, HAVi clusters may be composed of many non-HAVi components with FAV devices propagating HAVi commands from the network into commands understood by a non-HAVi device. While this is undoubtedly a useful and unique approach to providing a common protocol for a home network, it may be difficult to include existing home appliances to the protocol. High-speed communication is crucial for communication between multimedia devices. It would be difficult for this communication to take place between HAVi and non-HAVi devices, such as between a HAVi compliant DVD player and a standard television.

3.4 X.10

The X.10 protocol is an open industry standard, designed to support the automation and control of devices. X.10 supports devices which are simple on/off components in terms of functionality, such as lamps, ceiling lights, fans

Figure 3.7: An X.10 Appliance Module

and motors. An X.10 network consists of Controllers which command X.10 enabled devices, and Modules which allow every day powered devices to ‘plug in’ to the protocol.

3.4.1 Plugging into X.10

X.10 is well suited to the home domain as the transport medium used by the protocol is already installed in every home. The X.10 protocol uses power lines to communicate between Controllers and Modules, using radio frequency signals to transmit data. Both Controllers and Modules plug into standard power sockets, with devices then plugging into a Module to become X.10 enabled. Figure 3.7 shows a typical X.10 Module.

Addressing

Each X.10 device has a unique address within the network. This address is a composition of a house code and a device code. A house code is a letter between ‘A’ and ‘P’, while a device code is a number between 1 and 16. For example, if the Living Room had house code ‘G’ and the Television had device code 8, the X.10 address of the device would be ‘G8’. This addressing system allows for 256 unique addresses (16 house codes 16 device codes) within a single network.

3.4.2 Control

A Controller sends control signals through the power line to the Modules in the network. A control signal consists of a X.10 address and a command, for example ‘F5 ON’. A Controller can specify multiple devices before the command, for example ‘F5, G8, A6 ON’, allowing a single command to be applied to multiple devices. A Module will listen for control signals which apply to its address, and then carries out the command specified. In general, X.10 is based upon a peer relationship, with Modules simply listening for commands from Controllers, unable to output any data of their own, such as status (e.g. ‘F5 is OFF’). Commands are usually limited to ON and OFF, but some specialised Modules can understand other simple commands, such as dimming instructions for lights.

3.4.3 Discovery

Unlike the protocols discussed so far, X.10 has no capability for discovering devices or appliance modules within the network. X.10 controllers have no scope for discovering if any appliance modules existing on the home power line, or to retrieve any state information about any modules present. Controllers simply send on/off commands to specified device addresses with no means to ensure these commands are received. Despite these limitations, X.10 provides a simple solution to networking existing home appliances, and integrating them into the home network.

As an aside, while a description-less protocol such as X.10 would seem to be of little use within a home network environment, similar protocol specifications, such as EIB and KNX (www.knx.org), are seeing large investment and utilisation within the smart homes domain. X.10 has also been utilised within existing home care projects [64, 41]. Using these protocols, the robust aspect of component discovery is traded for a reliable, lightweight communication protocol which can be easily integrated into everyday appliances, such as light

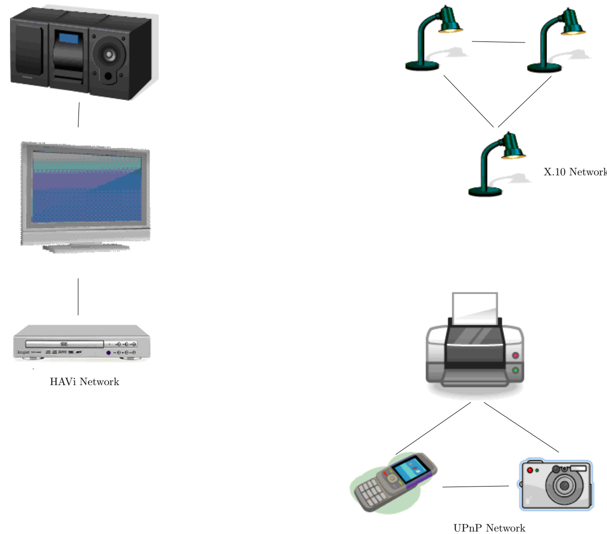


Figure 3.8: Muti-protocol Home Network

switches and air conditioners. Rather than a 'market-place' of resources, appliances are programmed with explicit knowledge of what devices to use, and how to interact.

3.4.4 Issues in X.10

Using power lines as a transport medium causes potential issues within the home. If homes share power line networks, it can be difficult to restrict X.10 control signals to a single domicile. Signals which travel outside the intended home may interfere with X.10 devices on other home networks. The lack of standardised control commands limits the potential use for X.10 to simple on/off devices (such as lights) and safety cut outs (for example, on cookers and electric fires).

3.5 Interoperability in the Home

As this chapter has demonstrated, there are a choice of protocols designed for home networking. As discussed in section 2.3.1, the Home Network using a single protocol avoids numerous compatibility issues. Some protocols are designed with a specific purpose in mind. For example, HAVi was designed for audio/visual devices, and X.10 for on/off powered devices. While the scope of protocols can be extended to incorporate other devices, it is more feasible that a Home Network will contain a number of protocols, all chosen for a specific reason.

Figure 3.8 shows a hypothetical Home Network, where audio/video devices lie within the HAVi network, the house lighting system is controlled through X.10 and other multimedia devices utilise the UPnP protocol. This Home Network contains three distinct networks which are unable to interact with each other. The television has no knowledge of the house lighting system, while the UPnP enabled mobile phone does not know about the presence of a networked stereo system. Section 3.6 describes a home network discovery scenario in such an environment.

3.6 A Naive Home Network Scenario

In an environment where a client service receives no discovery support from the network, the client requires to make a number of interactions with the environment when performing discovery. It is assumed that the client can directly or indirectly interact with any protocol within the network. An initial difficulty arises from the nature of the home network. The environment is unbounded in the number of possible protocols which can exist. Each potential protocol represents a new interaction the client must make with the environment.

For example, suppose there are three protocols within the home environment. [1]

[1]Throughout this chapter, the three protocol example is reused to represent the three main approaches to descriptions identified within Chapter 3.

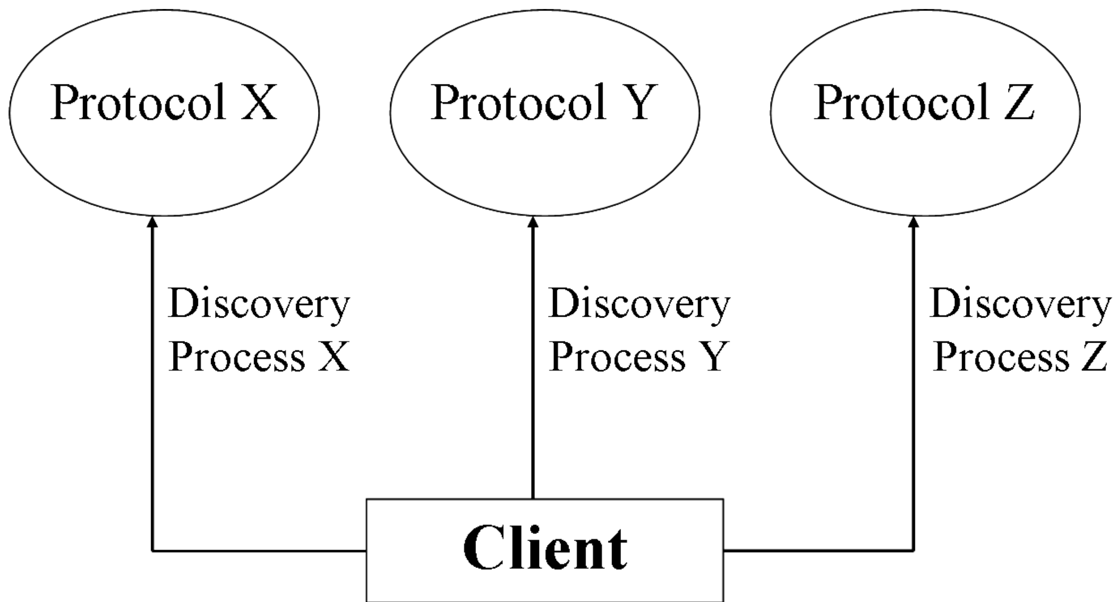


Figure 3.9: A Naive Home Network

Taken from the scenario above, a client service wishes to discover suitable lamp devices in home (or relevant devices which can provide light). For each protocol, the client requires to know how to perform discovery in that domain. In this example, the client requires to know about discovery in three domains, and is required to explicitly interact with each domain to discover any suitable devices, shown in Figure 3.9. An initial (and perhaps naive) assumption can be made in that for in an environment where x number of protocols exist, a client would require to know x number of different ways to perform discovery.

It is not unconceivable that within a multi-protocol domain, two protocols may share a similar approach toward discovery. For example, a protocol may adhere to a standardised approach toward description, while having a unique approach to execution. This possibility *may* improve on the many-to-many relationship in terms of discovery, but a worst-case scenario would still involve a x:x relationship to be resolved.

The main issue with home network descriptions still remains unresolved. Vocabularies and terminologies used by protocols remain largely heterogeneous. In an home network which is unsupported by middleware for discovery, clients have a heavy responsibility to know in advance the terminologies to be used for discovering desired components. Protocols may be described through schemas, attributes or less structured forms of vocabulary, and semantically similar concepts may be described in distinct ways. To this end, to discover a lamp device (or similar), the client service will require a specific discovery message for each potential protocol based on specific terms. In our three protocol example, the client service will require to construct three separate messages based on each protocol's description approach, shown in Figure 3.10.

In essence a new relationship emerges. The relation between the number of protocols in the environment and the distinct messages which the client requires to know to discover also becomes an x:x relationship. For every new protocol added into the environment, a new discovery message format may be required in order to perform true cross-protocol discovery. For each component the client wishes to discover, x number of discovery message may be required, or a 1:x ratio between desired component and messages required. For example, if a client wishes to discover a Lamp, every protocol within the network may use a different vocabulary set and description format. A client must take into account every possible protocol, and construct a discovery message accordingly.

In summary, two issues require to be addressed to ensure real-world implementations of a home network that match the characteristics demanded by the home network vision, presented in Section 2.2. These issues are:

- A reduction of the x:x relationship between numbers of protocols and methods of discovery needed to be known by a client.
- A reduction of the 1:x relationship between the service type the client wishes to find and the number of possible messages or vocabularies needed to discover the service in each protocol domain.

It may be suitable that the networks described in Section 3.5 remain distinct and have no interaction with each

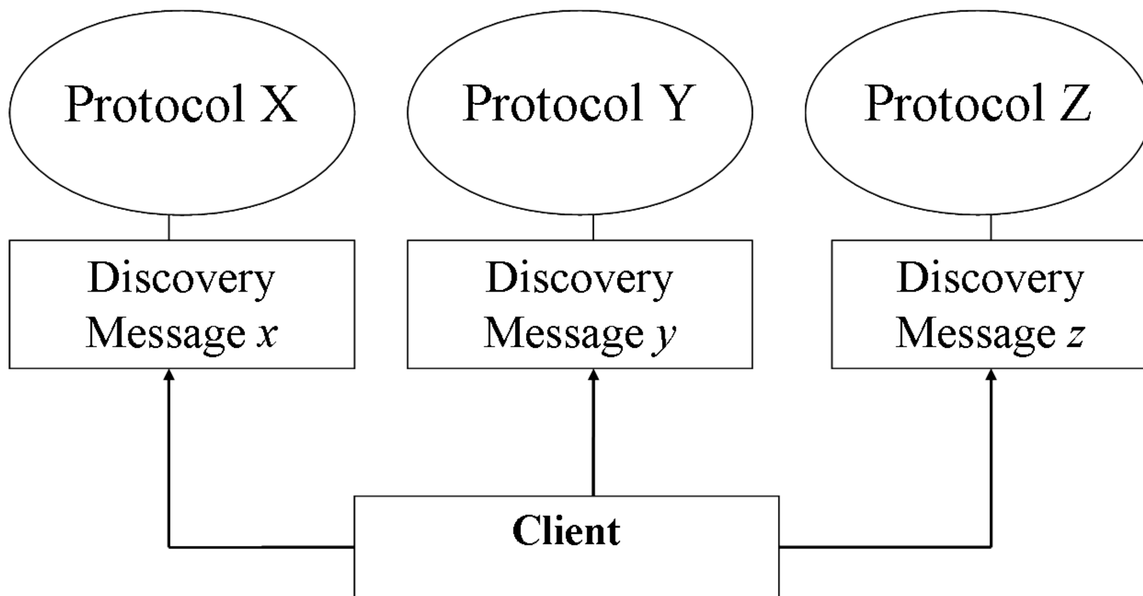


Figure 3.10: Naive Discovery

other, but this would seem to contrast with the goals of the Home Network, namely to enable devices and services within the home to interoperate. In a natural language environment, if two persons wish to converse, but do not speak the same language, an interpreter is used. The interpreter listens to dialogue in one language, and then translates the dialogue into another language. In the case of network protocols, a similar method can be used.

3.7 Protocol Bridges

For one protocol to interact with another, translation from one protocol to another is needed, and is commonly known as a protocol bridge (as it bridges between two protocols). Using the example in Figure 3.8, a UPnP-to-HAVi bridge could allow the UPnP enabled phone to stream music to the HAVi stereo, shown in Figure 3.11. Protocol bridges require to know how to interact on both sides of the bridge. The extent to how the bridge acts is usually determined by the purpose of the bridge.

For example, the purpose of the UPnP-to-HAVi bridge may be to allow UPnP devices to control HAVi devices. To achieve this, the bridge must have knowledge of how to perform service discovery within the HAVi network, must be able to understand the messages involved, and must know how to convert UPnP commands into HAVi commands. The bridge may have a further purpose, that of offering HAVi services to the UPnP as UPnP services. To achieve this, the bridge must, in addition to the requirements already specified, act as a HAVi component, listening for new services joining the HAVi network, translate their descriptions into the UPnP format, and then announce them to the UPnP network as UPnP services.

Protocol bridges may be uni or bi directional, with the UPnP-to-HAVi bridge discussed being uni-directional. (The bridge acts on behalf of the UPnP network). A bi-directional bridge acts on behalf of both networks, translating HAVi commands into UPnP commands, and vice versa.

Software components, called Middleware, can be used to support protocol bridges in this task. Middleware

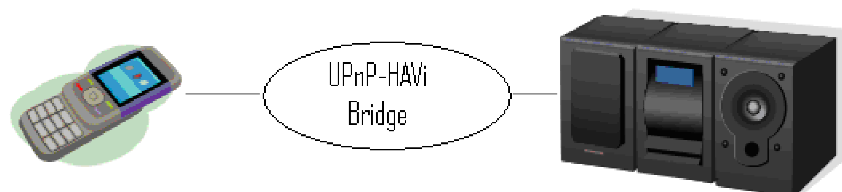


Figure 3.11: Using a UPnP-to-HAVi Bridge

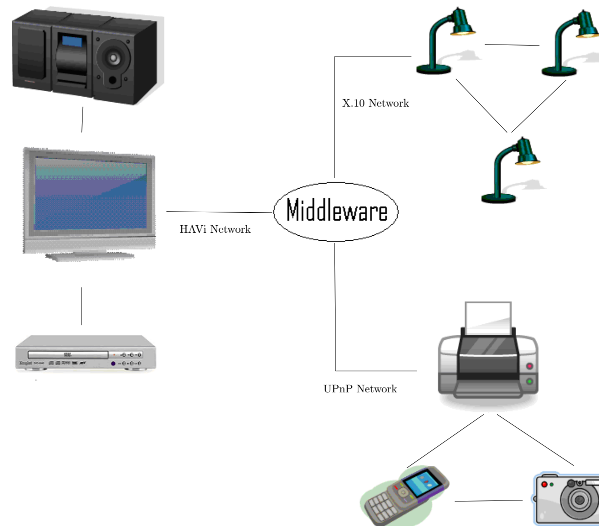


Figure 3.12: Middleware in the Home Network

can provide computational resources, as well as maintenance and management functions. Multiple bridges can share the same middleware environment, which leads to the Middleware becoming a central component to the Home Network (see Figure 3.12).

3.8 Middleware

As has been shown, middleware can play a pivotal role within the Home Network. In addition to housing protocol bridges, middleware environments can also offer a programmable interface for users to compose their own services. For example, suppose a user wishes to make use of the functions of their HAVi DVD player and their X.10 lights. Every time a DVD is played, the X.10 lights within the room should dim or switch off. To make this possible, the Middleware within the network would require bridges into both HAVi and X.10. The user would create instructions using tools provided by the middleware to control the HAVi and X.10 devices.

This scenario highlights an important aspect of middleware. Middleware contains many of the properties which characterise home networking protocols, such as a programmable environment and classification of 'types' of devices and services. For example, to provide the service composition described above would require the middleware to contain the concept of a HAVi DVD service, and an X.10 Light. In this manner, some home networking middleware platforms can be considered a protocol within their own right. These middleware platforms may have their own system of service and device discovery.

The Open Services Gateway Initiative (OSGi) is a prominent middleware platform which is attracting interest from the home network research domain [3, 24, 61].

3.9 OSGi

The Open Services Gateway Initiative specifies an architecture for creating, managing and deploying Java-based services within a single framework or runtime environment [67]. Services within the framework share resources, and may themselves share resources with the rest of the framework. In this manner, services can be both providers and clients. In simple terms, an OSGi service is an implementation of a well known interface. A service within OSGi is offered by an OSGi 'bundle'. An OSGi bundle can offer zero or more services to the framework. (A bundle which offers no services may still be offering other resources, such as libraries, to the framework).

A typical OSGi framework offers a number of features, such as service registration, security, http support and framework recovery. The OSGi framework also manages the life cycle of all bundles.

3.9.1 Life cycle Management

The life cycle of a bundle has six main states:

- Installed: This state represents a bundle which has initially been installed into the framework, awaiting its dependencies to be resolved.
- Resolved: This state represents a bundle which has had all its dependencies resolved. Bundles remain in this state until started by the framework.
- Started: This state represents a bundle which has been started by the framework. The state of this bundle immediately moves onto the Active state.
- Active: This state represents a bundle which is currently acting within the framework. The bundle remains in this state until it is stopped.
- Stopped: This state represents a bundle which has its dependencies resolved, and has been active within the framework, but is no longer active.
- Uninstalled: This state represents a bundle which is no longer part of the OSGi framework.

The OSGi framework maintains the state of each bundle in a persistent manner, allowing the framework to recover easily in cases where a restart is needed. If a restart of the framework is needed, each bundle does not need to re-register themselves, as the framework assumes their states are unchanged.

3.9.2 Services And Bundles Within The OSGi Framework

A bundle encapsulates all required OSGi information within a manifest file.

```
Manifest-Version: 1.0
Bundle-ClassPath: ., libs/Keyboard.jar
Bundle-Version: 1.0.0
Bundle-Name: DisplayManager
Bundle-ManifestVersion: 2
Bundle-SymbolicName: DisplayManager
Bundle-Activator: uk.ac.stir.cs.lsd.providers.display.impl.Activator
Bundle-Description: Manages Swing Interfaces
Bundle-Vendor: Liam Docherty
Import-Package: javax.swing, org.osgi.framework
Export-Package: uk.ac.stir.cs.lsd.providers.display
```

Figure 3.13: An Example Bundle Manifest

A manifest file contains simple properties of the bundle, such as developer and bundle version. The manifest also states the required Java libraries for the bundle, denoted by the bundle classpath. These libraries are already contained within the bundle, and are loaded into the framework along with the bundle. The final part of the manifest states which resources are required by the bundle in order to be started, and what resources from within the bundle are to be shared with the framework.

Using the example shown in Figure 3.13, the Display Manager bundle requires both the javax.swing and org.osgi.framework libraries to be available in order to be started (denoted by Import-Package). These libraries are the dependencies of the bundle. The Display Manager bundle then exports the library uk.ac.stir.cs.lsd.providers.display to the framework (denoted by Export-Package), which can then be used by other bundles within the framework.

One important property not yet covered is the Bundle-Activator property. This property instructs the framework where the Activator for the bundle can be found. Bundle Activators are found within all bundles which offer services to, or use services within, the OSGi framework.

Interface	Implementation	Properties
LightInterface	KitchenLight	Location=Kitchen
LightInterface	BedroomLight	Location=Bedroom
LightInterface	LivingLight	Location=LivingRoom

Figure 3.14: The OSGi Service Registry

3.9.3 The Bundle Activator

An activator is the first point of contact for the OSGi framework after a bundle has been started (and, only if an activator exists!). Within its activator a bundle may register services with the OSGi service registry, or search the registry for required services (or both!). These processes will be covered in sections 3.9.4 and 3.9.5. The bundle activator is also called by the OSGi framework as the bundle is moved to the stopped state. This allows the bundle activator to perform operations, such as unregistering its services, or removing links to external services, ensuring that the bundle can leave the framework cleanly.

3.9.4 Registering Services

The OSGi registry maintains a list of services (or implementations of interfaces) along with a set of attributes of that service (as shown in Figure 3.14). For a bundle to register a service with the registry, it submits three items:

- An interface: This is a well known service interface which service users will know about in advance. In abstract terms, the service interface acts as the classification of the service.
- The service: As previously mentioned, the service is an implementation of a particular interface.
- A set of properties: Properties can be used to distinguish between services. For example, two services which implement a LightInterface can be distinguish by the value of a Location property.

On submitting a service for registry, the OSGi registry assigns a service registration, which acts as a unique reference for the service. This provides the providing bundle with a link to the service object within the registry. Service properties are largely left to the developer to define, as there is a lack of properties defined by the OSGi specification.

A bundle is able to register multiple services with the OSGi registry, and is responsible for notifying the registry when the service is no longer available.

3.9.5 Discovering Services

Within the bundle activator, a bundle can discover desired services within the framework by using the OSGi registry. To discover a service, the bundle submits an interface and property query to the framework. The interface is used to match implementations within the registry, with the property query used to further match suitable services to the query. In simple terms, the bundle asks the registry to 'find a service which implements this *interface* and has these *properties*'.

Property queries are expressed using a Lightweight Directory Access Protocol (LDAP) syntax, which allows a more expressive form of querying, over simple property matching.

On discovering suitable matches, the registry returns a list of service references which correspond to services within the registry. The querying bundle can then select a reference from the list, and submit this reference back to the registry. The registry then provides the service to the querying bundle.

Bundles can place requests within the OSGi framework to be notified in the event of a particular service joining the framework. In this manner, bundles which offer multiple services can join the framework in stages. For example, a LightControl service provides a graphical user interface for controlling the lights within the home. As a new LightService joins the framework, the LightControl service can add a new light object to the interface.

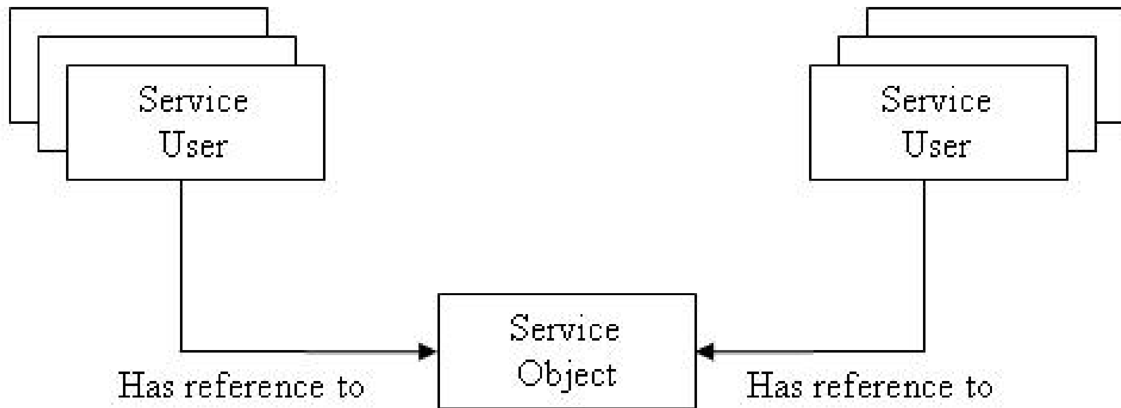


Figure 3.15: Relation Between Services and Users

3.9.6 Using Services

As Java is an object orientated language, services are modeled as objects. When services are retrieved from the registry, the client bundle receives a direct reference to the service object. The service object may be shared between many bundles, as all users share the same implementation (see Figure 3.15). The client bundle interacts with the service by applying the service interface to the service object. In simple terms, this means the service client uses functions defined by the service interface.

As a service unregisters from the framework, service users are notified by a service event. This alerts the service user that the service reference is no longer valid, and should be discarded. In this way, services can leave the framework gracefully, ensuring references are stable and sound.

3.9.7 Scope of OSGi

OSGi is executed within a single Java Virtual Machine (JVM), along with all registered bundles. This places a limitation on an OSGi framework, as all elements of a OSGi network can be susceptible to faults affecting the JVM, such as power failure and class loader errors.

Sharing a single JVM also limits all bundles to share the same implementations of libraries. For example, suppose an existing bundle both offers and uses a LightInterface Java class. If a new bundle joins the framework with a new version of the LightInterface class, it will be unable to start as only one definition of the LightInterface class can exist within the framework.

An OSGi-supported network can also be constrained by OSGi's inability to federate requests between all existing frameworks within the domain. Suppose two OSGi frameworks exist, one which contains entertainment services and one which contains home care services. Service A is contained within the first domain, and is designed to play background music when the user enters the living room. This service wishes to discover if any movement detection services are available. Movement detection services are located within the second OSGi framework, and are used by a number of home care services. Service A would be unable to discover these movement detection services, as it can only discover within it's own framework. The OSGi frameworks are unable to share discovery requests between each other. This OSGi issue has attracted an amount of research and development to resolve. OSGI-R (r-osgi.sourceforge.net/) is one solution to this issue, but has yet to be integrated into the OSGi specification.

3.10 Jini

Jini is another Java-based architecture designed to support distributed computing [93]. The architecture is designed for complete management of the network, covering the behaviour of both network management components and network clients. Network management components may also be distributed within the network. Jini was originally developed by Sun and has been transferred to Apache for future development and refinement[34]. Due to the well defined specifications, Jini has also found itself being applied within home network research projects [49]

3.10.1 Jini Clients

Clients within a Jini network can be said to be mobile. The code of which they comprise can be moved throughout the network. This is necessary, as each Jini client may be executing within its own JVM.

A Jini client can be a service provider, a service user or both. Jini service providers register their services with a service registry. This registry can be queried by service users. Services take the form of *service objects*. These objects model the functions of the service. For example, a Clock service may provide functions for users to get the time, or set alarms.

3.10.2 Registering Services

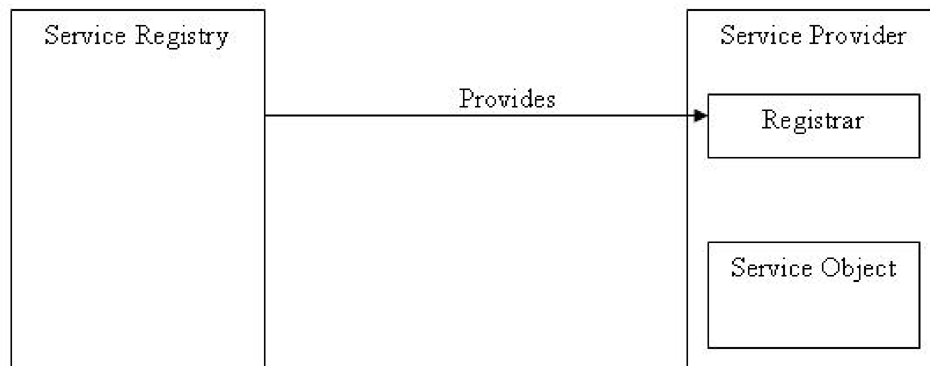


Figure 3.16: Jini Registry Providing a Registrar

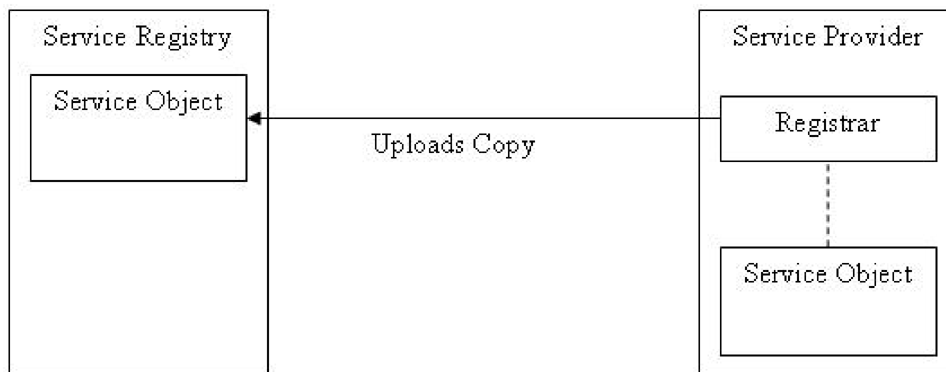


Figure 3.17: Copying The Service Object Into The Registry

The Jini architecture specifies a Service Registry architecture which is designed to handle all aspects of the discovery process. To register, a service provider first discovers the registry by either a unicast to a specific address (if the register address is known) or a multicast to the whole network. The multicast may result in more than one registry responding.

Once the provider has a reference, it obtains a *Registrar* object from the registry (see Figure 3.16). This object acts as a proxy to the registry, providing a local interface for the provider. The provider then passes a copy of the service object to the registrar along with a list of properties about the service. The registrar then places the copied service object within the service registry (shown in Figure 3.17). The provider is then given a unique reference for the service.

Entry Objects

The properties submitted in a service registry take the form of Entry Objects. An Entry is a class which embodies a specific attribute, such as the location affected by the service, the developer of the service, or the man-

ufacturer of hardware which the service controls. Encapsulating attributes within a concrete class allows for a more relevant matching of properties. For example, the service registry would be able to differentiate between *uk.ac.cs.stir.lsd.toaster.Location* and *uk.ac.cs.stir.lsd.gps.Location*. Both properties describe a *Location* attribute, but the context in which it applies is different for each Entry object.

An Entry object simply contains a value, with the property name being identical to the classname of the object. The Jini architecture specifies a number of Entry attributes for describing services. Users can also implement their own Entry objects.

3.10.3 Discovering Services

The initial step of discovering services follows that of registering services. A service user must first find a service registry, if one is not already known. On contacting a registry, the service user receives a *Registrar* object.

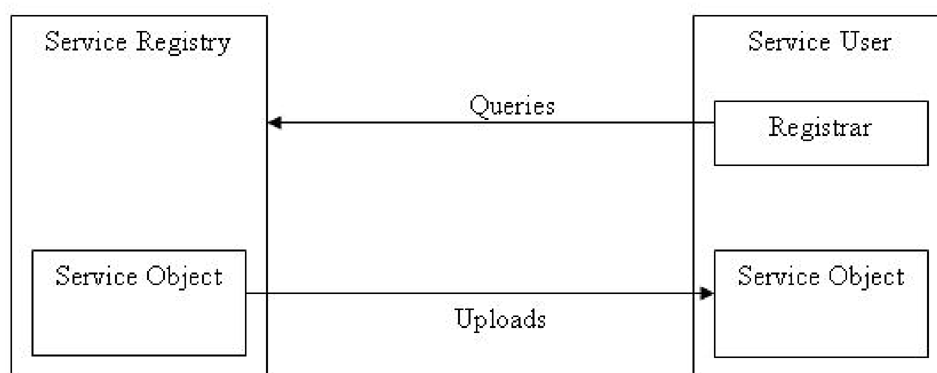


Figure 3.18: Interaction Between the Registry and User

To discover services, a service user must construct a *ServiceTemplate*. This template requires a list of desired service types (defined by a service interface) and a list of desired properties (defined by Entry objects). The template is then submitted to the Registrar, which queries the service registry. If a suitable match is found, a copy of the service object is passed to the service user. The interaction between these components is shown in Figure 3.18. As a Jini network can be distributed, the service user receives a copy of the service to execute within its own JVM.

3.10.4 Service Registry Groups

Jini permits categories of service registries, with each registry deployed for a particular group of services. A number of registries may exist within a single network domain. One service may be registered in more than one registry. In this manner, a service may wish to register itself within a single registry, within a select group, or in all registries within the network.

Similarly, service users may wish to browse a specific group of service registries. Jini services can discover registries by unicast requests (where registry addresses are known) and multicast requests (where addresses are unknown). As part of this registry discovery mechanism, a Jini client can submit parameters denoting which types of registries they wish to discover. For example, departments within a University may maintain their own Jini service registry. A Jini client may wish to discover the registries of the Computing Science department only.

Having multiple registries allow Jini network users to maintain a list of available services relevant to a specific purpose. This may also be as a result of access restriction, where high-priority services are maintained on a controlled registry. Regardless of registry type, service discovery and registration processes are the same.

3.10.5 Proxies within the Jini Network

A service may control a single device. For example, a Jini television may offer a service which allows a user to control the television. Using this service, a user may, for example, change channels and volume levels. If a number of service users wished to control the television, conflicts may occur (for example, if two users wish to

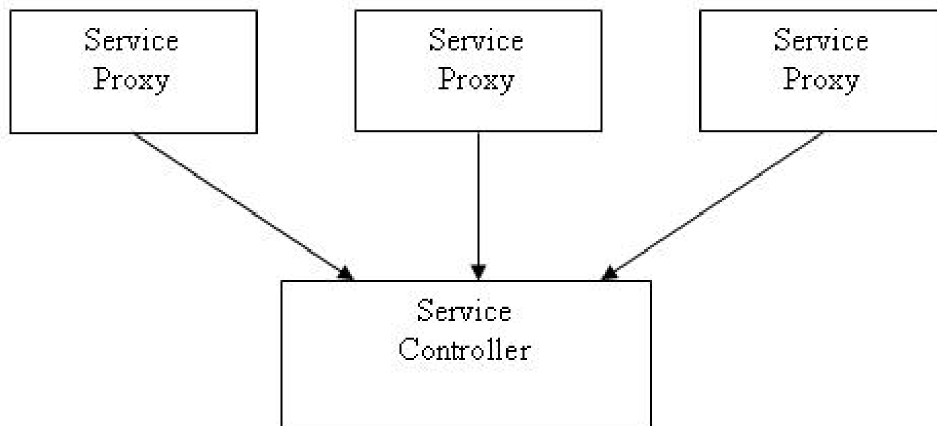


Figure 3.19: Service Proxies and Service Controller

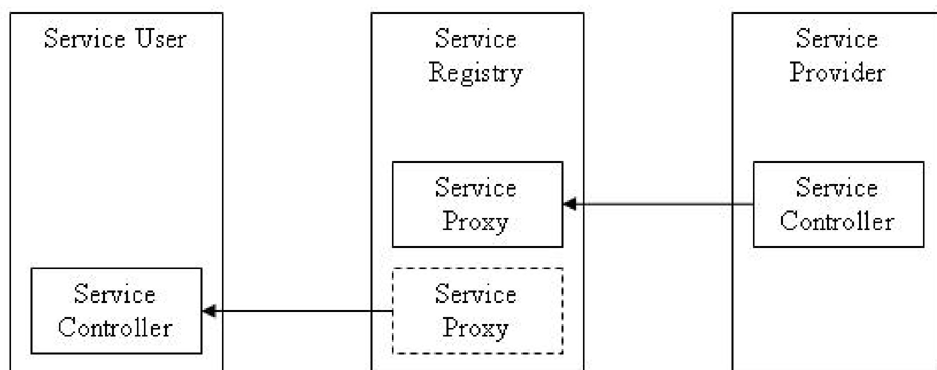


Figure 3.20: Registering and Retrieving the Service Proxy

select two different channels). This scenario would be akin to a single television being controlled by two or more remote controls.

In this situation, rather than providing a direct control service, a proxy service could be provided. This proxy would appear to provide the same functions to the service user, but instead federates functions requests to a service control component (shown in Figure 3.19).

To this end, rather than register the controlling component with the Jini service registry, service proxies can be registered instead. When service users retrieve a copy of the proxy from the service registry, they act as if they hold a copy of the service controller itself. The process is highlighted in Figure 3.20.

In this manner, the service controller can federate requests from users, prioritising actions and resolving conflicts. Using proxies also allows the service controller to configure certain aspects of service usage. For example, if the service user is based upon a wireless device, the controller may provide a proxy with specific communication endpoints. Similarly, if the service user is based upon an Ethernet-based device, a different proxy, with different communication endpoints could be provided.

Section 3.11 presents a real-world implementation of a home network supported by an OSGi middleware framework. This implementation was designed to address the issues identified in Section 3.6. OSGi was chosen because if its popularity and support within research and industry domains. In this experiment, the outcomes are evaluated and presented as a case for this work.

3.11 An OSGi Supported Home Network

A typical OSGi framework has various means to support discovery within the home network. A scenario was created to provide a realistic aim for this experiment: a client wishes to locate a lamp device with a power service.

In this implementation, an OSGi framework is supported with driver bundles, capable of interacting in specific protocol domains. For the purpose of evaluation, driver bundles for X.10, UPnP and Jini have been deployed within the framework to support service and device discovery in this home network domain.

This implementation simplifies part of the discovery process for a client. The driver bundles communicate with their specific domains for the purpose of discovering available components, and subsequently enter a description of discovered components into the OSGi registry. This process removes the requirement from the client to understand how to discover in each specific domain. The client need only know how to interact with the OSGi registry. In this manner, the relationship between the protocols and the methods of discovery is reduced to a simple x:1 relationship. (For x protocols, there is only one method of discovery needed).

This implementation requires that all drivers be configured to interact with the OSGi registry. For this to be achieved, the terminologies and formats of protocol specific descriptions require to be recompiled into a format suitable for the OSGi registry. This requires descriptions to be expressed through an attribute/value pairing. The strengths and weaknesses of this approach have been previously discussed, but it is sufficient to say difficulties arise when sharing an attribute/value environment with multiple parties.

3.11.1 Service-Orientated Discovery

An initial difficulty with an OSGi-registry approach is that it is *service-orientated*. Attributes expressed in the OSGi registry are intended to describe services on the framework, or specifically, attributes of the service implementation. There is little support for describing devices which offer services to the framework, or provide context information.

It could be argued that this is not necessarily an issue: Clients are inherently service-orientated, looking to manipulate real-world objects or data through relevant services. Services are also software-bound where as devices are hardware-bound. These arguments would seemingly suggest that the OSGi approach is sufficient, that service descriptions by themselves are sufficient in a home network. This assumption does not hold. Suppose a client wishes to stream a media file to a component capable of both audio and visual output. From a real world perspective, the client wishes to discover a television, which would offer both services. In a service orientated environment, it is difficult to describe an audio output service and visual output service which can be guaranteed to be offered by the same component.

To simply attempt to solve this by using a defined attribute, such as **offeredByDevice**, proves ineffective. A query in OSGi takes desired attribute values as input. There is scope within the query syntax used by OSGi to match upon wild card values (as OSGi uses LADP for querying over attributes) but there is no function to retrieve meta data about services. For example, it is not possible to discover an audio output service, retrieve its providing device and then query again for a relevant visual output service. This feature would be need to be provided by the service itself.

3.11.2 Registrations

As discussed in Section 4.2, attribute/value approaches to describing components can suffer from unbounded vocabularies. Two services which are identical in what they achieve may be described in logically distinct ways. This is a result of syntactically different, but semantically similar vocabularies. In an interface-driven environment, such as OSGi, this issue is especially prevalent. An OSGi description takes the form of *(interface implemented, service properties)*. The interface provides the classification of the service type. For example, an audio output service offers an implementation of the audio output interface. This approach is suitable in an environment where services can be expressed in an interface/implementation format. A service is bound onto a well known service interface, and a client knows in advance what functions will be offered by the service.

This approach cannot be carried easily into an environment where non-interface based protocols exist. For example, suppose a well known interface existed for Lamp devices: LampInterface. When an X.10 Lamp joins the network, existing X.10 drivers register the device as an implementation of the X.10Device interface. At this point, the X.10Device implementation is syntactically different from a LampInterface implementation, and is therefore logically distinct.

A complex solution to this issue could involve pseudo-implementations, where protocol drivers generate well-known interface implementations, customized for a particular protocol. In this example, the X.10 driver could register an implementation of the LampInterface, customized to control the X.10 Lamp. While this solution appears attractive and straight-forward, it does not scale well. For every possible service interface, a driver must

```
X10Device device = new X10DeviceImpl("A1") ;
Properties props = new Properties() ;
props.put("device-type","Lamp") ;
props.put("location", "LivingRoom") ;

context.register(X10Device.class.getName(), device, props) ;
```

Figure 3.21: An X.10 Registration in OSGi

provide an implementation. As service interfaces are inherently unbounded in number, possible implementations are equally unbounded. Developing a protocol driver requires a choice be made.

If developers choose to ensure their driver is capable of registering a particular service under a well-known interface, then the driver is only suitable for a small selection of protocol specific components. For example, an X.10 Lamp Driver able to register X.10 Lamps under the LampInterface would only be suitable for X.10 Lamp devices. All other X.10 devices would remain unsupported and unusable, unless further X.10 drivers were available.

The second choice is to provide a protocol driver capable of supporting all protocol specific components, but at a more generic level. In the X.10 driver example, this would be akin to simply registering all X.10 devices as implementations of the X.10Device interface. In this manner, all components would be supported, and discoverable - but at the cost of distinction from well-known interfaces. It is the experience of this work that existing protocol drivers are of this second variety. Protocol specific services are registered as implementations of a generic interface, with service attributes providing the necessary discovery information, such as service category. In this manner, compliance with well-known interfaces is sacrificed to ensure all relevant protocol components are registered. This is not an issue limited to OSGi, but to all typical interface-driven registration processes.

3.11.3 Environment Evaluation

This experiment involved using three driver bundles to provide descriptions of three Lamp devices and corresponding services, with one Lamp device per protocol domain. Rather than evaluate each description provided (which would in truth be an evaluation of the protocols rather than OSGi), the OSGi discover mechanism has been evaluated in various aspects including continuity between protocol terms and number of interactions required by a client to discover all relevant components.

3.11.4 X.10 Registration

In the OSGi-supported environment evaluated, the X.10 driver registers X.10 devices under an ‘X.10Device’ interface, along with some descriptive properties. A specific attribute is used to denote the device category: ‘device-type’. Code shown in Figure 3.21 is used to register a Lamp device within the Living Room. It should be mentioned that the X.10 and UPnP protocols have no scope for describing the location of a device. For the purpose of evaluation, locations are derived by way of user interaction.

3.11.5 UPnP Registration

The UPnP driver used within this evaluation is built upon an implementation of an UPnP control point. [2] As the control point discovers UPnP devices, the driver compiles a description based upon the UPnP attributes. This description is then entered into the OSGi registry.

An UPnP device is registered as an implementation of a UPnPDevice interface. UPnP devices cannot be registered as implementations of a generic (or Java based) interface, for the reasons discussed in Section 3.11.2. Instead, a UPnP registration mirrors that of an X.10 registration: the category of device must be obtained from the descriptive attributes. As a UPnP description is schema based, an OSGi based description can be generated by the driver, without the need for user intervention (aside from the addition of a location attribute).

[2] The control point is provided by Satoshi Konno - www.cybergarage.org

```
<deviceType>urn:schemas-upnp-org:device:light:1</deviceType>
<friendlyName>CyberGarage Light Device</friendlyName>
<manufacturer>CyberGarage</manufacturer>
<manufacturerURL>http://www.cybergarage.org</manufacturerURL>
<modelDescription>CyberUPnP Light Device</modelDescription>
<modelName>Light</modelName>
<modelNumber>1.0</modelNumber>
<modelURL>http://www.cybergarage.org</modelURL>
<serialNumber>1234567890</serialNumber>
<UDN>uuid:cybergarageLightDevice</UDN>
<UPC>123456789012</UPC>
```

Figure 3.22: UPnP Description of the Light Device

```
UPnPDevice device = new UPnPDeviceImpl(deviceLocation) ;
Properties props = new Properties() ;

props.put("UPNP_DEVICE_TYPE", "urn:schemas-upnp-org:device:light:1") ;
props.put("UPNP_DEVICE_FRIENDLYNAME","CyberGarage Light Device") ;
props.put("UPNP_MANUFACTURER","CyberGarage") ;
props.put("UPNP_MANUFACTURER_URL","http://www.cybergarage.org") ;
props.put("UPNP_MODEL_DESCRIPTION","CyberUPnP Light Device") ;
props.put("UPNP_MODEL_NAME","Light") ;
props.put("UPNP_MODEL_NUMBER","1.0") ;
props.put("UPNP_MODEL_URL","http://www.cybergarage.org") ;
props.put("UPNP_SERIAL_NUMBER","1234567890") ;
props.put("UPNP_UDN","uuid:cybergarageLightDevice") ;
props.put("UPNP_UPC","123456789012") ;
props.put("location", "LivingRoom") ;

context.register(UPnPDevice.class.getName(), device, props) ;
```

Figure 3.23: UPnP Description Converted for OSGi

UPnP Device Description Conversion

The UPnP description of the Cybergarage Light Device used in this evaluation is given in Figure 3.22.

As this description is already in a attribute/value format, it is a relatively simply transition into an OSGi description, shown in Figure 3.23. A location attribute is added by means of user interaction as each UPnP device is discovered.

In a similar manner the UPnP service description (a power service) is translated from the UPnP terminology into an OSGi registration, shown in Figures 3.24 and 3.25

The UPnPService class contains functionality which allows clients to retrieve the identification of the parent device. This is required to provide linkage between the service and the device which is offering it to the service. It should be stated that services require to be registered independently of their parent device. If a parent device only offers one service, a case could be made that a service reference can be made within the device description. For example, the Light Device description could be modified with a new property:

```
props.put("UPNP_SERVICE","urn:schemas-upnp-org:service:power:1") ;
```

While this would be sufficient for single service devices, it is not feasible for multiple services, as only one 'UPNP_SERVICE' entry is permitted per registration. Similarly, it would be unfeasible to mix device and ser-

```
<serviceList>
   <service>
      <serviceType>urn:schemas-upnp-org:service:power:1</serviceType>
      <serviceId>urn:schemas-upnp-org:serviceId:power:1</serviceId>
      <SCPDURL>/service/power/description.xml</SCPDURL>
      <controlURL>/service/power/control</controlURL>
      <eventSubURL>/service/power/eventSub</eventSubURL>
   </service>
</serviceList>
```

Figure 3.24: UPnP Description of the Power Service

```
UPnPService service = new UPnPServiceImpl(deviceLocation) ;
Properties props = new Properties() ;

props.put("UPNP_SERVICE_TYPE","urn:schemas-upnp-org:service:power:1") ;
props.put("UPNP_SERVICE_ID", "urn:schemas-upnp-org:serviceId:power:1") ;
props.put("UPNP_SCPDURL", "/service/power/description.xml") ;
props.put("UPNP_CONTROL_URL", "/service/power/control") ;
props.put("UPNP_SUB_URL", "/service/power/eventSub") ;

context.register(UPNPService.class.getName(), service, props) ;
```

Figure 3.25: UPnP Description of the Power Service Converted for OSGi

vice attributes within the UPnPDevice registration, as this violates the integrity of the description (i.e. a device description should describe only attributes of the device, and similarly for a service description).

3.11.6 Jini Registration

The Jini protocol driver has a more straight forward role in converting between the protocol specific description to an OSGi based description. As previously discussed, Jini uses Entry objects to provide attributes about a service. A Jini registration takes a similar format to that of an OSGi registration: (*service interface name, service implementation, list of Entry object attributes*). The only change required for an OSGi registration is to convert an Entry object into an attribute/value pair. An Entry object contains one or more fields, with each field representing a particular attribute. For this evaluation, the conversion process is done by combining the class name of the Entry object and field as the attribute name, and the value of the Entry as the attribute value.

A Jini service has been developed which implements the LampInterface, and includes a Location Entry object, which is used to describe the location of the physical component of the service. The steps involved in creating a Jini description are given in Figure 3.26

```
LampInterface lamp = new LampInterfaceImpl() ;
\\A location at 12 Tree Walk, on the ground floor, in the Living Room
Location loc = new Location("12 Tree Walk", "0", "LivingRoom");
Entry[] entries = new Entry[] {loc} ;
ServiceItem item = new ServiceItem(null, lamp, entries) ;
```

Figure 3.26: Jini Description of a Lamp Controller

The ServiceItem object contains both the service instance and its descriptive properties. As this is passed to the Jini driver (which is built upon a Jini Registrar) an OSGi description is derived, shown in Figure 3.27.

```
LampInterface lampImpl = lamp ;

Properties props = new Properties() ;
props.put("net.jini.lookup.entry.Location.building","12 Tree Walk") ;
props.put("net.jini.lookup.entry.Location.floor","0") ;
props.put("net.jini.lookup.entry.Location.room","LivingRoom") ;

context.register(LampInterface.class.getName(), lamp, props) ;
```

Figure 3.27: Jini Description of the Lamp Controller converted for OSGi

It is important to provide an unique URI reference for Jini attributes, rather than simply use the field name of the Entry object. In this manner, the driver can ensure that two Entry objects with similar field names remain distinct. Suppose two Entry objects existed with a field named **location**, where the intent from one provider is to describe the physical location affected by a service in the home, and the other provider describes the download location of the service by field. It would not be enough to represent both attributes with the same string value, as a client would have no way to differentiate between the two intentions.

The testbed for this evaluation is now complete. There are three instances of Lamps or Lamp-controlling services within the OSGi registry. This allows OSGi clients to have a centralised point of discovery, highlighting the x:1 relationship between existing protocols and methods of discovery required. The evaluation will concentrate on the process involved by a client in discovering each instance of the Lamp components.

3.11.7 Evaluation of the OSGi Discovery Process

A LightSwitch service has been developed to operate the various Lamp services and devices in the home. This service is able to use all Lamp services within the framework. The LightSwitch service first requires to discover the available Lamp components. The LightSwitch service has knowledge of how each component is described within the OSGi registry. In order to discover the components, the LightSwitch must query the OSGi registry.

Querying the Registry

In order to discover each type of Lamp component, the client must query the registry three times, with three distinct sets of terminology. Firstly, to discover the X.10 Lamp, the client must submit a query in the format shown in Figure 3.28.

```
ServiceReferences[] x10References =
context.getServiceReferences(X10Device.class.getName(),"device-type=Lamp") ;
X10Device lamp = (X10Device)context.getService(x10References[0]) ;
```

Figure 3.28: Discovering the X.10 Lamp Service

The variable **lamp** now holds a reference to the X10 Lamp controller. This query required the use of protocol specific terminology: '*device-type=Lamp*'. The property 'device-type' is specific to the X.10 domain.

Discovering the correct UPnP service is slightly more complex. Within the UPnP domain, the desired service is a 'power' service which is offered by a Lamp device. It is not enough to simply discover any power service, it is crucial that the service is offered by a Lamp device. Figure 3.29 shows the steps taken to discover the correct UPnP service.

Querying over the UPnP domain requires a completely separate terminology from the X.10 domain. Terms such as 'UPNP_DEVICE_TYPE' and 'urn:schemas-upnp-org:device:light:1' are specific to the UPnP domain within OSGi. A service user who wishes to be able to discover UPnP resources must have knowledge of the UPnP terminology.

Jini provides a unique case of discovery. As the Jini service implements the LampInterface interface, the service type is implicit within the interface class name. In other words, the client knows that if the Jini service

```
ServiceReferences[] upnpReferences =
context.getServiceReferences(UPnPService.class.getName(),
"UPNP_SERVICE_TYPE=urn:schemas-upnp-org:service:power:1") ;
UPnPService service = (UPnPService)context.getService(upnpReferences[0]) ;

//check for correct parent
ServiceReferences[] lampReferences =
    context.getServiceReferences(UPnPDevice.class.getName(),
     "UPNP_DEVICE_TYPE=urn:schemas-upnp-org:device:light:1,
         UPNP_DEVICE_FRIENDLYNAME="+service.getDevice().getFriendlyName()) ;

//if references are not empty, we can assume a correct match
```

Figure 3.29: Discovering the UPnP Power Service

implements the LampInterface interface, it controls a Lamp device within the network. Figure 3.30 shows the steps taken to retrieve the Jini service. No terminology is required in this instance, as the service type is not considered an attribute of the service.

```
ServiceReference jiniReference =
   context.getServiceReference(LampInterface.class.getName()) ;
LampInterface lamp = (LampInterface)context.getService(jiniReference) ;
```

Figure 3.30: Discovering the Jini Service

Evaluating the Query Process

Discovering three separate protocols required two different terminologies to be used, and one instance where no terminology was required. At face value, the OSGi experiment has shown that, in some cases different sets of terminologies are required, but in others no terminology is required. It is difficult to draw any clear conclusions from this simple experiment.

Clearer conclusions can be drawn by evaluating a more complex experiment. To this end, more realistic queries have been constructed and submitted to the OSGi registry. In this case, rather than simply search for available Lamp devices or services, the client provides a specific criteria, namely those in a specific location: The Living Room. In natural terms, the client is searching for a Lamp service which affects the Living Room.

Discovering relevant X.10 and UPnP services simply requires an extension of the attributes used within the query process. The X.10 query adds an additional property and value:

- ‘location=LivingRoom’

Similarly, the UPnP query requires an addition property. In this case, it is added to the second part of the query, ensuring that the parent device of the service is located in the Living Room:

- ’location=LivingRoom’

The Jini query requires to be restructured, as it now includes a criteria for determining suitable services. The Jini query takes a form similar to the queries of the X.10 and UPnP queries, shown in Figurefig:OSGi-Jini-Mod.

This experiment forces the Jini query to include protocol specific terminology as part of the query. For a client to discover a Lamp service in the Living room now requires distinct terminologies for each protocol domain. The X.10 and UPnP queries include a common attribute, **location**, which is abstract from the protocol specific terminologies. This common attribute does not succeed in preventing the overall query from being protocol specific.

In the experiments carried out in this work within an OSGi supported home network, the relationship between the number of protocols in the domain, and the number of vocabularies required to be known by a client remains

```
ServiceReference[] jiniReferences =
   context.getServiceReferences(LampInterface.class.getName(),
      "net.jini.lookup.entry.Location.room=LivingRoom") ;
LampInterface lamp = (LampInterface)context.getService(jiniReferences[0]) ;
```

Figure 3.31: The Modified Jini Query

a x:x relationship. If one aspect of a query requires protocol specific terminology, then the whole query becomes specific to that protocol domain. It is not possible to reuse queries constructed for one protocol in discovering components within a second protocol domain. In short, there is no scope for one query being applicable across protocols.

This observation is not based on a whole OSGi registry query per say, but only on the attribute criteria submitted as part of the query. An OSGi query requires a desired interface to be submitted as part of the query. It is assumed that this interface may, in many cases, always be protocol specific, for the reasons described in Section 3.11.2. The interface can be thought of describing *how* clients interact with the service, rather than attributes of the service. As has been previously mentioned, this work is concerned with describing *what* services are, rather than how they can be used.

3.11.8 OSGi Evaluation Conclusion

OSGi provides a great deal of support for cross-protocol discovery of network components. As has been shown, utilising the registry of an OSGi framework reduces the (protocols : discovery method) relationship to x:1. This provides a solid base for component discovery. The issues surrounding protocol specific terminology still remain. A x:x relationship between protocols and vocabularies is not scalable within a home network environment. Existing services which do not update their range of protocol specific vocabularies are left behind, potentially becoming obsolete as new protocols emerge.

In summary, this experiment provided the following outcomes:

- The relationship between the number of protocols within the network and the methods of discovery needed is reduced to x:1 relationship. The OSGi registry provides a central point of discovery, as protocol drivers are responsible for ensuring protocol specific components are discovered. This is a perfect relationship, even in a worst-case scenario.

- The relationship between the service type the client wishes to find and the number of possible messages or vocabularies needed to discover the service remains at 1:x. The use of differing vocabularies and attributes requires clients to have knowledge of how an attribute is represented in each domain. This issue remains unchanged from the naive scenario presented in Section 3.6.

It is clear that, to provide an extensive and scalable middleware framework, a new approach toward component discovery is required.

Chapter 4

Service Discovery Approaches in the Home Network

Despite a number of approaches to supporting a Home Network, there are a few common approaches to service discovery. These approaches can be loosely separated into a three categories:

- System Architecture
- Component Description
- Service Usage

4.1 System Architecture

The architecture of each service discovery approach can bring differing solutions and issues to the network environment. There are two main architectures found within the home network: decentralised and centralised. Rather than apply to all aspects of network life, these categories represent the management aspects of networks, such as service discovery, error recovery, and message infrastructure. Service usage, in terms of architecture, is considered to be service to service.

4.1.1 Decentralised Architecture

UPnP (discussed in section 3.2), and to a lesser extend HAVi (discussed in section 3.3), operate a decentralised approach to service discovery. Within a decentralised architecture, there is no main component, or components, tasked with maintaining a list of services. A discovery request for a specific service is broadcast to the entire home network. Components which offer the desired service respond directly to the querying component.

With a decentralised approach, the network is less prone to component failure. For example, suppose an UPnP Lamp joins the network. Various control points utilise the functions of the component, with others subscribing to Lamp events. If the Lamp component fails, client services are no longer able to utilise the Lamp, and subscribers no longer receive updates. Other components within the network remain unaffected. Service and device discovery can continue to take place. Users of the failed Lamp can discover alternatives and continue to function normally. Figure 4.1 depicts this architecture in use.

HAVi networks may suffer more from node failure due to FAV and IAV devices acting on behalf of more basic devices within the network. Failures of FAV or IAV devices can result in services of a number of devices being withheld from the network.

In a decentralised network, all components are responsible for keeping information about themselves consistent. This usually means components requiring to continually broadcast messages concerning their existence or current state. This can result in a large number of messages being transmitted to ensure the network is maintained in a consistent state.

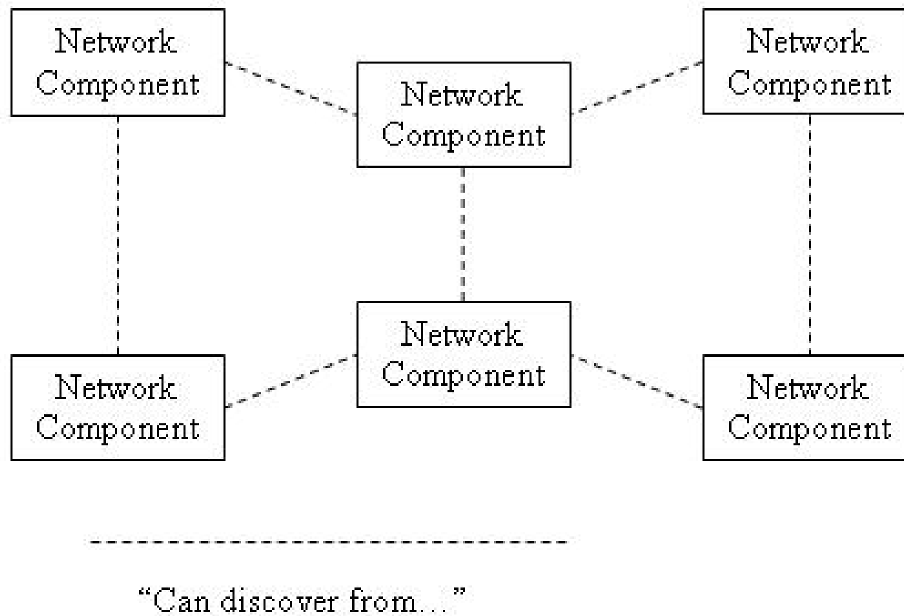


Figure 4.1: Discovery in the Decentralised Network

4.1.2 Centralised Architecture

OSGi typifies a centralised approach to supporting the Home Network, with Jini holding some similar characteristics. In a centralised network, there is a main point of contact for all components joining the network. The central component can be responsible for a number of operations, such as a service registry or network security (shown in Figure 4.2). A centralised service registry provides a single point of reference for service clients. In this manner service users will know that all services will be registered in a single place, and can be confident that if the service is in the network, it will be in the registry also.

A centralised approach allows network management components to apply priorities and authorisation levels to network actions. For example, Component A may wish to discover a Home Security service. The centralised security component can limit the types of services which Component A can discover. In this example, Component A does not have a high enough authorisation to discover Home Security services, and as a result is returned no results. Having a single entry point to major components, such as service registries, allows a more reliable and robust security implementation.

Centralised Architectures also ensure that every possibly discoverable network component is discoverable. In other words, when a client searches a centralised registry, they can be confident that all available components are represented within the registry. Centralised architectures also allow for registries to ensure descriptions are consistent.

In terms of network robustness and reliability, centralised networks are more prone to network failure. Failure may be caused by a number of issues, such as component overuse power loss. If the components responsible for major network actions, such as service registry, are unavailable, the network is unable to function.

4.2 Component Description

Approaches to service descriptions vary amongst protocols. Some protocols use no descriptions (such as X.10), some contain an implementation of and attribute/value approach (such as Jini and OSGi), and others have a defined description schema (such as UPnP). The extent of the description approach has some degree of influence over the ease of integration into the home network. For example, it is easier for middleware to extract service attributes from a description, if the component adheres to a protocol specific schema.

This section will describe the approaches to integrating protocols with middleware, with respect to their description approach. This section will also evaluate each approach on 3 important areas within the Home Network

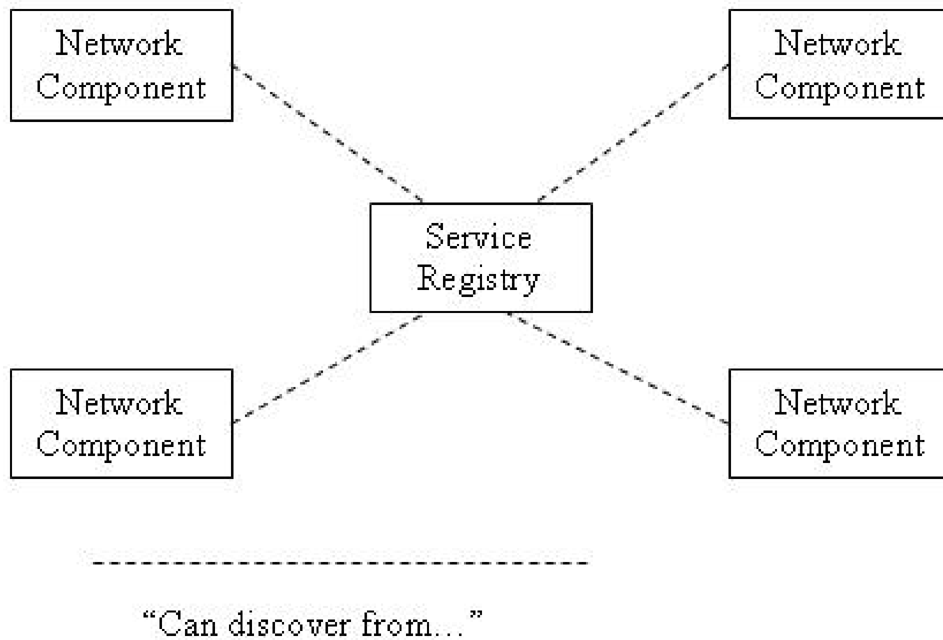


Figure 4.2: Discovery in the Centralised Network

environment.

- Configuration - This area refers to the ease and level of involvement required in setting up a network with a particular description approach
- Interaction - This area refers to how automated the service discovery process is with respect to the particular approach
- Integration - This area refers to the ease of integrating protocols of a particular description approach into a home setting

4.2.1 No Description

Protocols which have no descriptive specification provide a unique challenge in an multi-protocol network. There is no possibility for a middleware framework to perform any kind of classification upon the device or service. Protocols which contain no descriptions are largely hardware based, such as X.10.

Configuration

With the emphasis on the simplification of networking, descriptive properties have been traded for ease of installation and configuration. Such network devices are, in terms of description approach, the equivalent of current domestic appliances, such as televisions and washing machines. In this manner, little configuration is required to integrate these components into the network.

Interaction

With no descriptive classifications of devices or services, interaction in this approach requires being direct. For example, it is not possible to carry out the action 'Switch on the Lamp in the Living Room', as there is no concept of Lamp or Living Room (or even Switch On). Instead interaction is limited to 'Switch on Device X' or 'Pass these arguments to Service Y': There is no computational concept of what component the client is interacting with. Interaction therefore can either be hard coded, or human user initiated.

Integration

Understanding the *type* of device requires the intervention of the human user. Any kind of description has to be applied externally to the device or service. The easiest approach to this is a description wizard application, which allows the user to describe the device or service to the framework. With a rudimentary description present, these network components can now be integrated (from a descriptive level) into the home network. With a lack of protocol specific vocabulary, the user requires applying descriptive properties used by the middleware to the service and device descriptions.

4.2.2 Attribute/Value Pairs

When using a service registry, services are typically stored with a set of descriptive attributes, (see sections 3.10.2, 3.9.4 and 3.3.4). These attributes are essentially a set of attributes with corresponding values. The attributes typically accompany an interface, or classification of the service or device they are describing. In essence, the whole description takes the form:

```
"I am a service/device of type X and I hold these descriptive properties"
```

The attributes are syntactic representations of a specific semantic attribute. For example, a *Location* attribute may be defined to represent the location of a particular device in the home. The word *Location* is the tag, or anchor onto this semantic concept. Knowing what the attribute represents allows middleware applications to understand the values it extracts from the service properties. In a similar manner, the interface or classification of the service is a textual anchor to the service type.

Configuration

Having an open approach to providing descriptive attributes allows this approach to be easily configured. As there are no restrictions or requirements as to what attributes are present, developers are free to provide any number of attributes which they believe to be suitable. The only requirement in this approach concerns the service or device interface. The component must adhere to the functions offered by the component interface.

Interaction

This approach, while lightweight in deployment, suffers from one main issue - the requirement of well defined properties. Suppose a device provides a Lamp service whose *Location* attribute has the value *LivingRoom*. A LightControl service wishes to know about all Lamps which have their location within the home. Suppose the LightControl service understood the values representing the rooms in the home, such as *LivingRoom, BedRoom, and Kitchen*. The LightControl service, however, is unaware of the *Location* attribute, instead using a *Room* attribute to tag the concept. The LightControl service would not be able to discover the Lamp service as the criteria over the Lamp query would not match. The Lamp would be offering (*Location, LivingRoom*) as a descriptive property, while the LightControl would require services with property (*Room, LivingRoom*).

In order for the Attribute/Value approach to succeed in a network environment there needs to be a well defined set of properties which service providers and users can adhere to. For example, suppose a certain middleware framework is chosen to support a Home Network environment. The developer of the middleware, or the network technician, could define a set of properties which will be used in service discovery. If all network components comply with this set, service and device discovery will be well supported. If some components choose to use their own set, they risk being unused by service users. As illustrated by the above example, two or more properties may be designed to tag the same semantic concept. In many implementations of the attribute/value approach, the matching of service user criteria against service provider properties is purely syntactical.

Regardless of the issue highlighted, discovery in an attribute/value environment can be fairly well automated, as components can use any available services or devices which firstly adhere to the desired interface, and secondly hold the desired attribute values.

Integration

The issue described above continues to exist within an integrated environment. If two protocols share this description approach, more care is required to ensure attributes are well understood and known. In a multi protocol

environment, where multiple parties may be involved, the issues of multiple tags for the same concept becomes more difficult to resolve. Not only does there require to be an agreement between developers within a single protocol as to what terminology to use, there also requires a cross protocol agreement to adhere to the same terminology.

If this issue is resolved, the integration of the attribute/value approach is straightforward. While different protocols may use different service discovery techniques, the content of the querying messages can traverse protocols. In integrating into an environment with differing approaches to descriptions, the requirement of an taxonomy for adherence becomes more vital. Having a defined set of attributes allows external protocols to access the descriptive content, and therefore fully participate in the discovery process.

4.2.3 Description Schemas

Automation in the home can require a great deal of configuration. Some home automation requires the user to configure the whole network, for example in a X.10 network. Some scenarios can involve a mixture of human and computer configuration, where the user installs the device (and perhaps device drivers) but the device then configures itself within the network. A third scenario exists where the entire configuration process is handled by the network device itself, such as in UPnP networks.

In a *zero configuration* network, every aspect of the configuration is required to be programmed into the network device. In such an environment a simple attribute/value approach to descriptions is not sufficient, as this approach suffers from a lack of standards. In a tightly controlled environment, a description schema can be used to support the self configuration. Such a schema specifies an exact set of attributes which will be present in the description of a specific device or service. For example, a Lamp description schema may specify that a description of the device *must* have a *Location* attribute present. For each component, a schema or template exists. In order to be valid, a description must adhere to the defined schemas.

Configuration

The main drawback to description schemas lies in the amount of time and resources required to develop such schemas. For each new device, a new schema must be created, and similarly for a new service. As networked components may be developed by a range of vendors, it may be more efficient for a third party to develop schemas for the protocol (this is the approach that the UPnP forum have taken). In this manner vendors can adhere to a common schema, and interoperability in a multi-vendor environment is possible. This approach however requires the third-party to create schemas which are relevant to the widest range of vendors, in order to encourage uptake and adherence to the schemas.

Interaction

Description schemas provide a robust discovery environment. Service users can be confident that descriptive attributes will be present in the provider descriptions. Similarly, users can be confident that a particular device will offer a set collection of services. For example, a description schema for a printer may specify exactly what services will be offered (such as printDocument, getTonerLevel). In this manner, the description acts as a soft interface for devices and services.

Discovery within a schema-based network can be said to be fully automated, as no interaction should be required from the human user. In previous approaches, users may be required to explicitly create service or device links (between clients and providers) or resolve syntactical issues. Utilising schemas ensures no unexpected situations arise at the discovery stage.

Integration

Integrating a schema-based protocol into a network benefits from similar advantages described in the previous interaction section. A middleware component can easily extract protocol specific attributes from a schema-driven description. Knowing in advance what attributes will be present allows the protocol to be adapted into either a similar schema based framework or an attribute/value framework. Search agents designed to work in a multi-approach environment can be pre-programed with knowledge about schema attributes from multiple protocols, including identification of where attributes from different protocols represent (or tag) the same semantic concept.

	No Description	**Attribute/Value**	**Schema**
Configuration	Simple	Medium	Heavyweight
Interaction	Unautomated	Fairly automated	Fully Automated
Integration	User Intervention	Medium	Easy

Table 4.1: Table of Comparisons

4.2.4 Comparison of approaches

Table 4.1 summerises the advantages and disadvantages associated with each approach. Description schemas offer clear advantages when provisioning for a potentially complex network environment. Having a well structured approach to configuration, interaction and integration provides a reliable environment for interoperability. As interaction procedures are well defined, possible outcomes can be known in advance. Providing a high level of automation also reduces the number of direct interventions by a human user. The heavyweight approach to the description (configuration) of network components using description schemas can restrict the usefulness of this approach, in terms of development time, and ensuring the schemas are *well known* enough by other protocol users.

This work proposes the use of ontology languages to address the issue of heavy-weight description approaches. Ontology languages are designed to provide a common reference vocabulary [43], while being able to unify existing vocabularies [45]. Chapter 5 provides a detailed discussion of ontology languages, and how they can be used to alleviate issues of multiple terminologies being used within a common domain. In this manner, this work, described in Chapter 8 allows home networks to retain the advantages of schema-based protocols, while supporting the configuration aspect of protocols of all three approaches.

Chapter 5

Ontology Languages

An ontology is a data structure which is concerned with describing various aspects of a domain [47]. The data within the ontology describes concepts within the domain, and the relationships between them. For example, an ontology about a library may describe a list of books and for each book, a number of attributes relevant to them, such as author and publisher. A number of ontology languages exist which provide a range of capabilities for describing information. This chapter will provide an overview of the language used in this work, including the foundation languages used. As a prologue to the ontology discussion, this chapter will first discuss the use of meta-data within the ontology domain, and its relevance to the discovery process.

5.1 Meta-Data

Ontologies assist search agents in locating relevant information, using typically meta-data to enhance the discovery process. Meta-data can be defined as data about data, or attributes of the data. For example, a search agent could search the library database for all books written by a particular author, and published in a particular year. In this example, the name of the author and the year of publication would be considered meta-data.

Without meta-data, searching in any environment would typically be a shallow process. For example, within common library online catalogues, it is possible to search for a single book by title. Suppose we wish to find out if the library owns a copy of 'PC Building for Beginners'. Searching for this book simply involves entering the title into the relevant field. If the book is not in the library, the search fails. This simple discovery process, where the search is only dependent on a specific title, is one dimensional. The search space, which is a collection of possible matches to the search, is a simple list of titles which may match the desired title.

Suppose that the reason we searched for a particular title is because it was the only title we knew dealt with PC building. Rather than limit the search to a specific title, a more robust approach would be to search the library on the subject of the book. In this example, the search parameter is concerned with the subject of the book, 'PC Building'. Searching over the subject of all books within the library provides a larger search space for the search, and therefore may return a greater list of relevant books.

Meta-data, such as a book subject, provides a richer search environment, where the search process can be relevant and robust. If attributes of the library books, such as book subject, are kept within the catalogue, the search mechanism can provide a more supportive search.

This environment is in contrast with existing search mechanisms for the internet, which is largely devoid of well-structured meta-data. Initial approaches to online searching made use of keywords embedded in webpages, in order to identify their content. A search engine would utilise these keywords in order to determine the relevance of the page to the search. It could be argued that keywords are a form of meta data, as they attempt to convey some meaning about the web document. With naive keywords, there was no way to determine what attribute is being represented. For the most part, a keyword identified content, which limited their usefulness to searches over content. Searches over any other attributes, such as author or product supplier, required a more descriptive approach.

The keyword approach also suffered from lack of control and verification over the keywords used in a webpage. For example, if one web page was merely concerned with receiving the most number of visits, it could embed a number of popular keywords in order to be returned as a match for a variety of searches. In a less malicious example, an online car sales site could notify that it deals with vehicle sales. Using keywords, a search for 'car

sales' would not find the online car sales site as it does not contain 'car' as a keyword.

Due to these issues, simple keyword searches have been largely abandoned in favour of more sophisticated ranking, indexing and discovery algorithms. These approaches are still dependent on meta-data being available about web pages and resources. For this reason, a number of description frameworks for structured meta-data have emerged.

5.2 The Resource Description Framework

The Resource Description Framework (RDF) [43] was motivated by the need to structure data on the web. The initial purpose of RDF was the definition of webpages through their meta-data (such as author, location, last update) [43]. Defining the meta-data allows human readable attributes of a webpage to become machine readable. With a defined framework, searching for relevant webpages could become more like the searching within a library system. The framework contains three main elements: *resources* and *properties* are used to describe data, and are encapsulated within a *statement* to represent facts.

Resource

A resource can be anything which has a Uniform Resource Identifier (URI). A URI uniquely identifies the resource within its domain. The most simple example of a resource is a webpage, whose URI would be the Uniform Resource Locator (URL) of the webpage. This URL uniquely identifies the webpage within the domain of the Internet, and therefore can act as the URI.

Property

A property describes an attribute of a resource. For example, **Author** could be a property denoting the author of a webpage. A property is also a resource, as it needs a unique reference within its domain. It then follows that a property may also have its own properties.

Statement

An RDF statement consists of a Resource, a Property and a value. A statement is a triplet of information, expressed in the logical format: **(subject, property, object)**. The subject is always a Resource, and a property is always a member of the Property class. The object within a statement may be a literal value or it may be another resource. An example of a literal value statement would be **(www.cs.stir.ac.uk/˜lsd, Author, 'Liam Docherty')**. An example of a statement with a resource as the object would be **(www.cs.stir.ac.uk/˜lsd, Author, www.lsd.org)**. Here www.lsd.org could refer to a webpage about the author.

5.2.1 Expanding the Boundaries

The boundaries of RDF have expanded into describing more general resources which are described on the web, but are not actual web pages. For example, an online music store offers CDs for sale. Each CD has some meta-data associated, such as price and artist. Using RDF, this meta-data could also be captured and expressed using resources, properties and statements. For example, the price of a Queen album could be expressed in the form **(Queen's Greatest Hits, Price, '£9.99')**.

At this level, RDF can provide meta-data support for web searches, resembling our library search example earlier. For example, we could customise a search agent to find a website which sells 'Queen's Greatest Hits' for £10 or less.

5.2.2 Using Resources

As mentioned, the object of an RDF statement can be a resource. Using the example of the website, suppose the following statements existed:

- `(www.cs.stir.ac.uk/˜lsd, Title, "Liam's Departmental Page")`
- `(www.cs.stir.ac.uk/˜lsd, Author, www.lsd.org)`

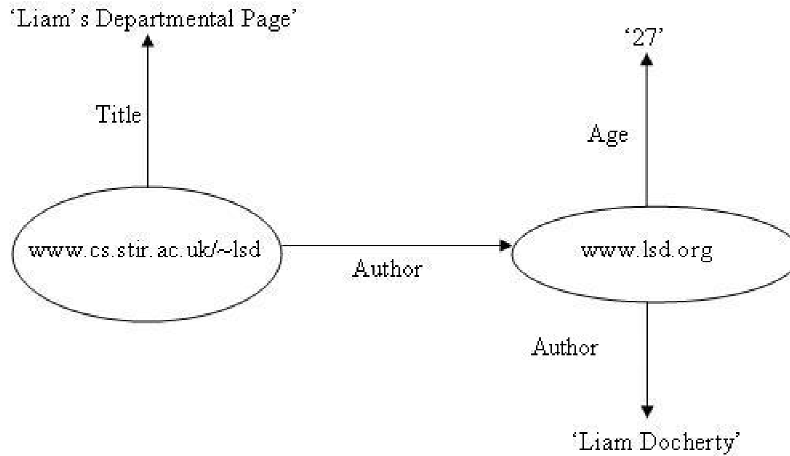


Figure 5.1: RDF Graph about www.stir.ac.uk/ lsd

```
<rdf:Description about ="www.lsd.org">
   <Age>27</Age>
   <Author>"Liam Docherty"</Author>
</rdf:Description>
```

Figure 5.2: Expressing RDF Information in XML

- `(www.lsd.org, Author, "Liam Docherty")`
- `(www.lsd.org, Age, "27")`

The second statement contains a resource as its object, and therefore must then have a URI. As this object is a webpage, the URI is also a URL and can be located within the Internet. A search agent could be tasked with finding out the age of the author of **www.cs.stir.ac.uk/˜lsd**, and by following the URL it would be able to extract the age from **www.lsd.org**.

The use of resources on the internet is a useful attribute within RDF, as it allows information about a particular resource to be defined elsewhere. As demonstrated, the age of the webpage author is not defined on every webpage, but rather on his own personal webpage. This mirrors how information is held on the Internet, where information about particular resources are defined independently from where the resource is used. For example, Amazon may sell CDs and books, but information about these items are defined by their publishers, rather than the Amazon website. In this manner, information needs only to be defined once for each resource, and can then be displayed in other locations by extracting this information from the original source. A conceptual diagram of these statements can be found in Figure 5.1.

Using RDF properties

Figure 5.1 highlights an important issue found within RDF. There are two instances of the **Author** property, which involve two different resources as the property object. In one instance, the resource is an object, and in the other, a literal reference. In each case, the property statement is valid, as there are no restrictions placed on the property as to what it refers to. This issue is discussed further in Section 5.2.5.

5.2.3 Transporting and Exchanging RDF

Representing RDF data in statements allows the meta-data to be quick to access and easy to index. In contrast, when storing RDF data in a persistent manner or exchanging data, statements are an inefficient approach, due to the potential number of statements. For both storage and exchange, RDF triples can be expressed in an XML format. An example is given in Figure 5.2.

Using XML allows RDF data to be encapsulated in class descriptions. Using the example in Figure 5.2, all information about www.lsd.org is encapsulated within a single description element. XML descriptions about classes are limited to a single instance. That is to say, once a class description is encountered, a user of RDF can be sure that no other information will be added to the description later in the document. This point is important, as it streamlines the parsing process. As XML is a standard data exchange format, users of RDF may utilise a wide range of tools available for parsing the document. If a user applies a SAX parser to an RDF document, they can quickly jump to the description of classes they are interested in, without having to parse the rest of the RDF document. Using XML also allows the use of namespaces, reducing the amount of text within the document.

The lightweight format imposed by the XML serves to ensure that RDF documents are well formed. Having a standardised format to represent RDF data facilitates the exchange of RDF properties between users.

As RDF properties are shared and reused, a standard set of properties will emerge to describe a domain. For example, say Amazon, Play.com, iTunes and Walmart websites all use the same RDF properties to describe their products. Intelligent search agents, which can make use of the RDF meta-data, can be developed which can cover all four websites. If an emerging website also uses these RDF properties to describe their products, the search agent can then include the new website in its search domain. In this manner, the most used properties become the standard for the domain.

5.2.4 RDF/XML

RDF/XML [42] is the syntax designed to facilitate the exchange of RDF between agents and parties. The syntax requires no DTD, as there are no restrictions on order, cardinality and existence of elements within the document. As explained in section 5.2, two of the main elements within RDF are resources and properties. These elements transfer seamlessly into XML.

Resources in RDF/XML

Using the graph shown in Figure 5.1, we can represent the **www.cs.stir.ac.uk/˜lsd** resource as:

```
<rdf:Description about ="www.cs.stir.ac.uk/˜lsd">
```

The <rdf:Description> tag annotates that this element will describe a resource in the RDF data. This element can contain an attribute of either 'rdf:about' or 'rdf:ID', followed by the name of the resource it is describing. (The difference between these attributes will be discussed later). After being described, or even just declared (where there is no immediate description), these resources may be referenced from anywhere else in the document, provided that the reference occurs after the resource has been declared.

Properties in RDF/XML

Properties are represented as elements within a <rdf:Description> element. The RDF statements about **www.cs.stir.ac.uk/˜lsd**, shown in 5.2.2, could be represented in RDF/XML as:

```
<rdf:Description ID ="www.cs.stir.ac.uk/˜lsd">
   <Title>"Liam's Departmental Page"</Title>
   <Author>www.lsd.org</Author>
</rdf:Description>
```

One important feature this example shows is the use of literals within RDF/XML descriptions. As previously mentioned, the value of a property can be a literal value, such as a name, or another resource. This is conveyed within the syntax, with values of properties which are literal enclosed in single quote marks. Values which are not literal are objects, and may have been previously defined in the document. As mentioned, resources need to be declared before they can be referenced. When translating RDF statements to RDF/XML, the translator may encounter a resource as a property value. If this is the first time the resource has been encountered, there will be no pre-existing declaration of the resource. RDF/XML avoids this issue by allowing resources to be declared within the body of the property element. For example, suppose **www.lsd.org** had not yet been declared within the document, the above XML statements could be rewritten as:

```
<rdf:Description ID ="www.cs.stir.ac.uk/~lsd">
   <Title>"Liam's Departmental Page"</Title>
   <Author>
      <rdf:Description ID ="www.lsd.org" />
   </Author>
</rdf:Description>
```

In simple terms, the document reads that the Author of **www.cs.stir.ac.uk/~lsd** is a new resource by the identity of **www.lsd.org**. Properties in RDF/XML may occur in any order, and have no constraints on their cardinality. As long as the properties of a resource are stated within the resource description, the description will be considered valid.

Declaration and Definition

As mentioned earlier, it is enough for a resource to simply be declared for it to be referenced. A resource can be declared in one part of the document, and then defined later in the document (or even in a separate document). This allows the syntax to be robust, with an ability to function if nothing else is known about the resource apart from that it exists. It also allows for a distributed approach to describing resources, allowing different documents to describe different aspects about the domain.

Namespace

Using XML allows RDF/XML to leverage namespaces within documents. If properties or resources are defined external to the document, a fully qualified URI is required for the document to be parsed correctly (and for the RDF data to be valid). In situations where an RDF document contains many external references, the vast majority of the URI text may be identical (for example, if the document makes several references to single external document). The namespace of the external document is declared at the start of the RDF/XML document with a local reference name, which can be substituted throughout the document. For example if our RDF/XML document referenced many properties defined at www.cs.stir.ac.uk/~lsd/descriptions, the RDF/XML document would contain the following namespace declaration:

```
rdf:RDF xmlns:lsd="http://www.cs.stir.ac.uk/~lsd/descriptions#"
```

An application parsing the RDF/XML description would replace the 'lsd' namespace with **http://www.cs.stir.ac.uk/~lsd/descript** This would allow descriptions to be shortened, for example:

```
<rdf:RDF xmlns:lsd="http://www.cs.stir.ac.uk/~lsd#"
   ...>
   <rdf:Description ID ="www.cs.stir.ac.uk/~lsd">
      <lsd:Title>"Liam's Departmental Page"</lsd:Title>
      <lsd:Author>www.lsd.org</lsd:Author>
   </rdf:Description>
   ...
</rdf>
```

In a similar manner, we can declare a base namespace for the current document. This allows us to define a set of terms which are relevant to our document, without having to provide the full URI for each resource.

5.2.5 Issues in RDF

RDF/XML was a successful step forward in providing meta-data within a domain, due to the simple and light-ruled approach to expressing RDF data. RDF/XML however was simply a syntax for tagging important values within a domain description document. It allows a meta-data description about a resource to be stated (such as **www.cs.stir.ac.uk/~lsd**), but it does not define what the resource represents. For example, **www.cs.stir.ac.uk/~lsd** is a webpage, but this is not conveyed in the RDF description.

Similarly, properties are expressed through simple markup tags, and do not impose any kinds of restrictions on what the value represents. For example, the on-line CD store may describe the cost of a CD using a 'Price'

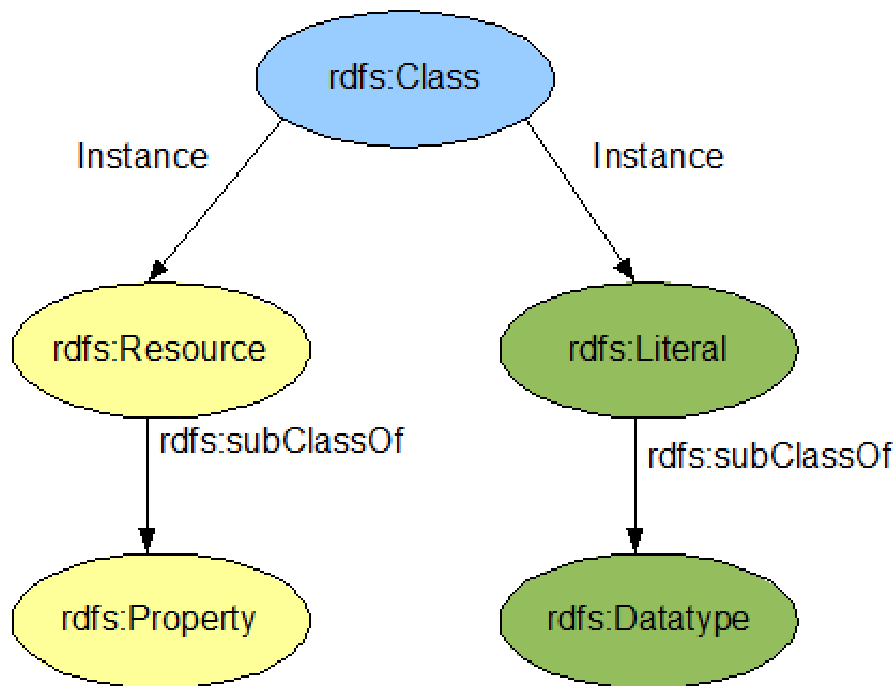


Figure 5.3: RDFS Relations

property. One store may represent this cost as <Price>'9.00'</Price>, another as <Price>'£9'</Price>, and a third as <Price>'Nine Pounds'</Price>. This potential for ambiguity limits the usefulness of using RDF/XML across domains, as each party may have a different intention for a syntactically similar property. These issues were addressed by the extension of RDF/XML, through the RDF Schema.

5.3 RDF Schema

The RDF Schema (or RDFS) [42] was developed to facilitate the need for extra semantics within RDF descriptions. It was designed to allow richer descriptions of web resources, and the terms used to describe them. RDFS allows the use of classification terms, such as 'instance' and 'subclass', when describing the domain. Leveraging these logical features, rudimentary inference and reasoning can be performed over descriptions. The schema brings additional features which allows information to be embedded into resource and property descriptions.

5.3.1 Resources in RDFS

Resources within RDFS take on logical properties which allow them to be identified as a given type of resource. Class concepts are introduced to allow resources to be grouped together under a single category of class. For example **www.cs.stir.ac.uk/~lsd** and **www.lsd.org** could be declared as members of the Website class. The following elements have been introduced through RDFS for conveying this feature.

rdfs:Class

All resources are an instance of rdfs:Class. This element represents the root of all classes. Figure 5.3 displays the relationships between rdfs:Class and the other main RDFS classes.

rdfs:Resource

As has been discussed, all things described through RDF are resources. All resources are an instance of rdfs:Class. This means that a resource always belongs to an instance of a rdfs:Class element, even if it is the root rdfs:Class

element.

rdfs:Literal

Properties may have literal values, and RDFS introduces the rdfs:Literal class to facilitate this. The rdfs:Literal class is an instance and subclass of rdfs:Class. An instance of a rdfs:Literal class is also a subclass of rdfs:Resource.

rdfs:Datatype

This element represents the class all valid RDF datatypes, as specified by the RDF syntax. A rdfs:Datatype element is a subclass of rdfs:Literal.

5.3.2 Properties in RDFS

The intention of property elements within an RDF description was to convey a relationship between two resources. Semantically, the actual RDF property element conveyed no real meaning. Conventionally, the type of property used to describe a relationship should convey, in some sense, what the types of of agents are involved. For example, consider the RDF statement **(QueensGreatestHits, hasMusicGenre, Rock)**. The property **hasMusicGenre** would convey that the subject of the property is a MusicAlbum, and the object is a member of the MusicGenre class. It could then be logically determined that **QueensGreatestHits** is therefore a **MusicAlbum**, and **Rock** is therefore a **MusicGenre**.

RDFS allows semantic meaning to be included in resource properties definitions through the following elements:

rdfs:Property

This element represents a RDF property, and is an instance of rdfs:Class.

rdfs:subClassOf

As mentioned in section 5.3.1, using Class elements allow resources to be grouped into specific categories. Resources can be defined to be members of a particular class using the rdfs:subClassOf property. For example, the RDF statement **(QueensGreatestHits, rdfs:subClassOf, MusicAlbum)** defines the resource **QueensGreatestHits** to be a member of the **MusicalAlbum** class.

Subclass relationships provide logical relationships between resources. For example, if A is a subclass of B, and B is a subclass of C, then A is also a subclass of C. The deducing of the relationship between A and C is considered a forward chaining relationship, where successive relationships provide additional entailments. In logical terms, the rdfs:subClassOf property is transitive property. If our **Website** class was considered a subclass of the **Document** class, it could be inferred that **www.lsd.org** was also a member of the **Document** class.

In RDFS the subclass property observes the same limitations as the logical subclass property. The subclass property is not symmetrical, and so it does not immediately follow on that all **MusicalRecording** members are also members of the **QueensGreatestHits** class.

Property domains and ranges

RDFS introduces terms to convey some semantics about properties. A property **domain** represents the class or classes of resources which may be described by the property. For example, members of the **Building** class would be viable domain members of a **hasAddress** property. A property **range** describes valid property values which may be found in the property description. For example, a member of the **PostalAddress** class would be a valid value for the **hasAddress** property. To convey these property attributes, RDFS introduces the **rdfs:domain**and **rdfs:range** classes.

rdfs:domain

Suppose the domain of the **hasPrice** property, shown previously, was of type **MusicAlbum**. This could be expressed in RDFS as **(hasPrice, rdfs:domain, MusicAlbum)**. This annotates the property to state that any resource found within the domain of the property can be considered of type MusicAlbum.

rdfs:range

Following on from the domain of the **hasPrice** property, the range can be expressed in a similar manner: **(hasPrice, rdfs:range, DecimalPound)**. Using this property, it can be deduced that any resource used as the object of this property can be considered as an instance of the **DecimalPound** class. Using this property, the ambiguity of the example described in Section 5.2.5 can be resolved: The valid representation of CD price would be **(CD, hasPrice, 9.00)**.

rdfs:subPropertyOf

Similar to the rdfs:subClassOf property, rdfs:subPropertyOf is a transitive property which denotes relationships between two properties. Consider a new example. Theres exists a **Person** called **Peter**, and a **Country** called **Spain**. Suppose **Peter** is on holiday in **Spain**, then his current location is also Spain. Two properties have been created to describe these facts, **onHoliday** and **hasLocation**. The relationship between these two properties can be expressed as **(onHoliday, rdfs:subPropertyOf, hasLocation)**. So if **Peter** is on holiday in **Spain**, he is currently located in **Spain**. Again this relationship is not symmetrical, just because he is located in Spain does not mean he is on holiday!

Properties of Properties

As mentioned in Section 5.2, properties are also resources and can therefore have their own descriptive properties. To express this in another way, we can now express meta-data about meta-data. Adding meta-data to properties allows additional logical entailments to be made. For example, suppose a search agent examined the RDF statement **(Peter, onHoliday, Spain)**. The first entailment which could be made has been described previously, that **Peter** is also located in **Spain**. The second entailment which can be made is that the resource **Spain** is an instance of the **Country** class.

This may seem a trivial example at first, but given a complex search criteria, the advantages of RDFS are clear. Suppose a company held a list of where all their employees were currently located in the world. The company wishes to discover all employees which are currently located in countries where the climate is classed as warm. The entailments which can be made from the above statement provide a great deal of information which the search agent would find useful. Firstly, **Peter** is in a **Country**, and may be a match for the search criteria. To check the second part of the criteria, the agent would require to parse all information about **Spain**, which may be in the company list, or may be external. On returning from the information retrieval, the agent concludes that **Spain** is a warm country, and returns **Peter** as a match for the search.

5.3.3 RDFS Issues

RDFS presents a solid base for describing resources, by embedding simple semantics into descriptions. Using the syntax, users can provide a structured description of resources, where properties are defined to eliminate ambiguity, and developers can convey a level of *meaning* in their descriptions. In environments where these descriptions are tightly controlled, search agents are able to function by using known descriptive terms in their search process. RDFS descriptions, however, are still unable to provide rich representations or address the issues of 'syntactically different, semantically similar' vocabularies [52]. For example, there is no scope for a search agent to discover similarities between a **MusicCD** and a **CDMusicAlbum**, as both classes are still described using literal string values. In order for similarities to be detected and resource classifications to be better represented, a further level of definition is required.

5.3.4 The Semantic Web

The Semantic Web is a vision for the Internet, where information is well defined and represented in a logically rich manner [22]. In this vision, search agents can discover desired data and resources in an *intelligent* manner, based upon descriptions using machine-understandable terms and logical rules. Both RDF and RDFS are part of the Semantic Web vision ([22]). A further language, namely the Web Ontology Language, is also used to address the logical aspects of the vision, allowing for issues, such as 'syntactically different, semantically similar', to be resolved at the machine level.

5.4 The Web Ontology Language

The Semantic Web is concerned with providing better definition to information available within the Internet.

In order for agents to 'understand' information, the data requires to be described at a level above that of RDFS. The Web Ontology Language (OWL) [45] has been developed on top of the RDF/XML format, and leverages all properties of RDFS. OWL provides additional property information which moves towards being able to describe a resource in a logical manner, and as such allows machine processing to perform reasoning and inference upon the data.

Defining the properties of a class, allows for automatic classification based upon available information. For example, suppose a webpage is classed as a document which also has a URL. On parsing the description of **www.lsd.org**, an agent would identify that the document has a URL (which is www.lsd.org) and as such can be considered a webpage. This information need not be explicitly stated in the original description as it can be inferred from the context. In this example, the context is the classification rule that all documents with URLs are webpages.

5.4.1 OWL Resources

OWL can be used to provide template descriptions of resources, in order to describe exactly what constitutes a particular resource. For example, a **CD Music Album** could be defined as a **CD** which contains one or more tracks of music. Any **CD** resources which contain at least one track of music can therefore be classified as a **CD Music Album**.

OWL brings a number of additional expression which can be applied to a resource.

owl:equivalentClass

Using our above example of the **CD Music Album** resource, suppose another party defined their own resource, **Music CD**. If these two resources shared the same domain space, and both definitions were equal (both were defined as a **CD** with at least one track of music), then a **CD Music Album** would be equivalent to a **Music CD**. This fact can be expressed in OWL as **(CD Music Album, owl:equivalentClass, Music CD)**. Any agent encountering an instance of a **CD Music Album** could treat the resource as if it were a **Music CD**. This property is symmetrical, and so the statement **(MusicCD, owl:equivalentClass, CDMusicAlbum)** is implied.

owl:disjointWith

Suppose there exists two similar classes, which contained similar properties but were semantically different. Due to the logical nature of the language, two classes may be deduced to be similar, due to a lack of information to the contrary. In such situations, the owl:disjointWith property can be used to explicitly declare two resources as being logically distinct.

For example, suppose an agent encountered the statement **(Car, owl:disjointWith, Bus)**. In this case, the description is stating that a **Car** cannot be a **Bus**, and vice versa (the property is also symmetrical). In real world terms a car is likely to be owned by a person, and a bus owned by a company. This information may be missing from the ontology, but it is enough for a developer to declare the two classes to be distinct.

This property can be used to detected inconsistency within descriptions, in situations where a resource would appear to be both a **Car** and a **Bus**.

OWL Individuals - owl:Individual

Within OWL, an Individual is an instance of a class member. For example, **Ford** may be an individual of the class **CarManufacturer**. An individual can be related through properties to other individuals, but they may not be related to other resources. An instance of a class member can be annotated with owl:Individual, for example **(Ford, owl:Individual, CarManufacturer)**.

owl:sameAs, owl:differentFrom

As owl:equivalentClass can only be applied to resources, owl:sameAs is the property for expressing the same relationship between individuals. For example **(Ford, owl:sameAs, FordUSA)** states that **Ford** and **FordUSA** can be considered the same individual. In a similar manner owl:differentFrom allows individuals to be declared

logically distinct from each other. Both properties are symmetrical, in that if it is declared **(A, owl:sameAs, B)**, then it can be infered that **(B, owl:sameAs, A)**. Both properties are also mutually exclusive with each other. It is logically inconsistant for a description to contain statements **(A, owl:sameAs, B)** and **(A, owl:differentFrom, B)**.

5.4.2 OWL Properties

OWL provides a range of new property attributes to enhance descriptions. The key to machines being able to process, compute and discover new data is by expressing attributes of resources in the most descriptive manner possible. In this way, all information about a given resource may not be explicitly stated, but instead implicitly inferred from the property attributes. The OWL syntax introduces the following property attributes.

owl:equivalentProperty

Two properties may share the same intention. For example **ownsProperty** and **ownsBuilding** may both convey the same relationship between a **Person** and a **Building**. In such cases the properties may be annotated to show equivalence: **(ownsProperty, owl:equivalentProperty, ownsBuilding)**. This property is symmetrical.

owl:inverseOf

The inverse of a property can be thought of as the reverse relationship between two classes or concept. For example, **hasParent** describes a relationship between a child, which is the subject or domain of the property, and a parent, which is the object or range of the property. The inverse of this relationship is **hasChild**, which describes the relationship from the parent point of view (i.e. the parent is the subject and the child is the object). This can be stated as **(hasChild, owl:inverseOf, hasParent)**.

owl:ObjectProperty

As previously discussed, a property may have a rdfs:Resource or rdfs:Literal element as its value. In section 5.2 it was also mentioned that a property is also a resource. OWL contains two classes of properties which are both subclass of rdfs:Property. The first of these is owl:ObjectProperty. An object property always as another resource as its value, for example **(Lamp, hasLocation, LivingRoom)**. **LivingRoom** is a resource described elsewhere, and can assumed not to be a literal value. Object properties are declared in a similar fashion to rdfs:Property objects in RDF/XML: <owl:ObjectProperty rdf:ID='hasFriend'/>.

The fact that this property is an object property can assist in consistency checking and error detection. Agents can assume that if the property is of type owl:ObjectProperty, that the value of the property is always a resource.

owl:DatatypeProperty

Datatype properties always have a literal value as its value, for example **(Lamp, hasSerialNumber, '193F3GF8U')**. The property hasSerialNumber always point to a string representation of a device's serial number. In this case, hasSerialNumber can be said to be a Datatype property and is expressed in RDF/XML as <owl:DatatypeProperty rdf:ID='hasSerialNumber'/>. Values of Datatype property objects are never assumed to be concrete resources.

owl:SymmetricProperty

The OWL property attributes described so far are all symmetrical properties. OWL also allows for explicit declaration of this attribute within a property description. For example, suppose the property hasFriend describes the relationship between two Person resources. This property can be said to be symmetrical as if **(Peter, hasFriend, Alice)** then the inverse is also true, **(Alice, hasFriend, Peter)**. This property attribute is itself symmetrical. In RDF/XML, this attribute is expressed in the following manner:

```
<owl:ObjectProperty rdf:ID="hasFriend">
   <rdf:type rdf:resource="&owl;SymmetricProperty"/>
<owl:ObjectProperty/>
```

This denotes that the property is both an ObjectProperty and a SymmetricProperty. Datatype properties cannot be symmetrical, for obvious reasons.

owl:FunctionalProperty

Functional properties are unique in that the object of the property may only have one instance of the property. For example, hasMainAddress describes the relationship between a Person and their main point of residence. A **Person** may only have one main residence and so the **hasMainAddress** property is a functional one. This can be expressed in RDF/XML as:

```
<owl:ObjectProperty rdf:ID="hasMainAddress">
   <rdf:type rdf:resource="&owl;FunctionalProperty"/>
</owl:ObjectProperty>
```

Functional properties are 'one-way' properties. A **Person** may have one main residence, but that residence may be the main residence for many **Person** objects. The **Person** is not unique within the residence, but the residence is unique to the **Person**. This is true within the OWL domain, and is taken into consideration during consistency and error checking.

owl:TransientProperty

Transient properties denote forward changing relationships between resources. For example, the hasAncestor relationship is a transient property describing relationships between family members. If **(Anna, hasAncestor, John)** and **(John, hasAncestor, Claire)** then it follows that **(Anna, hasAncestor, Claire)**. The relationship always points forward, Anna -> John -> Claire. Transient properties are not intrinsically symmetrical properties (i.e it does not follow that Claire -> John -> Anna) but if annotated with an inverse property can also be backward changing too.

xsd datatypes

As discussed in section 5.3.2, RDFS allows the specification of the range and of a property. This range can be a Class or a Literal. OWL allows further definition in situations where the value is literal. Rather than specify an abstract class, OWL allows specifying a particular datatype, mainly corresponding to a built-in XML Schema datatype. For example, the range of a hasName property would specify that the value is a String. This can be expressed in RDF/XML as:

```
<owl:DatatypeProperty rdf:ID="hasName">
  <rdfs:domain rdf:resource="#Person" />
  <rdfs:range  rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
```

owl:Ontology

OWL allows ontologies to embed information about the ontology document itself, such as version and purpose information (in a human readable format). The owl:Ontology element is a class defined to contain such information, and is commonly found at the beginning of the document. It can be expressed in the following manner:

```
<owl:Ontology rdf:about="">
  <owl:versionInfo>v2.4 2008/06/13</owl:versionInfo>
  <rdfs:comment>A Music CD ontology</rdfs:comment>
  ...
</owl:Ontology>
```

Properties contained within the owl:Ontology element are all instances of the owl:OntologyProperty class. Developers can create their own meta-data header properties to be contained within this element. An important property which can be included within the owl:Ontology class is owl:imports which specifies external ontologies which are referenced within the document. This property is discussed later in section 5.5.1.

5.4.3 OWL Restriction Classes

As mentioned in section 5.4.1, classes can be defined to have particular attributes associated with them. If a resource displays these properties, it can then be logically classified as a member of that class. Recall the definition of a **CD Music Album**. If a resource is described as being a **CD** and contains at least one music track, it can then be classified as a **CD Music Album**.

OWL provides scope for classification by using restrictions on properties and their values. For example, suppose the description of a **CD** contained the statement **(CD, containsMedia, MusicTrack_A)**. For a CD to be classified as a **CD Music Album**, there must exist a **containsMedia** relationship to an instance of a **MusicTrack**. In this example, the restriction is that any instance of the **containsMedia** relationship must point to a **MusicTrack** resource. If this restriction is satisfied, then the owner of the description may be classified as a **CD Music Album**.

OWL expresses these relationships through the following elements:

owl:Restriction

The Restriction class contains information about the kind of restriction on the resource. All restrictions are expressed through a Restriction class.

owl:onProperty

This element denotes which property has the restriction placed upon it (e.g. **containsMedia**). Two kinds of property restrictions can be placed upon a property.

owl:someValuesFrom

As already specified, for a resource to be considered as a **CD Music Album** it must contain at least one music track. The restriction (denoted by owl:onProperty) is on the **containsMedia** property. There may be many instances of this property within a resource description, but if at least one instance has a MusicTrack resource as its value, then the restriction is satisfied. This constraint can be captured using owl:someValuesFrom. This property denotes the required class which must appear at least once in the resource description. It can be expressed in RDF/XML as:

```
<owl:Class rdf:ID="CD_Music_Album">
   <rdfs:subClassOf>
     <owl:Restriction>
       <owl:onProperty rdf:resource="#containsMedia" />
       <owl:someValuesFrom rdf:resource="#MusicTrack" />
     </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>
```

In this example, the value of the owl:someValuesFrom property is MusicTrack.

owl:allValuesFrom

As owl:someValuesFrom represents the case where at least one member of a class must be present, owl:allValuesFrom represents the other end of the spectrum. Using the owl:allValuesFrom property within a restriction specifies that for all instances of the restricted property, every value must be of a particular class. For example, suppose a **CD** could only be declared to be a **CD Music Album** if all media resources on the **CD** were music tracks. This would mean that if a **CD** contained anything other than music, it could not be said to be a **CD Music Album**. In a similar manner to owl:someValuesFrom, the owl:allValuesFrom property is expressed through RDF/XML as:

```
<owl:Class rdf:ID="CD_Music_Album">
   <rdfs:subClassOf>
     <owl:Restriction>
       <owl:onProperty rdf:resource="#containsMedia" />
       <owl:someValuesFrom rdf:resource="#MusicTrack" />
     </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>
```

owl:cardinality, owl:minCardinality, owl:maxCardinality

One of the most simple property restrictions found in the logical domain is cardinality. OWL includes cardinality properties with the syntax for simple restrictions of properties. Descriptions can specify how many times a property instance may occur within a description. For example, a family ticket for entrance to an amusement park specifies that no more than two children can gain entry on the ticket. This cardinality restriction can be expressed through RDF/XML as:

```
<owl:Class rdf:ID="FamilyTicket">
   ...
      <owl:Restriction>
         <owl:onProperty rdf:resource="#adultEntry"/>
         <owl:cardinality rdf:datatype="&xsd;int">2</owl:cardinality>
      </owl:Restriction>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#childEntry"/>
         <owl:maxCardinality rdf:datatype="&xsd;int">2</owl:maxCardinality>
      </owl:Restriction>
   ...
</owl:Class>
```

In this example, the description specifies that no more than 2 children, and exactly 2 adults, can be admitted on a family ticket.

5.5 Domain Descriptions

Defining complete domains can be a long and difficult task. Ensuring that all important concepts are captured accurately and all domain properties are represented in a relevant manner can require an iterative approach. Issues can occur when describing small specific domains or large generic domains. Some issues are particular to the size of domain being described, while others can appear in any domain. To alleviate some issues involved when describing domains, OWL allows developers to import existing ontologies into their domain ontologies. This section will discuss the syntax used, and highlight the issues which can be alleviated by ontology reuse.

5.5.1 Using owl:imports

As mentioned in section 5.4.2, OWL allows for meta-data about the ontology to be embedded within the document. References to external ontologies reused within the document are contained within the Ontology element. The owl:imports property is used to indicated the ontologies required for any machine-based management of the ontology. A component which is responsible for managing all aspects of ontology use, such as consistency checking, inference and reasoning, requires all relevant data to be available. If Ontology A references Ontology B, then the component would require both Ontology A and B within its domain model. An ontology which reused elements from the ontology contained at www.lsd.org/ontology would include the element:

```
<owl:Ontology rdf:about="">
  ...
  <owl:imports rdf:resource="http://www.lsd.org/ontology"/>
</owl:Ontology>
```

5.5.2 Describing Small Domains

Describing a small domain may allow those who describe to also develop the search agents for that domain. In this manner, developers can capture classes and properties which will be relevant to their purpose. Complete descriptions may not be required, as the important attributes will be based on a particular viewpoint. For example, if a developer was to create an ontology to describe a customer list, they may define a **Customer** class. It may

```
<owl:Class rdf:ID="Customer">
   <owl:equivalentClass>
      <owl:Class>
         <owl:intersectionOf rdf:parseType="Collection">
            <owl:Restriction>
               <owl:onProperty rdf:resource="#hasAddress"/>
               <owl:allValuesFrom rdf:resource="#Address"/>
            </owl:Restriction>
            <owl:Restriction>
               <owl:onProperty rdf:resource="#hasCustomerNumber"/>
               <owl:allValuesFrom rdf:resource="#CustomerNumber"/>
            </owl:Restriction>
            <owl:Restriction>
               <owl:onProperty rdf:resource="#hasName"/>
               <owl:someValuesFrom rdf:resource="#CustomerName"/>
            </owl:Restriction>
         </owl:intersectionOf>
      </owl:Class>
   </owl:equivalentClass>
</owl:Class>
```

Figure 5.4: Description of a Customer class

be enough for this domain that a **Customer** class contains only a **CustomerNumber**, a **CustomerName** and an **Address**. This class is expressed in Figure 5.4.

From a different viewpoint, it may be important to also capture extra attributes about the **Customer**, such as **CardNumber**, **SecondAddress** and **PhoneNumber**. In small specific domains, descriptions can be tailored to the purpose of the agents. Two similar parties can be concerned with the same domain (e.g. Customers) but describe the domain differently based upon their intentions. In such situations it may be acceptable for both descriptions to remain logically independent from each other. The intention of the OWL syntax, however, is to unite domains which have similar purposes and meaning. In this instance, the second domain, which requires more detailed information, can reuse the description of **Customer** already specified, and then add additional attributes. Suppose our initial ontology was found at the web location www.lsd.org/customer. An ontology about the second customer object, called **DetailedCustomer** can be described using a mixture of new and imported terms, shown in Figure 5.5.

This document would be considered syntactically valid as a parser can retrieve the initial ontology as defined by the owl:imports property. As **DetailedCustomer** is defined as being a member of the **Customer** class (through the rdfs:subClassOf property), it is subject to the restrictions imposed upon the **Customer** class, as well as those specified in **DetailedCustomer**. An agent designed to search through **Customer** objects would now be able to operate in a domain concerning **DetailedCustomer** objects, without any need for modification.

Reusing ontologies within small domains allows developers to base their work upon a common root ontology, and then expand descriptions for their own purpose.

5.5.3 Describing Large Domains

Suppose a developer wishes to define an ontology which is concerned with on-line media sales. Such a domain may involve concepts such as CDs, DVDs, Books, Computer Games and downloadable music. To specify such a large ontology may become difficult, as there can be many concepts and attributes which require to be accurately and consistently described. There is also issues with determining the scope of the ontology. For example, is it enough for the ontology just to describe the media, or should it also describe the website?

Scope

The scope of an ontology can be determined by purpose of the ontology. If the reason for developing the media ontology is to support our search agent (and enable other search agents) then the scope is concerned with de-

```
<rdf:RDF
   xmlns:cust="http://www.lsd.org/customer#"
   ... >

   <owl:Ontology rdf:about="">
      ...
      <owl:imports rdf:resource="http://www.lsd.org/customer"/>
   </owl:Ontology>

   <owl:Class rdf:ID="DetailedCustomer">
        <owl:equivalentClass>
            <owl:Class>
                <owl:intersectionOf rdf:parseType="Collection">
                    <owl:Restriction>
                        <owl:onProperty rdf:resource="#hasEmailAddress"/>
                        <owl:allValuesFrom rdf:resource="#EmailAddress"/>
                    </owl:Restriction>
                    <owl:Restriction>
                        <owl:onProperty rdf:resource="#hasPhoneNumber"/>
                        <owl:allValuesFrom rdf:resource="#PhoneNumber"/>
                    </owl:Restriction>
                    <owl:Restriction>
                        <owl:onProperty rdf:resource="#hasSecondAddress"/>
                        <owl:allValuesFrom rdf:resource="#SecondAddress"/>
                    </owl:Restriction>
                </owl:intersectionOf>
            </owl:Class>
        </owl:equivalentClass>
        <rdfs:subClassOf rdf:resource="&cust;Customer"/>
    </owl:Class>
```

Figure 5.5: Description of a DetailedCustomer class

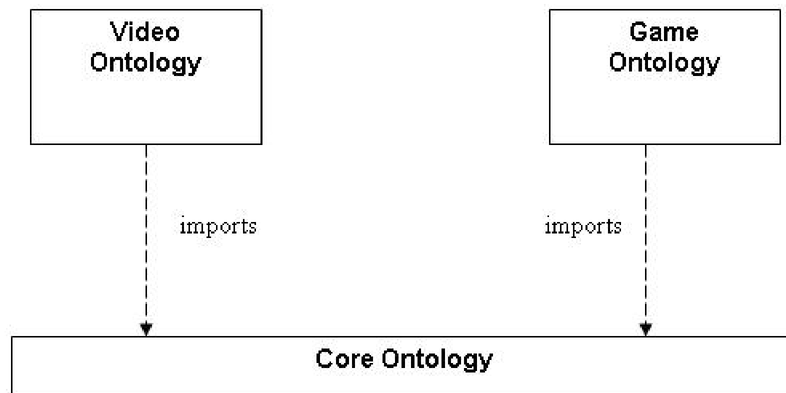


Figure 5.6: Relationships through Ontology Reuse

scribing media. If, however, the purpose is to allow external agents to discover our available media, and enable indirect purchasing (where third party agents act on behalf of clients), then the scope may well be concerned with describing our whole environment, including the webpage, accepted payment types and delivery details.

Ontology arrangement

When developing large scale ontologies, which may have more than a single purpose (create an internal description catalogue AND allow for indirect purchasing), defining all information within a single ontology document can be an inefficient approach. As the amount of information grows, the size of the document grows, which in turn can increase parsing time. Single files can also suffer from inconsistencies, as one invalid statement makes the whole document inconsistent.

The solutions to consistency and accuracy issues which were addressed in small domains can be leveraged in large domains. Rather than have a single large ontology which describes a large generic domain, we can break the ontology into smaller ontologies. Each ontology would describe a particular subdomain. This approach can then facilitate ontology reuse, where new information is built upon existing information.

Suppose we have a Core ontology which describes important concepts within the overall media domain. This ontology would specify classes such as Author, Title, Genre, Developer and Price. A new ontology, which is concerned with film media (DVDs), is then created to deal with aspects of the film domain. In this ontology, a new class called 'Film Genre' is created and declared to be a subclass of the Genre class in our existing ontology. Subsequent Film Genre classes can then be described, such as Action, Romance and Comedy. A similar ontology is created which deals with the Computer Game ontology. This ontology also contains a subclass of Genre, 'Game Genre', which then has various subtypes such as Action (which is logically distinct from Action in the Film ontology), Sports and Strategy. In this manner, as each ontology is processed and declared consistent, new ontologies can rely on the information already expressed. The relationship between the ontologies can be seen in Figure 5.6.

Having all domain information within one document also forces any developers using the ontology to have all the information in memory. For example, if a developer was creating ontologies about specific films, they would initially require to have all ontology information about the Film domain in memory. If this information was contained within a single ontology document, along with the Game domain, and Music domain, then a large amount of redundant information would be loaded into memory unnecessarily. Using the above example, describing a DVD would only require knowledge of the Film ontology and the Core ontology. All information about Games, Books and Music is not needed, and is therefore not required to be loaded into memory. In addition, an online media store may reuse multiple ontologies describing Games, Books and Music.

5.6 Ontology Conclusion

Logic-based ontologies provide a unique approach to describing domains, and unifying multiple vocabularies (provided they are expressed in the same language). Using the ontology elements described, developers can create a vocabulary which contains implicit rules, which can allow reasoning and inference to be performed on the

knowledge base. In this manner, environments described by ontology languages provide better support for search agents, than keyword-based or syntax-based search environments. Descriptions can convey a level of *meaning* through the logical properties of the language. This characteristic would seem well suited to describing network domains.

Chapters 6 and 7 discuss existing network-related frameworks and projects which may be suitable for addressing home network issues. Chapter 6 describes the environment of web services, while Chapter 7 describes relevant ontology-based work within the home network domain.

Chapter 6

Ontologies Within Web Services

Web services are an existing network domain where ontologies are being applied in order to promote a more automated and semantically rich approach to both service descriptions and service usage [102, 20]. This domain has emerged as a candidate environment for the use of ontology languages. Work from this domain has motivated the appliance of ontology languages to other domains, such as peer-to-peer, workplace and home networks.

This chapter will introduce web services, and discuss the aims behind the technology. This chapter will then discuss relevant work within the Web Service domain, and critique existing work on its suitability to other domains, such as the home network. This chapter will conclude by highlighting the different approaches and applications of ontologies to achieve a variety of goals.

6.1 Web Services

A web service is a software application which performs a specific function, much like a service within the home network. Rather than limit the domain of use to a small network area, a web service can be used by any qualified machine connected to the internet [103]. A service offered in Edinburgh, Scotland may be used by a client service in Sydney, Australia.

An example of a web service is a hotel booking service which allows clients to book a reservation. Typically such a service can be represented by webpage interface. This requires any clients to interact using web forms, which can be difficult to automate. Rather than require users to explicitly visit a service's webpage in order to use the service, a web service publishes a service description which can be used to remotely interact with the service provider. In a similar manner to common home networking protocols, web service registries can maintain a repository or directory of web services. By utilising registries, web services can be combined to provide complex applications [54]. For example, a price comparison application can provide information on the best prices for a particular product. This can be achieved by a main web service aggregating various price lists which are each represented through a web service. Thus, the price comparison application removes the need for a human user to perform various quotation requests sequentially.

As clients and providers may be operating on differing platforms, a common approach to service interaction is required. This protocol will allow service providers and users to exchange messages understandable to both parties. To satisfy these goals, the Web Service Description Language was created.

6.1.1 Web Service Description Language

The Web Service Description Language (WSDL) [46] is a XML-based specification for describing services in order to allow protocol independent interoperation. A WSDL description defines a service as an abstract set of messages, message types and actions or functions, which operate on a network endpoint. A WSDL description then binds the abstract descriptions onto a concrete description specific to a particular implementation or protocol.

In simple terms, the abstract description allows WSDL to define a lightweight description of the parameters used within the message passing. Parameters may be a single single element, such as a string, or a combination of elements. For example, a BookRequest object may involve both an Author and PublicationDate. The abstract description also includes a representation of a function offered by the service. For example, a BookLocation function may require a BookRequest object as input. The concrete part of the description provides a binding

between the abstract description, and a protocol specific implementation of the service. Each function will have a binding onto a network endpoint, which is the point of contact with the network. A user can retrieve the protocol specific details and interoperate with the service.

Messages are exchanged between providers and users using the SOAP protocol described in section 3.2.9

WSDL allows providers to describe *how* to interact with the service. In order to allow web clients to discover services, an approach to describing *what* the service does is required. To this end, a standardised approach to web service descriptions and service registries has emerged in the form of Universal Description Discovery and Integration or UDDI for simplicity.

6.1.2 UDDI

UDDI provides an XML-based directory service for web services. Clients query a UDDI registry using SOAP (see section 3.2.9) to discover suitable services for use, and retrieve WSDL descriptions of those services. UDDI was designed to allow businesses to list their services in a manner which would allow other businesses to discover and utilise them [33].

UDDI Description

A UDDI description comprises of a mixture of human and machine readable elements. A description contains 3 main elements:

- White Page: This element contains basic contact information about the service. This information may include details about the company offering the service, and contact details of personnel involved with the service (for such actions as billing and technical support).
- Yellow Page: This element describes *what* the service does. This element contains references to external taxonomies which represent classifications of a service. A number of taxonomies may be referenced to increase the chances of the service being discovered.
- Green Page: This element contains the operational details of the service, namely the WSDL description. Interrogating this element allows a client to make use of the service.

The Yellow Page element is used in the discovery process, notifying the taxonomy classifications of which the service adheres to. There is, however, no explicit relationship between the taxonomy classification and the implementation described in the Green Page element.

UDDI Nodes

A UDDI node is a server which supports the UDDI specification, and contains a collection of services. A node may be owned by a single company, providing a collective storage for UDDI descriptions.

UDDI Registries

A UDDI registry is a collection of one or more UDDI nodes, and acts as a listing of available web services. Registry users require to know the location of a UDDI registry before searching the directory. A registry can limit responses depending on the type of service offered, either private (limited to certain clients) or public (available to all clients). This lightweight approach to restricted access allows businesses to offer access to their UDDI registry to the public, while ensuring that critical or sensitive services listed are unusable.

Discovery

UDDI offers an attribute/value approach to providing meta data about services. Such meta-data can describe a number of attributes, such as geographical location, cost-per-use, and reliability. Issues surrounding this approach have been discussed in Section 4.2.2. UDDI attempts to circumvent these issues by allowing multiple taxonomy attributes to be cited, increasing the chance of an attribute matching. Due to the syntax-based matching, there is no scope for expressing the relationship between attributes. The onus is therefore on the client, as much as on the provider, to submit as many possible representations of a semantic concept. In other words, both provider and user may be required to submit multiple attributes which represent the same semantic concept, in the hope of finding a successful match.

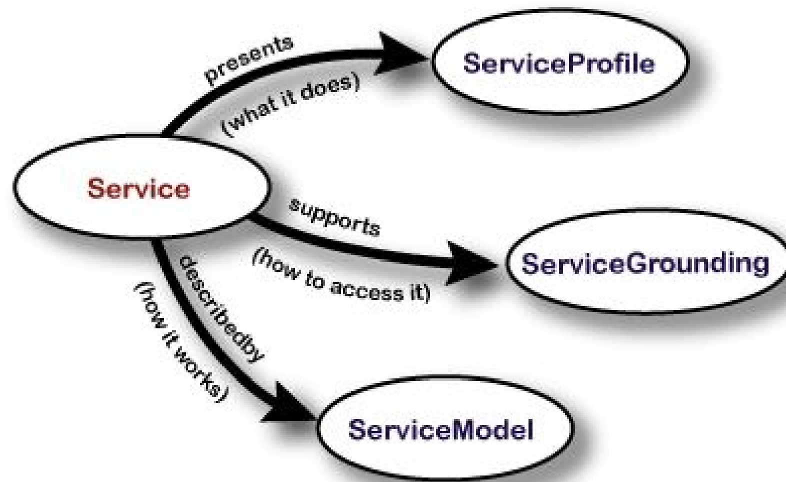


Figure 6.1: Overview of an OWL-S Description

The Success of UDDI

The structured approach towards the description of web services offered by UDDI has led to the specification being integrated into the Web Service Interoperability standard (WS-I). This standard is defined by the Web Services Interoperability Organization [100], and is concerned with creating guidelines and testing procedures for interoperability amongst web services. Work also exists towards providing a degree of *security* within UDDI registries, and WSDL descriptions [2].

UDDI is very much suited to tightly controlled environments where a specified taxonomy for categorisation and attributes is adhered to by all parties. Expanding this domain to open up UDDI registries to a larger network, such as the internet, may result in a low uptake due to the open-standard approach towards describing the *what* part of a service description [8].

One popular approach to addressing this solution is by describing the *what* part of the service using ontology or logical languages [20, 21, 1]. By providing machine-understandable descriptions, many of the risks associated with wide-domain web service provision can be addressed [78]. One approach to providing ontology descriptions of web services can be found within OWL-S.

6.2 OWL-S

OWL-S [20] was developed to provide a language for richly describing web services. Rather than simply specify some keyword attributes, or syntax-based attribute/value parings, OWL-S provides a schema and vocabulary for describing semantically *what* a service does. This builds upon the simple approach of specifying a category of service, by describing *how* a service carries out its function, and how the *world* is affected by its execution.

OWL-S is built upon OWL, as the name implies. This allows the language to inherit the logical properties offered by OWL. OWL-S also includes scope for describing how a client can interact with a service. This is achieved by fusing OWL-S elements with WSDL descriptions. An OWL-S service description has three main parts (depicted in Figure 6.1):

- A ServiceProfile: This element describes *what* the service does. This element contains a mixture of human and machine readable information about the service.
- A ServiceModel: This element describes *how* the service achieves its purpose. The ServiceModel describes the processes involved for the service to carry out its task.
- A ServiceGrounding: This element describes the bindings between OWL-S elements and WSDL messages. It allows clients to understand how to interact with the service, based on semantic concepts expressed in the ServiceModel.

6.2.1 The Service Profile

The Service Profile part of an OWL-S description is used for the initial matching stage within service discovery. The ServiceProfile firstly contains details about the business or owner of the service. Similar to UDDI, these details can include details on cost of use, business address, or contact details for support staff. While easily readable from a human perspective, these details are captured within a structured schema, and can therefore be extracted by a computer agent.

The ServiceProfile then specifies the operational details of the service. In particular, the inputs, outputs, pre-conditions and post-conditions of the service are specified. The inputs and outputs of a service are described in OWL, and represent concrete data-types. For example, a service may require a **CreditCardPayment** as an input. The data-types required to constitute a **CreditCardPayment** are defined later in the service description. Additionally, services users may already *know* about the CreditCardPayment concept, and can then search the inputs of available services for this concept.

The pre-conditions and post-conditions of a service are intended to describe to a potential service user how the service changes the environment around it. The definition of the service environment is service dependent. The syntax or language used to express these conditions is not defined by the OWL-S specification. The specification is purposely open-ended on these attributes, to allow service providers to use their own preferred approach. The OWL-S overview suggests the Knowledge Interchange Format (http://ksl.stanford.edu/knowledge-sharing/kif/) as one option [20]. The scope for expressing conditions of a service provides an OWL-S description with the ability to describe *what* the service does at a new semantic level.

For example, suppose the purpose of a service user was to make a hotel booking in a specific location. Rather than search over a taxonomy reference, a service user can search on the post-conditions of a service, or in other words search for a service which changes the environment in a certain way ('A hotel room is booked, a credit card is charged').

The decision not to specify a specific approach to describing conditions limits the usefulness of this part of an OWL-S description. With scope for multiple languages being used, discovery over service conditions may be limited to tightly controlled domains. In such domains, the languages used for describing conditions would be agreed in advanced.

The final part to the ServiceProfile contains a list of attributes of the service. The initial attributes classify the service, allowing service providers to refer to external classification taxonomies. The attribute list also includes scope for describing the 'quality' of the service. The definition of quality is not specified, and may refer to the reliability or availability of the service. This set of attributes therefore refers to an external quality-rating taxonomy. The final set of attributes are not specific to a role, and are decided by the developer or vendor of the service. For example, these may describe the location of the service, or average response time.

The list of attributes within a Service Profile can be parsed by any interested service user. This differs in approach from many service discovery approaches. Once a profile is discovered, all descriptive attributes are exposed to the search agent.

6.2.2 The Service Model

The ServiceModel element of an OWL-S is responsible for describing the *process* which the service takes to achieve its purpose. The process is described in terms of atomic and composite processes. An atomic process is a one-step event which closely maps to the description of the service given in the ServiceProfile, in terms of inputs, outputs pre-conditions and post-conditions. A composite process may contain many steps towards achieving the goal of the service. For example, a hotel booking system may first require to find an available room, then authenticate the credit card used, then retrieve payment from the payees account.

The service model describes inputs and outputs used throughout the service invocation, not only between the user and provider, but also between the internal process within the service. It describes how the outputs from one process are used within following processes. It also may describe the post-conditions possible if the service is invoked with some pre-conditions unsatisfied (for example, if a bank account does not have enough funds to settle a bill). By examining the process model, service users can *understand* how the service carries out its function.

6.2.3 The Service Grounding

The ServiceGrounding describes the relationships between concepts expressed in OWL, used in the ServiceProfile and ServiceModel, and elements of the WSDL service description. (Recall, WSDL explicitly describes a service

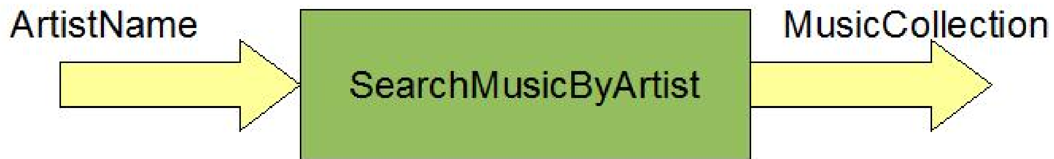


Figure 6.2: The SearchMusicByArtist Service

Input		Output	
ArtistName		MusicCollection	
Known	Unknown	Known	Unknown
FirstName SecondName	*none*	AlbumName	StoreRef

Figure 6.3: Service User Knowledge

in a manner which allows direct service invocation). OWL concepts specified within the input and outputs of the service description are mapped onto WSDL message parts and types. The service itself is mapped onto a WSDL function description. The WSDL description then specifies how to invoke the service.

6.2.4 The Feasible Impact of OWL-S

There are two main issues which require attention if OWL-S is to have a serious impact on web services and other networked domains.

Internal Classes

The first issue which plagues many attempts at providing a standardised framework for discovery, and one which OWL-S attempts to address, is lack of commonly held terminologies [7]. Providing an ontology based framework provides some support for cross-platform or multi-domain discovery. For example, the property **hasInput** within an OWL-S service profile is more than just a string tag. This property is defined as having a OWL-S Profile as the subject, and an Input parameter as the object. The Input parameter is not a simple string value, and instead is an ontology class specified elsewhere within the collection of ontologies describing the service. In this manner, service users can interrogate Input and Output parameters in order to *understand* what they represent.

Despite this new layer of semantic description, service users may still find difficulty in *understanding* Input and Output parameters. Suppose a user of an online music store wishes to discover all music by a specific artist. The music store offers a SearchMusicByArtist service which takes an ArtistName Input parameter and returns a MusicCollection Output parameter. The data flow involved in this service is shown in Figure 6.2.

On further examination of the ArtistName parameter, the service user discovers that an ArtistName class composes of a FirstName and SecondName elements, both of which are of type xsd:string. FirstName and SecondName classes are well known within the domain (perhaps defined in a standardised ontology), and are in turn known by the service user. This *knowledge* allows the construction of the service input. The MusicCollection parameter is defined as a list of MusicAlbum objects. This class contains an AlbumName element and StoreRef element, as defined within the service ontologies. AlbumName is a well known class (again, being defined in a standardised ontology), and is hence *understandable* to the service user. StoreRef represents a unique number given to every album which the store provides. This element is unique also to the online store, and is not necessarily *understood* by the service user. The service user may decipher that the element is of type xsd:string, but not fully grasp what the element semantically represents.

Figure 6.3 shows the classes and knowledge involved in the service invocation from the service user point of view. Despite not explicitly *understanding* what ArtistName represents, it has knowledge of the core elements involved in the class (FirstName and SecondName). This means it can construct an ArtistName object. To the service user, AlbumName may simply represent a collection of FirstName and SecondName elements. It is possible to store this knowledge for future use, and add this knowledge to its own knowledge base. Disregarding this knowledge, however, will not hinder the user making future use of the service, as it can simply rediscover the elements of the ArtistName class.

The MusicCollection class poses a problem to the service user. As Figure 6.3 shows, the service user knows about the AlbumName class. It understands what the class represents, and may be able to separate the class into more basic components (for example, the class may simply be of type xsd:string). StoreRef contains the challenge to the service user as it does not know what the class represents. Output parameters viewed as a whole object may be less important to service users than the elements of the output object. As the user knows about AlbumName elements, this may be all that the user requires to carry on its own goals. It may be sufficient to disregard the StoreRef part of the output.

Suppose to purchase a music album, the service user was required to provide a StoreRef which corresponds to the desired item. It is at this point that having partial knowledge of the parameters used in service usage becomes an important issue. As highlighted by Figure 6.3, it is possible to construct an Input parameter, if the user has knowledge of the elements which compose that parameter. If knowledge about any core elements of the parameter are missing, a complete parameter cannot be formed.

This issue can be addressed by referencing internal classes as little as possible. An internal class type is an ontology class defined within the OWL-S service ontologies, which is not fully constructible using external, well known elements. For example, the MusicCollection Output parameter contained a mix of elements which were well known (AlbumName) and those specific to the music store (StoreRef). In such cases, service users need to be programmed to interact with a specific set of service providers. This is in contrast with the web service philosophy, where services interact based entirely on open standards, including service invocation, and service parameters.

This issue is not quickly solved. A major reason for internal classes stems from a lack of well defined external ontologies or vocabularies. If a term or class does not exist externally for describing a parameter, the developer is forced to define their own. It is imperative that for any ontology based description framework to be successful, well defined external taxonomies must exist.

As an aside, this issue can often be found in environments where service interfaces are used for information exchange [83]. Interfaces must be well known in order for service users to interact with the service provider. Web service languages provide support for situations where interfaces are not well known. Languages, such as WSDL and OWL, are able to represent complex interfaces through simple types. If the simple components are well known, then a complex class can be created 'on-the-fly'.

Pre and Post Conditions

An interesting step forward in approaches to describing services has led to the inclusion of pre and post conditions. OWL-S is not alone in integrating these details into the service description. Other description languages, such as the Semantic Web Services Framework [21] and the Web Service Modeling Ontology [29], have taken a similar approach, although specify these conditions in differing ways.

As described in section 6.2.1, OWL-S does not specify how pre and post conditions are stated, leaving the implementation language up to the developer. This presents two issues. The first issue is concerned with the open-standard environment web services typically operate in. In a domain where multiple standards may exist for a single cause (for example, multiple terms describing the same concept), unspecified framework approaches may lead to multiple approaches existing. For example, as the syntax used for pre and post conditions is not specified, service providers and service users may use differing syntax. A provider may specify conditions in a syntax which the user does not understand. Similarly, a user may search for a service using a condition syntax which is unsupported by available service providers. As an aside, Lin and Arpinar have investigated the use of RDF statements within the pre and post conditions [63], and found the approach wanting for this very reason. In this manner, discovery can also involve matching conditions of a service to a user's requirements.

The second issue is larger than OWL-S, being a characteristic of OWL itself. OWL is unable to describe *variable* concepts, that is concepts which accept their value from somewhere else. This can be illustrated using an OWL-S example (given in [20]). Suppose a precondition of a AirlineBooking service is that the CreditCard submitted as part of the input is valid. From a programming point of view, this can be expressed by creating a relationship between the CreditCard instance in the input and the CreditCard instance within the precondition. A skeleton excerpt for this is shown in Figure 6.4.

As can be seen, the CreditCard instance used to test the precondition is the same instance used to invoke the service. It is specified that the CreditCard instance used within the precondition holds the same value as that given in the input.

From an OWL perspective, it is not possible to denote this relationship, as abstract variables are not represented within the OWL syntax [53]. Ontology languages are designed to express facts about an environment or domain. There is no scope within OWL for specifying concepts which will be given their value or *facts* at a later date.

```
AirlineBooking {

   void invokeService(CreditCard inputCC, Journey desiredJourney) {

      if (validCard(inputCC)) {
         makeBooking(desiredJourney) ;
      }
      else {
       showInvalidCardError() ;
      }
   }

   boolean validCard(CreditCard testCC) {

      if(testCC.isValid()) {
        return true ;
      }
      else return false ;
   }
}
```

Figure 6.4: Specifying the relationship between class instances

Specific to this example, there is no way in OWL to describe the relationship between **inputCC** and **testCC**. The scope of **testCC** is only within the conditions part of the description. Similarly, the scope of **userCC** is only within the inputs and outputs part of the description. The problem of scope is discussed within the OWL-S specification, with plans to address it in future revisions.

6.3 Other Ontology-based Web Service Description Approaches

OWL-S is one example of the increasing interest in the use of ontologies within the Web Service domain. The expressiveness of ontology languages present a unique approach to providing a robust, multi-platform and multi-agent environment. The Semantic Web Service Framework (SWSF) and The Web Service Modeling Ontology (WSMO) are two further attempts at providing a standardised approach to service description, discovery, invocation and composition. This section will briefly discuss these approaches.

6.3.1 SWSF

The Semantic Web Service Framework has been designed to provide a more expressive environment for the describing, discovery and invocation of web services. SWSF is comprised of two main parts:

- The Semantic Web Services Language (SWSL). This is the language used within SWSF for describing all aspects of the service domain (although the language itself is not domain specific). This language also specifies rules which are used to infer and reason over SWSL concepts. (In comparison with OWL-S, OWL is the language of the OWL-S approach, in which the basic concepts are stated, and logical rules are specified)
- The Semantic Web Services Ontology (SWSO). This is the actual specification of the SWSF framework. The vocabulary is built upon SWSL, developing and customising terms to describe the many aspects of a web service. SWSO (which is sometimes referred to as First-Order Logic Ontology for Web Services or FLOWS) contains rules as to how aspects of a service should be expressed, such as service profile, service conditions, and service invocation instructions. (As a comparison to OWL-S, the SWSO would mirror the OWL-S specification)

In contrast to OWL-S, which is a Description Logic language, SWSL is a First-Order Logic (FOL) language. This allows SWSF to provide a more expressive environment in which to describe services. For example, a SWSF description can detail the state of a service during each stage of the service execution, while also providing information about service exceptions. Rather than rely on external systems to define logic-based pre and post conditions as in OWL-S, SWSF includes its own approach to describing service conditions. This allows the entire service description to be rooted within SWSO, providing a complete approach to Web Service descriptions.

SWSF is partially motivated by the efforts of OWL-S [21]. Similar concepts to those described within an OWL-S Profile can be found within a SWSF service description. SWSF also adopts a similar approach to grounding inputs and outputs of a service to a WSDL service description.

In summary, SWSF aims to provide a more rich and expressive environment over those offered by OWL-S and other description languages. Having a FOL approach allows other logic-driven web service languages to be substituted into the SWSF environment [21], increasing the interoperability of the framework. One main drawback with SWSF, compared to OWL-S, is that it is a fairly heavyweight based approach, involving complex ontology specifications and rules. This, however, is unavoidable if an expressive environment is required.

6.3.2 WSMO

The Web Service Modeling Ontology is an effort towards providing greater support for web service usage and composition through detailed descriptions of a Web Service. WSMO is built upon the Web Service Modeling Language (WSML) which utilises a similar logic approach as offered by SWSL (see section 6.3.1).

WSMO distinguishes between a Web Service and a Service: A Web Service offers one or more Services. For example, Amazon.com would be seen as a Web Service, which offers Services such as ISBN lookup, and online purchasing [1]. WSMO uses this distinction to highlight the exchange of messages between services when a user invokes a service. One service may use another service embedded within the same web service, or it may make use of external web services. WSMO describes service usage information as the *Choreography* of the service, and service composition as the *Orchestration* of services. Description of a service Choreography and Orchestration is built upon that of Abstract State Machines. In this manner, service interaction can be described as a series of Event-Condition-Action (ECA) statements. WSMO allows multiple interfaces to be defined for a single Web Service [1]. Each interface will have a series of ECAs attached, allowing the complete trace of data passing, possible exceptions, and expected outcomes for each interface.

WSMO also includes features common to both SWSF and OWL-S, namely the description of a web service through inputs, outputs, pre and post conditions. WSMO also describes the *assumptions* made by a Web Service, and the *effects* of a Web Service. These attributes refer to the state of the *world* before and after service usage. The definition of *world* is not limited to the web service domain.

WSMO is concerned with *Goals* rather than service categories. A Web Service can allow the full or partial achievement of a Goal. In simple terms, a user will search for a service which fulfills the user's goal, rather than search for a specific category of service. Service composition, or *orchestration* can be used to satisfy goals which cannot be fulfilled by a single Web Service. WSMO makes use of *mediators* to support the usage of multiple services.

Mediators are used to interpret and provide bridges between differences in terminologies, service invocations and goal descriptions. Mediators can be used to interact with a number of services to achieve a main goal. Mediators can also be used to reconcile terminologies from external languages to terminologies described through WSML. For example, OWL terms can be interpreted into WSML terms. The WSMO framework does not include implementations of any particular mediators, but provides references to those used within the service descriptions. In this manner, when a client uses a desired service, it can know before hand what mediators are provided and used by the providing service.

In summary, WSMO provides a framework which is capable of providing structured descriptions of Web Services. This ECA approach to service invocation allows WSMO to inherit the advantages of abstract state machines, namely having a formally defined process on which users can rely. The move from service categories to user goals provides a more natural approach to using web services, from a human user point of view. Service categories can be ignored, so long as a service satisfies a certain goal. This, however, can lead to issues such as deriving low level goals, from a high level goal description.

6.4 Web Services Conclusion

Web services have emerged as a catalyst for the development and research into new ontologies and ontology languages. The web-based approaches to providing an environment for adhoc service usage would seem appropriate for transference into the home network domain. Having a common language for description, along with common protocols for communication, allow the web service domain to provide distributed computing without the need for heavy configuration on the behalf of the user.

The approaches in web services also allow for an error-tolerant service environment. For example, service A is a regular client of service B. Due to node failure, service B becomes unavailable. Service A does not need to wait for the service to recover, and can instead search the network for a replacement service using the common description languages. Service C can then be substituted for service B, without any changes to the way service A operates.

For these reasons, ontology languages and web-service based approaches have been applied to home network domains in a variety of projects.

Chapter 7

Ontology Related Work

Several network related research projects have been motivated by the emergence of ontology languages and their use within the web service domain. These projects are not limited to service descriptions within a single home network, but also involve peer-to-peer networks [31, 77, 70], context-aware systems [80, 17, 81] and policy control [41, 32]. This chapter will discuss relevant ontology-based projects which are either based within the home network, or use approaches which could be applied to an aspect of the home network. This chapter will also review relevant work concerned with using ontology languages to create generic vocabularies, which may be used within the home domain.

7.1 Home Network Ontology Projects

As discussed in section 2.2, the vision of the home network requires devices and services to act within an ad hoc environment [51]. In order to reduce the invasiveness (with respect to the user) of configuration and operation, network components require to discover and use each other in a robust manner. If a user wishes to complete a task, the network must attempt to carry out this task to the best of its ability. Open standard communication protocols and middleware frameworks support the interaction of network components. Just as in web services, ontology languages have been applied to the home network to provide a richly described environment for discovering desirable network components.

7.1.1 The Networked Appliance Service Utilisation Framework

The Networked Appliance Service Utilisation Framework (NASUF) is a middleware framework designed to support service discovery within networked environments, such as the home network [30, 70]. It is an extensive framework which supports service discovery and composition, while mediating between service descriptions of differing vocabularies. Services are described through their inputs, outputs, preconditions and effects (IOPEs). NASUF has a component responsible for managing service requests, which are submitted in an IOPE format. NASUSF uses ontology languages to describe services and service requests. In this manner, inference and reasoning can be performed over descriptions, allowing the framework to resolve compatibility issues. For example, [30] describes a scenario where a service user is looking for a service which accepts a Movie as input. In the network, a service exists which accepts a Film as input. The NASUF is able to resolve this syntactical difference, allowing the two services to interoperate.

NASUF is a unique approach to home network management. Rather than have a centralised service registry, NASUF is able to propagate service requests over P2P networks [31]. This approach allows NASUF to offer the advantages of a decentralised service discovery approach, namely an ability to function in the presence of component failure. As nodes within the home network can maintain their own service registry, the failure of one registry does not prevent other components from functioning. Nodes also maintain a list of composite services which they can offer to the network. Composite services may use other services external to the node. Nodes offer composite services to the network as if they hosted the entire service themselves.

7.1.2 The Gadgetware Architectural Style Ontology

The Gadgetware Architectural Style (GAS) Ontology [18, 19, 17] is an approach to service and device descriptions. This approach describes *artifacts* and *plugs*, rather than devices and services. This description is grounded within the GAS Ontology, which not only capture attributes of artifacts and plugs, but also defines how users can interact with these artifacts. The GAS Ontology is broken into two main ontologies, the GAS Core Ontology (GAS-CO) and the GAS Higher Ontology (GAS-HO).

The GAS-CO provides the core concepts, relationships and attributes relevant to the home network environment. These concepts are common to all artifacts within the domain, as well as providing some core service classification. A service is represented through an *SPlug*, and an artifact may contain zero or more SPlugs. A *Synapse* describes the relation between two SPlugs, it represents the current relationship between a service provider and a service user.

The GAS-HO describes instances of artifacts, containing attributes of the artifact, state information and its plugs and current synapse. The GAS-HO also contains knowledge that the artifact has gained, such as what applications it is involved with and recently used artifacts. The GAS-HO part of an artifact description can be dynamic, changeable by usage or environmental changes.

The GAS Ontology is a unique approach to supporting the home network. Describing current transactions (synapse) is a novel idea, providing real time information as to the current state of providing and consumer services. However, it is difficult to visualise how the GAS approach can be integrated into the current state of the home network. This approach relies on devices being compatible with the *eGadget* and *eWorld* views of the UbiCom world.

7.1.3 Other Home Network Projects

The initial purpose of using ontologies within web services was to allow a more automated, simplified and richly described environment for service usage, composition and communication. This idea has been applied to middleware frameworks, allowing home networks to benefit from similar gains.

The AIDAS middleware, [88], is an approach towards service discovery with dynamic environments, such as the Home Network. This approach relies on semantic matching between user requests and service abilities. Services are described in terms of identification, capabilities and requirements. The these service profiles are captured using OWL, and are stored within a Discovery Manager component within the mainframe. User profiles are also represented in terms of identification, capabilities and requirements. On joining the AIDAS framework, the Discovery Manager semantically matches services which conform to the capability and requirement restrictions of the user. This allows the AIDAS middleware to appeal to a wide range of 'users' which may only speak one language, or have limited computational and storage resources.

OSGi is used in a number of home network projects to support cross platform communication, and information management. Some research projects have enhanced the OSGi platform with ontology-based approaches. de Vergara et al. [26] propose an approach which attempts to automate the service discovery process within the OSGi framework. Services and OSGi bundles are described through OWL, and rules are specified which allows the automated updating of a service registry. The user's view of the registry is customised, depending on the user's profile, and availability of the service.

The SOCAM architecture, [48], is a project while takes a more natural approach to service discovery. The architecture is built upon OSGi, and leverages OWL in describing context information in the home. The SOCAM architecture maintains a registry of available services which can be queried by other services. Clients within the SOCAM architecture query about the information they want, rather than for a particular service which provides it. For example, rather than find a service which provides location tracking, a service user would query over the location of a particular person. The registry then returns a service which can provide this information. The SOCAM approach is currently aimed at building context-aware systems, discussed in Section 7.2, rather than supporting home networks. However, the semantic reasoning behind the querying approach is certainly noteworthy, and would be a next step toward discovery over information rather than service categories and interfaces.

7.1.4 Service Discovery within Home Networks

In the projects discussed so far within this chapter, discovery within the network can be separated into two distinct approaches.

- User Centric: When a device controlled by a human joins the network, all available services are exposed.

- IOPE Centric: Following the same approach as web services, the main part of discovery is performed over the inputs and outputs of a service, based upon client requirements. Only relevant services are then exposed.

User Centric Approaches

User centric approaches, such as those by Toninelli [88] and de Vergara [26], typically assume that the user is best supported by having a complete list of available services displayed to them. This approach is in stark contrast to the typical approach in home networks (described in Section 4), where a registry is maintained, but available services are discovered by user requests and queries. Rather than take an active role in the service discovery process, user centric approaches are passive - the service registry is openly viewable. It is comparable to a service within the home network submitting a query to discover *all* services within the network. This query would then be continually submitted to ensure the returned list was up to date at all times.

A user centric approach is useful in networks where the human user wishes a great deal of control over their network. Having a complete list of usable services removes the need for any further querying. This is a powerful advantage of this approach. No interpretation between human-readable and machine-usable querying vocabulary is needed, and the user can have an accurate view of the whole network at all times (assuming the list automatically updated). Human users can also determine services of identical categories, despite potentially being described differently. This issue was previously discussed in Section 2.3.

A user centric approach can make service-to-service interoperability difficult. In the absence of a defined querying language or interface, services must maintain their own list of network services. This may take place as a new device or service joins the network, where a notification message may be broadcast (as with UPnP). In the absence of a structured central registry, network services must rely on other services being aware of their environment and continually broadcasting their existence. While this approach is well suited to providing a great deal of control to a human user, it may not be suitable for automated service usage and composition [88].

IOPE Centric

Performing service discovery over the inputs, outputs, preconditions and effects (IOPE) of a service is an approach adopted from web services (see Chapter 6). As web service-like descriptions are applied to the home network, [18, 19, 30, 70], the service discover methods follow a similar approach. For example, 'Find me a service which is of type X and accepts Y as input'. As discussed in Section 6.2.4, discovery using the pre and post conditions of a service is difficult and complex [97]. IOPE centric approaches therefore tend to concentrate on the input and output parts of a service description.

7.2 Context Aware Systems

Many forms of information can be labeled as *context* information. For example the current time of day, or outside temperature are examples of basic context information. User location and current activity are examples of more relevant and higher level context information. At an abstract level, context information can be any information which is not available from within the system. This information describes attributes external to the network.

High levels of context information can be gained from lower levels. For example, if John is currently in 'Meeting Room A', and the room has 3 occupants, it can be inferred that John is currently in a meeting. Notice that it is *inferred* that John is in a meeting, and not simply that John *is* in a meeting. Ontology languages can be used to derive high level context information from low level information.

For example, consider the ontology class described in Figure 7.1. An OccupiedRoom is specified as a room with 3 or more occupants.

Suppose, through various means, an employee's location can be derived by the office system. The real-time description of the office meeting room may be that shown in Figure 7.2. As the meeting room has 3 occupants, the reasoning system can infer that the room is occupied. John's location shows that he is in the meeting room. In turn the meeting room is occupied, and so we can infer that John is in a meeting.

In this manner, some research projects have concentrated on describing context information through ontology languages. This allows them to derive high level contexts by inference. Wang et al.[98, 99] discuss scenarios where the availability of a user is derived from context information such as current activity and location. This derivative is specified through rules concerning ontology information. The Gaia project [16, 80] moves beyond

```
<owl:Class rdf:about="#OccupiedRoom">
   <rdfs:subClassOf rdf:resource="#Room"/>
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#hasOccupant"/>
         <owl:onClass rdf:resource="#Person"/>
         <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
              3
         </owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

Figure 7.1: OWL representation of an OccupiedRoom

```
<owl:Class rdf:about="#MeetingRoom">
   ...
   <owl:Restriction>
      <owl:onProperty rdf:resource="#hasOccupant"/>
      <owl:someValuesFrom rdf:resource="#Paul"/>
   </owl:Restriction>
   ...
   <owl:Restriction>
      <owl:onProperty rdf:resource="#hasOccupant"/>
      <owl:someValuesFrom rdf:resource="#David"/>
   </owl:Restriction>
   ...
   <owl:Restriction>
      <owl:onProperty rdf:resource="#hasOccupant"/>
      <owl:someValuesFrom rdf:resource="#Simon"/>
   </owl:Restriction>
   ...
   <rdfs:subClassOf rdf:resource="#Room"/>
</owl:Class>
```

Figure 7.2: OWL representation of the MeetingRoom

simple rule-based inference, exploring the use of probability, fuzzy logic and Bayesian networks in order to derive context information.

The use of context information within a home network allows a more personalised system [28]. Allowing the user to customise the behaviour of the system based on information, such as location or current activity, would only help in integrating home networks into daily life. For example, a HouseLight monitoring system could manage room lights, switching off those in rooms where no users are present. This service, in turn, could then be a component of an EnergySaver service. In a similar manner, service invocation could be time-driven. A MorningRoutine service could switch on the heating at 7am, and then switch on the coffee machine at 8am, while displaying the latest news headlines on a display which is in the same location as the user.

A context-aware home network is certainly an attractive prospect. A typical home network may contain a number of simple devices and services from which context information can be gained, such as IR sensors, chair occupancy sensors, on-line calenders and television recording schedules [9]. From such raw data sources, information about availability, current activity and location could be derived.

Projects exist where developers have integrated context-aware applications into a home care environment, for example the Amigo project [39] and Construct. Construct [23] is a middleware platform which supports context gathering in home care networks, and represents the data through RDF. This augments the support available from the home care network, where service invocation can be heavily *trigger* based. (E.g. If the user is watching TV, and it is after 12pm, present the current time vocally every 10 minutes).

Representing context information through ontology languages allows the use of the information to be independent from how it is represented [59]. This prevents issues of *standard lock-in* where only specific applications can understand what the information *means*. Using ontology languages also allows the context information to be disseminated back into more basic data. For example, suppose John is currently engaged in a BoardMeeting activity. In addition to first inferring that John is unavailable, an agent can then check the definition of BoardMeeting, perhaps discovering who comprises as the Board, and then inferring that these people are also unavailable.

The success of ontology-based context aware systems depends on the ontologies being tightly controlled and well defined. The use of ontologies is more suited to small, local domains, than that of an all-encompassing standard. The reason for this restriction is that definitions of high level context information, such as activity, availability and preference can be closely related to the particular domain in which they are being used. One view of a person's unavailability may be different to that of another. Having an ontology particular to a small domain allows developers to ensure it is strongly suitable for that particular domain. Using a universal approach may require heavy customisation of the ontologies in order for them to be correct.

7.3 Upper Ontologies

An upper ontology is a an ontology which describes concepts which remain the same across multiple domains [37]. These concepts are general enough that their *meaning*, or what they represent, is not dependent on the context or domain in which they are used.

For example, it could be argued that a Person concept is general enough to be applicable to many domains without having to change the definition. A Person within the home domain would be the same Person within the work domain. This Person concept would remain unchanged if moving from a location-based domain to a more natural domain such as a legal or family domain. The Person concept would be relevant to all domains. (This does not mean all attributes of the Person concept are relevant to all domains, but that the definition of the Person concept is true in all domains).

Upper ontologies are useful for building large scale ontologies [37, 4]. The upper ontologies can be used as foundations for future development. The use of a third-party ontology as a foundation increases the chance of agents, who use one set of ontologies, being able to act in an environment described using another set [4]. When encountering an unknown concept or term, an agent may be able to reduce the concept into terms found within the third party ontology. If successful, the agent can then act on lower-level information, perhaps reconstructing the terms into concepts found within its own ontologies. (Other advantages of reusing ontology terms has been previously discussed in sections 5.5.2 and 5.5.3).

In simple terms, upper ontologies are a standardised set of ontologies designed to provide a common foundation for applications or further ontologies across a range of domains. This section will review the process involved in creating an upper ontology, and discuss existing upper ontologies used within the research domain.

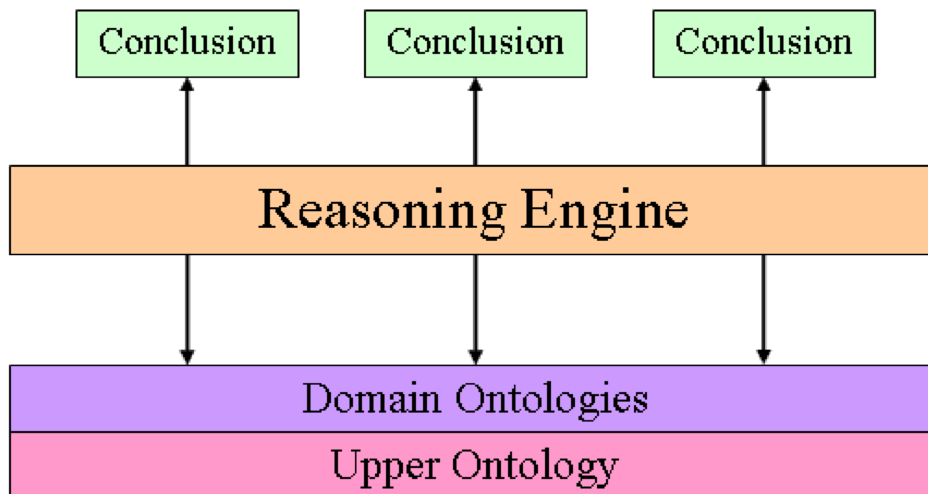


Figure 7.3: Customising an Upper Ontology for Use

7.3.1 Developing and Using Upper Ontologies

In creating an upper ontology, concepts, properties and rules are created and developed. These concepts may be simply stated entities, such as those in a taxonomy, or may have meaning defined through properties and rules. As already mentioned, upper ontologies may be wide ranging, but are rarely constrained to a particular domain. Upper ontologies may consist of a number of domain specific ontologies in order to be as inclusive as possible.

Developers can create their own ontologies using terms defined within an upper ontology [27]. At this stage, developers may tailor their ontology to a specific domain without invalidating the purpose of an upper ontology. This is because ontologies built upon an upper ontology will share a common definition of terms. A reasoner application, which can understand the ontology data and infer new information, can utilise the new domain specific information. The reasoner application can then draw conclusions based upon both the protocol specific ontology and the upper ontology data. Conclusions may be the result of a user query, or the adding of new information.

A number of upper ontologies have emerged within the research domain. Some have been developed for a specific purpose, while others are concerned with providing a consistent and structured approach to describing elements in multiple domains.

7.3.2 Cyc and OpenCyc

The Cyc project began in 1984, with the aim of developing an extensive set of ontologies capable of supporting artificial life [55, 84]. One main project aim is to support reasoning and recognition over natural language data. Using an example given by [55], the following sentences can cause problems for artificial intelligence agents:

- Fred saw the plane flying over Zurich.
- Fred saw the mountains flying over Zurich.

The Cyc project expresses information through CycL, a first-order logic language designed specifically for the project. Using CycL, facts about objects can be expressed, which can then be used to solve syntactic ambiguities like that shown above. Planes, of course, can fly, while mountains cannot. These facts can be *known* by a Cyc agent, and in turn the meaning of the sentences can be derived. In the first case, Fred is looking up at a plane. In the second case, Fred must be in a plane looking down over the mountains (while on his way to Zurich!). Understanding the sentence allows new information to be gained.

At present, the Cyc knowledge base contains 300,000 concepts, and nearly 3 million assertions. The Cyc project provides a reduced set of concepts through OpenCyc, which is an open-source version of the Cyc knowledge base. There exists an open-source version of the Cyc ontology, named OpenCyc. OpenCyc has also been translated from CycL into OWL, and contains a set of permanent end-points representing various concepts within the ontology [56]. Despite being an reduced version of the full Cyc knowledge base, OpenCyc still contains a substantial amount of information [57].

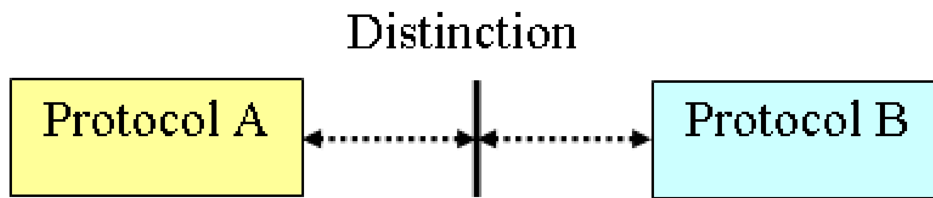


Figure 7.4: Boundaries Between Protocols

7.3.3 The Suggested Upper Merged Ontology (SUMO)

The Suggested Upper Merged Ontology is another movement towards creating an upper ontology to support multiple domains and applications [73]. It is written in a first-order logic language called SUO-KIF and is controlled by the IEEE, although SUMO is in fact open source. SUMO is a collection of ontologies from various general and specific domains, such as Economy, Geography and Government. It can also supply information from external sources, such as Wikipedia [79]. It boasts 20,000 terms (concepts) and 70,000 axonims (relationships). A translation of SUMO from SUO-KIF into OWL is also available [79]. SUMO, like Cyc, is concerned with expressing information in a structured way, so as to allow artificial intelligence to act on the facts, and inferences present within the knowledge base. SUMO is also well grounded within the WordNet lexicon, which is a large database of English terms, relationships and meanings designed to support artificial understanding of words.

7.3.4 DOLCE

DOLCE stands for a Descriptive Ontology for Linguistic and Cognitive Engineering. It is heavily engineered towards assisting machines in understanding the human language [75]. In this manner, as machines *understand* the language, they are able to identify subjects (such as a Person), actions and able to predict *consequences* of existing facts and statements. The DOLCE project is also involve in integrating WordNet into the DOLCE ontology (through the OntoWordNet project [75]). Rather than developing ontologies for a specific domain, DOLCE its aimed at defining what words *mean* within the general human language. These terms can then be carried into specific domains, as developers can be confident that the terms are both correct, and well defined.

7.4 Comment on Existing Approaches

As has been shown throughout this Chapter and Chapter 6, a large amount of effort and progress has been made toward providing seamless interoperation between networked components. Communication protocols, such as SOAP and WSDL, are being applied in web service environments (discussed in sections 6.2 and 6.3.1) to assist in the communication between heterogeneous components. UPnP follows a similar approach toward home network communication (see section 3.2). These protocols abstract the data being passed from the platforms or protocols of the components conversing. This abstraction *blurs* the distinction between protocols at the communication layer. In other words, providers and consumers see no differences between each other while communicating. A provider may be operating on protocol A, while a consumer may be operating on protocol B. Figure 7.4 represents this situation. There is a logical distinction between these protocols because of the communication protocols they each use.

If both parties use an abstracted communication protocol, then the provider has no need to know that the consumer is operating on a completely different protocol. Figure 7.5 shows two components sharing a communication protocol. The left hand component has no reason to believe the right hand component is executing on a different protocol than itself. At the communication level, both components are seen to be sharing the same protocol.

Description approaches such as UDDI, OWL-S and SWSF allow services to describe themselves in an manner independent of their operating protocol. As has been discussed in Section 7.1, projects exist which aim to achieve the same abstraction. The view that a web-service type approach can be applied to the home network to achieve the same aims may be a little naive. As thoroughly discussed in Chapter 3, several mature protocols already exist within the home network domain. These protocols already contain their own approaches for description and communication. In order to conform with an approach such as NASUF or GAS, entire protocols would require to

Figure 7.5: Common Communication Protocol

be re-engineered. Only once a protocol conforms to a description approach, can it be said to abstract from its own executional protocol.

This issue is partially solved by middleware (see section 3.8). Middleware can facilitate the exchange of protocol specific messages between network components. As already discussed, this can performed through *drivers*. Drivers are responsible for translating protocol specific messages. These messages may be translated into a format abstracted from any protocol, like that of web services. These messages may also be translated directly into another format designed for a specific protocol. Using middleware drivers removes the need for existing protocols to be altered to comply with a specific communication approach.

Figure 7.5 depicts the interaction between protocol drivers. Messages are translated into a common format shared by both drivers, which can then be reconstructed into a protocol specific format at the receiving end. Two important items must exist for this communication to take place:

- A driver, to do translation on behalf of a protocol domain
- A common message format, to allow information to be shared between protocol domains

It should also be mentioned that the common message format must be relevant to all forms of protocol specific messaging. What this means is that it is not enough to simply come up with a simple messaging format. The format must be able to encapsulate the kinds of data which would normally be passed between components sharing the same protocol. This is also true within the web service domain, where SOAP allows the exchange of multiple types of messages.

It becomes clear that to bring an ontology based approach toward describing network components into the home network, a similar approach is required. The aim of this work is to provide exactly this approach. A common description framework has been created which allows services and devices to be described in an protocol-independent manner. This framework captures relevant concepts in order to encapsulate the many types of approaches and vocabularies used by home network protocols. The framework is expansive enough to handle the description approaches described in section 4.1, while also having scope to retain protocol-relevant attributes.

At the same time, this work describes a system architecture designed to support the description framework within home network environments. The architecture allows existing and emerging protocols to participate in an ontology-driven discovery process, without the need to conform to the framework or vocabulary. This work also provides an implementation of this middleware framework, complete with protocol drivers. This framework is designed to provide a protocol-free way of discovering services and devices in the home, mimicking the web service approach to describing network components.

Chapter 8

Approach

This chapter will describe the approach taken toward specifying an ontology based description vocabulary. This chapter will discuss the purpose and scope of the vocabulary, followed by the methods taken to specify the terminology. An insight into the reasons for the methods used will also given.

The following Chapter will discuss a system architecture and implementation to support this approach within a home network environment. Chapter 9 demonstrates how this approach can support cross-protocol component description and discovery within existing environments, and be extended for future networks.

8.1 Purpose and Scope of the Approach

This work is unique in its aims and approach. Rather than simply creating a new, independent description framework, this work is designed to leverage and incorporate existing description approaches and vocabularies. This work is designed to be compatible with existing protocols, working in tandem with middleware frameworks to support service and device users. This work allows home network services and devices to be described independently from the description mechanisms imposed by their protocols. In addition, this work is able to be applied in existing home network environments, as well as being extensive and scalable to adapt to future developments. The aim of this work is to allow clients to discover desired components without the need to understand protocol specific vocabularies and discovery mechanisms.

This approach is independent of any specific discovery approach, from those described in Sections 4.2 and 7.1.4. There is no intent in this approach to describe interaction details of network components. Middleware already provides support for interactions between multi-protocol domains. Without the need to specify *how* services should interact, this work can be applied over multiple domains quickly, removing the need to cross over between the protocol description and interoperation boundaries.

This approach allows a generic vocabulary for describing home network protocols, devices and services. Having a common vocabulary provides a focal point, a single set of terms for discovering desired components. Currently, clients using middleware require to know the protocol specific terminology to discover suitable components. The discovery process for each protocol within the network may differ, and so a new discovery process may need to be constructed for each home networking protocol. This work simplifies this process, removing the requirement from the client to know about every possible home networking protocol. Clients need only construct a single discovery process, using the terms from the generic vocabulary, to discover available network components, regardless of protocol. The second part of this chapter will discuss how the implemented middleware framework federates this single process into a discovery program over all available.

This work shall demonstrate how the use of OWL allows for semantically rich component descriptions, which in turn provides a logical framework in which to perform discovery. These descriptions increase the scope of a standard discovery process, while reducing the complexity needed of a network client to participate. To this end, this chapter will highlight how, by using ontologies and middleware, a rich description framework can be applied to the home network environment, without the need to restrict the protocols used.

Base Ontology

Figure 8.1: The Base Level of the Stack

```
<owl:Class rdf:ID="Location"/>
<owl:Class rdf:ID="Component"/>
<owl:Class rdf:ID="Service"/>
<owl:Class rdf:ID="Protocol"/>
<owl:Class rdf:ID="User"/>
<owl:Class rdf:ID="Software_Module"/>
<owl:Class rdf:ID="Security"/>
<owl:Class rdf:ID="Device"/>
<owl:Class rdf:ID="Context_Variable"/>
<owl:Class rdf:ID="Vendor"/>
```

Figure 8.2: The Base Ontology

8.2 Approach Method

Section 5.5.1 introduced the ability of an OWL ontology to import the knowledge of external ontologies. Sections 5.5.2 and 5.5.3 discuss ontology reuse within different domain sizes. Ontology importing and reuse provides a structured approach toward defining a vocabulary. The vocabulary described in this work is built in a stack-like manner where ontologies below define information used by those ontologies above. In this stack, upper levels *reuse* information provided by lower level sources. This allows the generic vocabulary to be grounded in classes and attributes described in low level ontologies.

This stack has been named the Home Network Ontology Stack (HNOS). It has been designed to be scalable and expansive, allowing the vocabulary to be expanded in the future as the home network domain expands, as new protocols emerge and as new services and devices are developed. The ontology stack contains four main ontology layers:

- The Base Layer, described in Section 8.2.1
- The Core Layer, described in Section 8.2.2
- The Generic Layer, described in Section 8.2.3
- The Protocol Layer, described in Section 8.2.4

The base ontology layer contains a single ontology. The other levels of the stack are comprised of a collection of ontologies. Each level of the stack has a distinct purpose in supporting the description framework.

8.2.1 The Base Layer

The base level of the stack (shown in Figure 8.1) provides a foundation for the vocabulary. This ontology is responsible for stating relevant concepts in the home network domain. These concepts are given no meaning or definition, and are simply stated in order to exist. This level of the ontology stack is not unlike the upper ontologies described in section 7.3. The purpose of the base layer is to contain a collection of terms which can be reused by ontologies tailored to a specific domain. No assumptions are made as to how the concepts will be used by other ontologies, even though the ontology is designed for the home network domain.

Figure 8.2 displays the ontology information within the base layer. Each concept has some relationship to the home network domain. Each concept also has a *semantic* intention or meaning behind it, but is not expressed in this layer. By itself, this ontology simply contains string values, each concept differing from the next simply by the arrangement of characters.

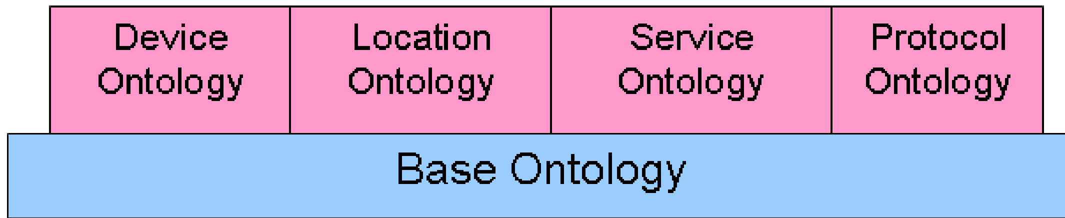


Figure 8.3: The Core Level of the Stack

Relating Base Concepts

Ontology based home network projects, such as those described in section 7.1.4, typically define a set of ontologies to support their approach. These documents exist independently of other ontologies which exist for similar purposes. Despite the use of ontologies, search agents can still be restricted by the issue of semantically similar, but syntactically different description vocabularies (introduced in Section 1.1). Advantages can be gained by developing ontologies in acknowledgment of existing vocabularies. This work attempts to acknowledge the work done by other ontology developers, specifically those who have developed upper ontologies.

Having a set of core concepts which are simple and undefined, allow relations to be formed with other upper ontology concepts. By using the **owl:equivalentTo** property, owl concepts from one ontology can be declared identical to those in others (as described in Section 5.4.1). By this method, the core layer of the ontology can be unified with other upper ontologies.

Using Base Concepts

As already mentioned, the base layer is designed to provide a simple, small set of terms to use as an anchor into the domain vocabulary. Without any kind of definition, these concepts offer little advantage over simple RDF data, or a keyword approach. Each concept represents an important domain within the home network environment, but in themselves do not provide a satisfactory vocabulary to start describing home network components. The second layer of the stack is charged with expanding the vocabulary for this purpose.

The base concepts are intended to represent the most abstract level of semantic classes within the home network domain. These classes have been captured through observations of existing description approaches. These approaches include those described in home network protocols, such as UPnP, HAVi and Jini, and the web service description languages, such as OWL-S and UDDI. A critique of this approach is given in Chapter 12.

8.2.2 The Core Layer

The core level of the stack (Figure 8.3) contains the main descriptive elements of the framework. In this layer, domain vocabularies concerning the base concepts stated earlier are giving *meaning*. In the scope of this work, to give a concept meaning is to provide descriptive attributes which can logically distinguish the concept from others. The core layer contains a number of ontologies, one for each of the concepts contained within the base layer. In this manner, each ontology can be developed independently of others within the layer. This characteristic is important for the vocabulary to be extensive and robust, and shall be highlighted later in this section.

Each ontology within this layer has been developed to provide descriptive attributes and new classes relative to the domain of the base concept. For example, the Device ontology contains descriptive attributes which can be applied to the device concept, such as **offersService**, **hasLocation** and **hasDeviceCategory**. The Device ontology also contains a parent concept, **DeviceCategory**. A DeviceCategory represents the purpose of the device. For example, **Speaker, hasDeviceCategory, Audio** denotes that a Speaker concept can be classified as a device is categorised as an Audio device. The Device ontology therefore defines many classes of **DeviceCategory** which can be used to represent the classification of a device.

More Than A Taxonomy

A taxonomy is a classification of particular domain. Within a taxonomy, classes share relationships between each other, such as parent-child relationships. For example, a taxonomy can contain the fact that a Car is a subtype of Vehicle. At first, it would seem that the core layer of the stack is merely a taxonomy for the particular domain,

but this is not so. As stated earlier, the core layer can be used to give meaning to classes. For example, when a member of the Speaker class is encountered, it contains attributes which denote it as being a Speaker. There is meta-data available to denote *why* it is a member of the Speaker class, rather than it simply just *being* a member of the class. If a device is a member of the Speaker class, this means it also **hasDeviceCategory: Audio**

Having Separate Ontologies

One of the main assumptions of this work is that there can be no complete ontology or set of ontologies which accurately capture every relevant aspect of the home network domain. To this end, taking an isolationist view toward ontology development protects the stack approach against situations where disagreements arise to the contents of a domain ontology. Section 5.5.3 discusses issues associated with developing large ontologies to cover a number of domains. Separating ontologies into specific domains provides a structured approach to ontology development. If a developer or user disagrees with assumptions or definitions made within one ontology, there is no need to disregard all other ontologies within the core layer.

Suppose a developer already has an ontology which describes locations within the home. Rather than program their search agent to work with a new definitions of location related concepts, they can simply substitute the location ontology within the stack with their own. When the developer now begins to create component descriptions, they can use a mix of terminology from both the stack inbuilt vocabulary and their own location ontology.

As all information concerned within the core ontologies is either self contained, or references concepts within the base ontology, there is no conflict or invalidation of the other ontologies within the core level. The developer merely requires to provide a relationship between their own base location concept and that within the base ontology. The logical properties of OWL allow for search mechanisms to bridge this combination of ontologies.

For example, suppose the developer's base location concept has the URI **ext:Room**, this can be declared equivalent to the Location concept within the Base ontology through the owl property **(ext:Room, owl:equivalentTo, lsd:Location)**. The Device ontology contains the property **hasLocation**, which is defined in the ontology by **(hasLocation, rdfs:range, lsd:Location)**. Through the equivalence property, **hasLocation** has an implied definition **(hasLocation, rdfs:range, ext:Room)** which can be inferred through parsing the OWL description. Suppose **ext:Room** contains a subclass **ext:LivingRoom**. Through the replacement of the Location ontology, but the use of the other ontologies in the core layer, the following description would be valid: **(Speaker, hasLocation, ext:LivingRoom)**.

This example highlights the advantages gained by separating domain ontologies. This approach increases the usability of the stack vocabulary, reducing the chance of a developer requiring to create their own home network ontology. This layer of the stack is not intended to be complete. New classes of devices and services may emerge, along with new representations of the other base concepts. For example, it is unfeasible to believe every class of Context_Variable can be captured within a single ontology, as this concept can be largely thought of as context dependent.

Abstracting from the Ontology

This core level is like a collection of dictionaries. Each dictionary is concerned with one term from the base level. A dictionary defines how the concept can be used. Concepts are given definition from the attributes stated in the dictionary. The reason the layers within the ontology stack are separated is this: Someone may disagree with one of the dictionaries, but this does not exclude them from using the other dictionaries. If everything was in a single ontology, disagreeing with one dictionary would invalidate the whole ontology. In this approach, they can define their own dictionary for a concept, but reuse the rest of the dictionaries within the layer.

8.2.3 The Generic Layer

The Generic Layer of the stack (Figure 8.4) is where general service and device descriptions are formed, using the ontologies from the lower layers. Ontologies in this layer are concerned with providing generic descriptions of common devices and service found within the home network. For example, Figure 8.5 shows the description of a generic audio speaker. This is a description of a device, using attributes stated in the Device core ontology (**device;hasCategoryType** and **device;offersService**). It also references a class created within the Service core ontology, **(service;Audio_Output_Service)**.

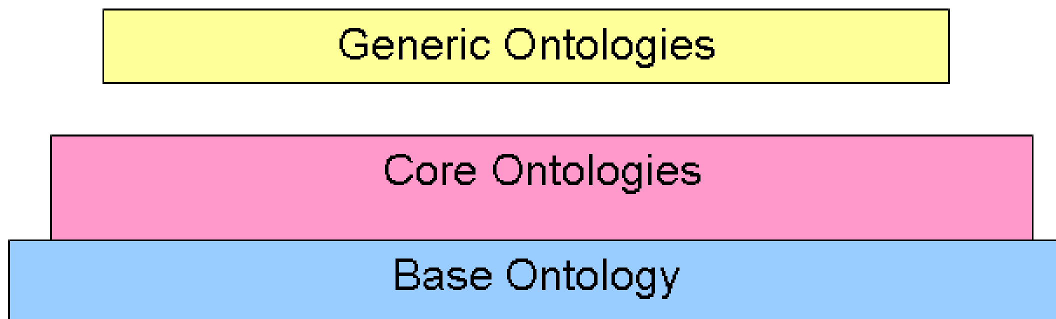


Figure 8.4: The Generic Layer of the Stack

```
<owl:Class rdf:ID="AudioSpeaker">
   <owl:equivalentClass>
      <owl:Class>
         <owl:intersectionOf rdf:parseType="Collection">
            <owl:Restriction>
               <owl:onProperty rdf:resource="&device;hasCategoryType"/>
               <owl:someValuesFrom rdf:resource="&device;Audio_Speaker"/>
            </owl:Restriction>
            <owl:Restriction>
               <owl:onProperty rdf:resource="&device;offersService"/>
               <owl:someValuesFrom rdf:resource="&service;Audio_Output_Service"/>
            </owl:Restriction>
         </owl:intersectionOf>
      </owl:Class>
   </owl:equivalentClass>
   <rdfs:subClassOf rdf:resource="&core;Device"/>
</owl:Class>
```

Figure 8.5: Description of an Audio Speaker

This description can be read as: A device (denoted by the rdfs:subClassOf property) which is categorised as an Audio_Speaker (denoted through the owl:restriction on **device;hasCategoryType**) and offers an Audio_Output_Service (denoted by the owl:restriction on **device;offersService**). This device has the id tag **AudioSpeaker**.

A parser can investigate the explicit definition of the service by visiting the service endpoint, denoted by the service URI. RDF/XML allows for namespace use (see Section 5.2.4). Substituting 'service' for its namespace will provide the URL.

The Generic layer of the stack is also not intended to be a complete work. As new services and devices are created for the home network, so new descriptions will be created. These descriptions can be added to the generic layer at a later date.

The ID Tag Illusion

Defining generic descriptions can be a difficult task. What one developer defines as being a lamp, another may class as a table light, or a standing lamp. Difficulties can arise because of the starting point taken when describing a device or service. When describing a television, for example, it is important not to apply attributes to the description just because it has been given the literal identification 'television'. The ID tag is simply another attribute of the class. It is designed to identify the class, *not to indicate what the properties of the class are*. Logically this makes sense: a literal value can not initiate any kind of inference. It is the class as a whole which is reasoned over, which can initiate inference, and which represents an actual real life *object*, accurately or poorly. Therefore it is not possible for a class description to be *wrong* based on its ID tag.

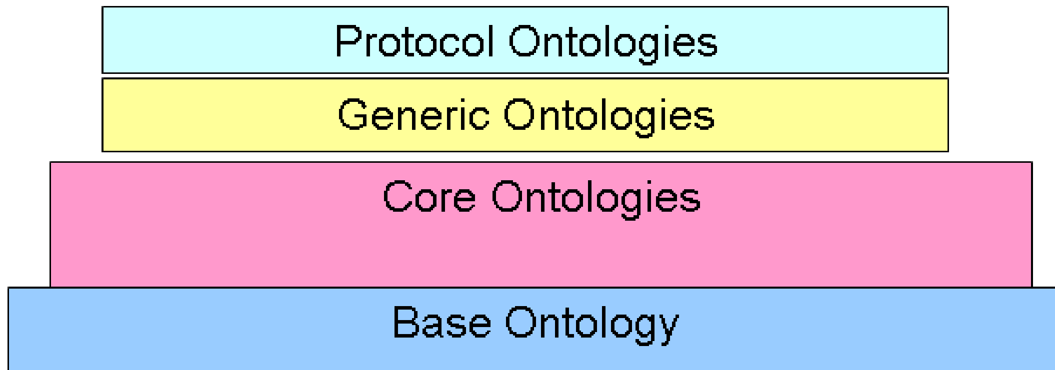


Figure 8.6: The Protocol Layer of the Stack

When performing any kind of discovery, services or descriptions can only be requested by ID tag once the requester is sure that the class description identified by the tag accurately represents the desired component. Without this assurance, discovery agents must discover components by an abstract criteria. For example *Find a device which offers service X. This service must be of category Y. The device must be located in Z.*

8.2.4 The Protocol Layer

The Protocol layer of the stack (Figure 8.6) is concerned with the various protocols which may be found within the home network. The responsibilities of this layer are two-fold. Firstly, this layer allows protocol descriptions which are semantically related to those in the generic layer to be declared so. Secondly, this layer allows the description framework to capture and retain property-specific attributes about devices and services, and allow them to be stored within the description.

Protocol Descriptions

The approach of this work is based upon the assumption that the middleware framework managing the home network has the capability to perform cross-protocol discovery. As Chapter 3 has shown, many home network solutions already contain their own mechanisms for discovery, which can be utilised by middleware frameworks. Other protocols, such as X.10, require more explicit interaction with the home user, but are still largely able to be included in a home network. With this assumption in mind, the protocol layer of the stack is built upon the presumption that the middleware can extract descriptions from the environment. Protocol ontologies are responsible for capturing classes of descriptions, services and attributes which can found in their own domain.

For example, the UPnP ontology contains classifications of UPnP devices and services, while also capturing the specific attributes defined in the UPnP schema. In this manner, as attributes are parsed by the middleware framework, they can be transfered into their ontology equivalent. At this point, a protocol specific description can be represented in a protocol specific ontology description. If left at this point, the description would remain understandable only to those agents who understood the particular protocol ontology. As stated, the aim of this work is to provide a single querying language for discovering network components.

To alleviate this potential issue, a protocol ontology also includes relationships between protocol specific classes and those within the generic and core layers of the ontology. For example, a UPnP_Device is declared to be a subclass of Device, the concept stated within the base ontology. At this point, the usefulness of this work becomes clear.

Suppose an agent simply wanted to discover all devices within the home network. The agent is executing within a middleware framework, and has access to the ontology framework. The agent can simply request a list of all instances of the Device class. At this point, assuming the UPnP ontology is being used by the UPnP domain, all UPnP devices will be returned. For each protocol ontology (and complimenting description translator) that exists within the network, the query will also return available devices in the instance list. The initial query makes no assumptions about what protocols the agent expects. A single reference is used, without the need to specify the inclusion of any specific protocols. This query covers all matching ontology classes, as they all share a common root within the Device concept. Figure 8.7 illustrates this query, denoting the relationships with the ontology

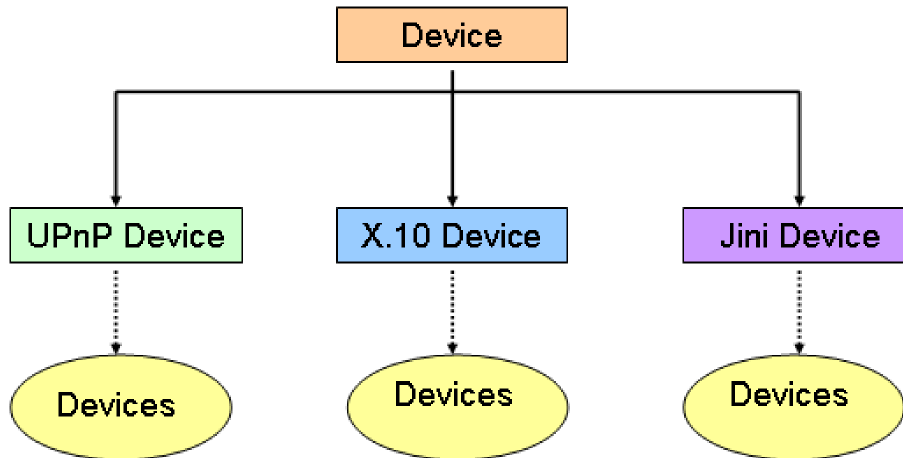


Figure 8.7: Architecture of Protocol Devices

descriptions. If all UPnP devices are denoted as being a subclass of an UPnP_Device, then logically they are all instances of a Device.

This example highlights the potential simplification of discovery when grounding protocol descriptions into abstract concepts. Protocol ontologies also relate devices and services to those described within the core layer of the stack. For example, an UPnP_Speaker is a subclass of the Speaker class created within the Device ontology. A UPnP_Audio_Input_Service is a subclass of the Audio_Input_Service described within the Service ontology. In this manner, if an agent wishes to discover an Audio_Input_Service, it will discover the UPnP service.

Figure 8.8 illustrates a similar example. Suppose an agent wished to discover available lamps within the home. On searching for instances of the class Lamp, created within the Device ontology, the agent discovers two specific instances:

- urn:schemas-upnp-org:device:light:1 - a UPnP instance of a Lamp Device.
- uk.ac.cs.lsd.LightInterface - a Jini implementation of a Lamp Device.

These instances can be considered by the agent to be generic instances of a Lamp. The only restrictions on what protocols can be discovered using this generic vocabulary depends on the protocol ontologies available (and their corresponding translators). If an agent wishes to discover devices of a specific protocol, they simply need to specific the classification of the device or service using the protocol specific ontology. For example, suppose an agent could only interact with Jini driven devices. Instead of requesting instances of Lamp, and then querying the protocol of each returned instance, the agent would request instances of the Jini_Lamp device. This request would return only Jini devices, ensuring the returned matches are usable.

Protocol Specific Attributes

It is not entirely impossible for a protocol to specify attributes which can find no root within the ontology stack vocabulary. The Service and Device ontologies attempt to provide a set of attributes which are common to home network protocols, without attempting to include every possibility. Rather than discard this information, protocol ontologies are encouraged to include specific attributes. This may at first seem inefficient, as these attributes will have no generic roots, and are undiscoverable using generic queries.

The reason for retaining them is this: once a search agent has discovered a suitable component, it is possible to determine what protocol the component is using. Principally, this information would be required to interact with the component. Suppose the protocol specific description approach contained contact information within its description framework. Without the specific attributes within the ontology description, an agent, once locating the component, would have to re-query the component using the protocol specific format. This scenario is in contrast with the aims of this work. To this end, having protocol specific attributes within a protocol ontology does not invalidate the approach, but provides robust support for the interaction part of a home network transaction.

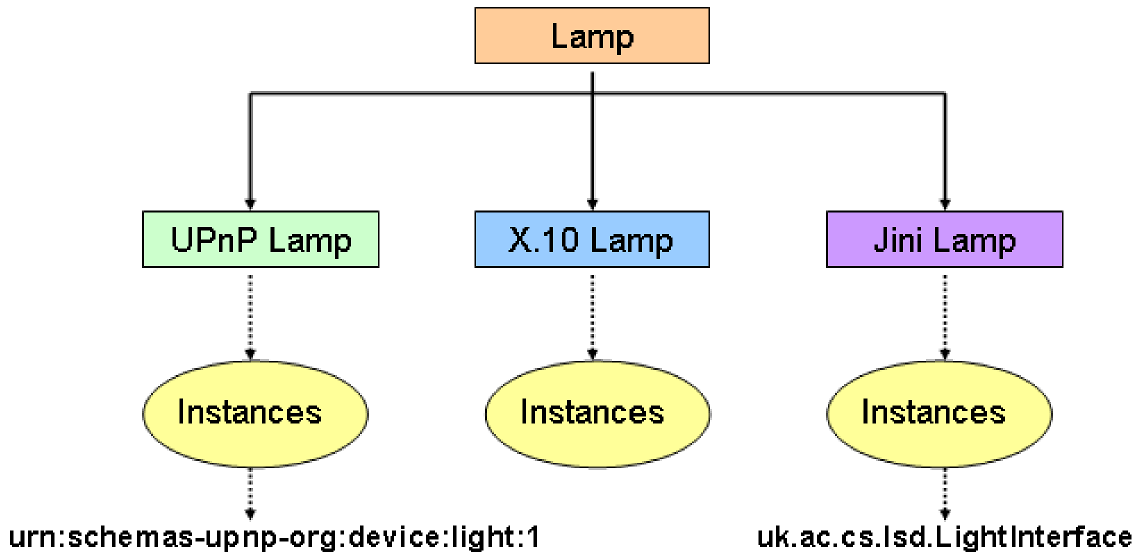


Figure 8.8: Architecture of Lamp Devices

8.2.5 Converting Between Protocol and Ontology Descriptions

The conversion between the protocol specific terminology and the ontology terminology varies between protocols, and is very much protocol specific. This work concentrates on three main protocols, UPnP, Jini and X.10. A set of ontologies have been designed and developed to represent descriptions of each of these protocols.

UPnP has a defined schema for describing services and devices. To this end, it is relatively easy to extract the description and convert it into an ontology representation. Jini does not provide a specification for describing services, instead making use of Entry objects to denote a particular attribute. A small set of Entry objects are defined in the Jini specification, and are used in this work to extract a Jini description for conversion. X.10 has no method for providing any kind of description, and so requires a user to set up a description through a user interface. This is not a limitation of this work, but a limitation of the protocol. To include the X.10 protocol in a home network, explicit user intervention will be required in all aspects of integration and description. Leveraging this requirement, an X.10 ontology description is generated as the user creates a description for the home network. In summary, ontology descriptions for each protocol are based upon:

- UPnP: The UPnP schema.
- Jini: Entry objects included in the Jini API.
- X.10: User generated description.

The methods used for generating ontology descriptions of these protocols is described more thoroughly in Section 9.

Protocol Layer Conclusion

The Protocol layer of the stack allows protocol descriptions to be related to more generic descriptions, while retaining their protocol specific attributes. Protocol components can be described at a protocol level. For example, an UPnP Audio Speaker is described as an UPnP Audio Speaker. As previously described, the protocol specific classes are logically grounded in classes within the lower layers of the stack. In this manner, when a UPnP Speaker is identified, further ontology information can be inferred about this class. For example if it is an UPnP Audio Speaker, then it is a member of the Audio Speaker class. This then infers that it offers an Audio Output service. The UPnP description may also describe this service at an UPnP level (through an UPnP Audio Output description).

```
<owl:Class rdf:ID="ACME_Television">
   <rdfs:subClassOf rdf:resource="&core;Television_Device"/>
</owl:Class>
```

Figure 8.9: Using an Existing Television_Device Description

If a protocol-based description does not redefine the attribute values of the generic class, then the generic values are used. For example, if a UPnP Audio Speaker does not denote that it has an UPnP Speaker device category, then it inherits the generic Speaker device category from the generic Audio Speaker description.

Finally, as new protocols may emerge, this level is not intended to be complete. New ontologies can be added to this level of the stack as required, even at run-time. This feature of the stack allows the overall approach to be extensive and scalable. As ontologies are reasoned over continuously, new ontology information can be added at runtime without the need to restart the middleware framework. As will be shown, error and conflict detection is a continuous process as new information is added. In this manner, the size of the vocabulary can grow as the network grows, without abandoning the principles of the stack-based approach.

8.2.6 Developer Defined Ontologies

Using the stack vocabulary, users and developers can create their own descriptions, knowing that their descriptions will be well grounded. As has been stated, the Home Network Ontology Stack is not intended to be a static vocabulary, indeed this work does not claim it to be a complete vocabulary. In an environment where new technologies may emerge, it is important that the stack is never assumed to be a closed, finalised work. Users can use the vocabulary in two distinct ways.

Using Existing Descriptions

Suppose a user has set up a home network, managed by a middleware framework. Suppose the middleware framework contains a software driver for a television device of a proprietary protocol. The user wishes to add this device to the description registry (which is built upon the ontology stack). The user can simply create an subclass of the Television_Device described within the Generic layer of the stack, shown in Figure 8.9.

This simply annotates that the ACME_Television is a member of the Television_Device class, and thusly inherits the attributes of the Television_Device. This description will also inherit the generic attributes of the services offered by a ACME_Television Device. As the middleware is responsible for the software driver, protocol specific implementation details are not required. Some annotation will be required to indicate that the middleware is the controller. This can simply be added with the **usesProtocol** attribute in the Device ontology.

A similar process is used to describe protocol specific devices and services. Protocol translators, described in Section 9, create customised classes based upon those within the protocol ontologies. These descriptions allow clients to first discover the component, and secondly determine the protocols used by the components. The protocol of the component provides further contextual meaning to the ontology information.

Adding Ontologies to the Stack

The stack is deliberately designed to be generic, in order to be suitable for a range of potential home network domains. Leveraging the support for multiple protocol domains offered by the ontology stack, users can customise layers of the stack to support their particular domain. For example, Chapter 12 describes how the ontology stack has been used in the MATCH project. Various ontologies have been added to the stack to support service and device discovery in a home care environment. For example, ontologies describing home care devices and services have been added to the Core and Generic layers of the stack, while ontologies describing aspects of the communication protocol used have been added to the Protocol layer.

Users can customise the stack to support their own domain, be it specific (such as home care) or wide-reaching (such as a home-network which extends into the web service domain). Figure 8.10 depicts this potential customisation, with the scope of user ontologies reaching as far as the core levels of the stack. As indicated in Section 4.2, the HNOS also provides the richness of a description schema approach (with defined attributes and class values). The HNOS also allows developers to reuse existing work, removing the heavy-handedness associated with creating schemas for each new device and service.

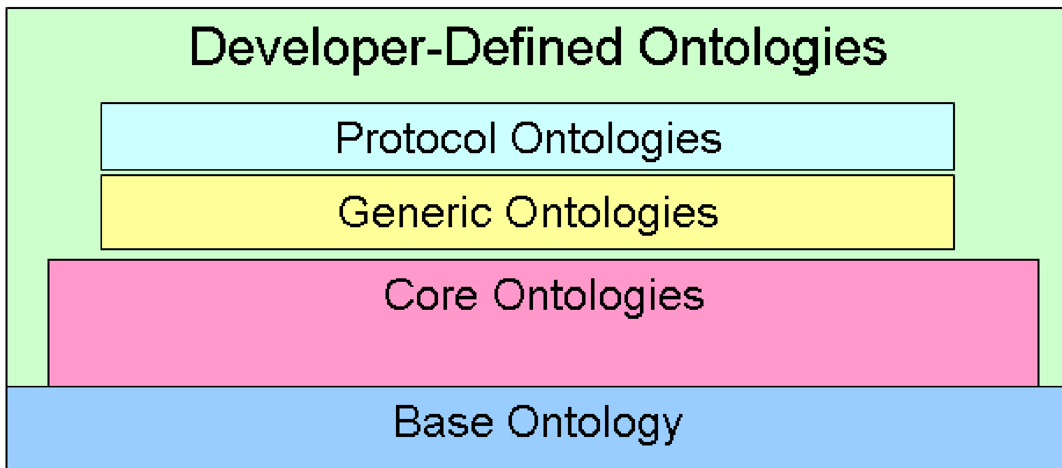


Figure 8.10: The Development Layer of the Stack

8.3 HNOS Conclusion

Sections 2.3 and 7.4 discuss the issues which require to be addressed in order to support a home network unhindered by protocol distinctions. Section 7.4 in particular highlighted two issues which continue to distinguish the home network environment from a web-service like environment. The need for a common abstracted vocabulary, and a method to interpret protocol specific description frameworks. Addressing these issues would remove the protocol distinction between network components at the description layer.

The HNOS addresses the issue of a common vocabulary which can be used to exchange service and device descriptions. The HNOS allows home network clients to use a single vocabulary and framework for discovery, removing the current requirement on clients to query using the discovery framework of each potential protocol. The HNOS approach also allows this work to be extensive and scalable. As new services, devices and protocols are added to the network, new ontologies can be added *at run time* without the need to restart the framework, or retrieve a new version of the HNOS. (Conceivably, the HNOS could be versioned as required, removing the need for new ontologies not included currently to be continually added).

The following Chapter describes how this work approaches the second issue which requires to be addressed: that of extracting protocol specific descriptions from the environment. The ontology vocabulary described within this Chapter requires to be supported from *within* home networks, rather than be applied externally by developers and vendors. Chapter 9 describes the second stage of this work: A system architecture designed to support the ontology approach at the middleware level. Chapter 9 will also describe an implementation of this architecture, using existing home network solutions and technology.

Chapter 9

System Architecture and Implementation

This work is concerned with supporting the home network through a common vocabulary for describing services and devices. One unique aspect of this work is the aim to include existing home network protocols within this approach. For this reason, no assumptions are made about the flexibility of existing protocols to adapt to a new description framework. To include existing protocols, this work proposes the use of middleware to support the translation of descriptions from protocol specific terms into general ontology based terms. Middleware also provides a central point of reference for discovering desired services and devices by maintaining a registry of ontology descriptions.

This work proposes a simple system architecture for support the ontology-based approach described in the previous chapter. This architecture contains two main component types for applying the approach to home networks:

- An Ontology Registry: This component is responsible for managing the ontology descriptions of all services and devices within the network. The registry owns an ontology repository containing the vocabulary specified within the HNOS, as well as an ontology description and classification of all participating network components. The registry is responsible for responding to network client queries, performing continuous reasoning over the ontology data, and adding relevant logical entailments to the existing ontology descriptions. The registry is more than a simply repository of attributes, as existing descriptions and ontology data can change over time, as the registry is provided with new information and classifications.

- Protocol Translator Components: Translator components are charged with interpreting between protocol and ontology domains. They are responsible for gathering the descriptions of components within a protocol domain and translating descriptions into ontology classes and instances found within the HNOS. Translators are responsible for a single protocol domain, but this relationship is not functional. A protocol domain may have one or more translators, as a translator may be designed for a specific group or category of network components. By using translators, a certain level of compliance can be had within a home network, without requiring intervention from either the registry or the protocol developers.

The ontology registry provides a single point of contact for network components to query for desired components while allowing a generic querying vocabulary to be used. The translation components remove the burden from both the registry and the protocol domains to conform to a specific approach. Chapter 10 discusses the advantages gained by using this system architecture in terms of the scalability of this approach. The rest of this Chapter is concerned with describing an implementation of this system architecture deployed within an OSGi framework.

9.1 Implementation Overview

Section 3.9 introduces the OSGI framework, a Java-based middleware platform capable of supporting a home network environment. This work provides a middleware implementation built upon the OSGi framework, allowing the use of the Java-based ontology tools available, as well as aspects of the OSGi framework itself. This implementation uses the Jena framework[1] to support the ontology registry, designed for the management and querying

[1](http://jena.sourceforge.net/)

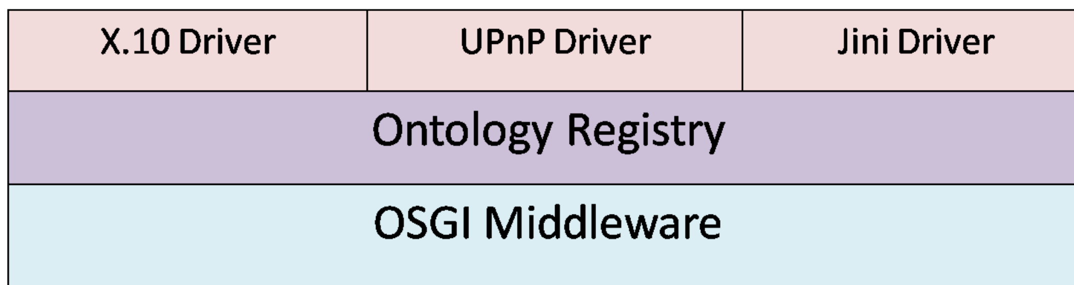


Figure 9.1: The Middleware Architecture

of ontology information. OSGi systems make use of driver bundles to control devices and services within the network. These bundles allow the framework to 'plug in' to a particular protocol. This work makes use of existing UPnP, Jini and X.10 driver bundles. Through these bundles, the descriptive attributes of protocol specific components can be exposed to the framework. Figure 9.1 contains a high-level representation of the middleware architecture. This section will expose implementation details of the registry and translation process used within this work. Chapter 12 evaluates the suitability and performance of this implementation in real-world systems.

9.2 The Ontology Registry

The registry contains a number of features to support and manage a home network environment. This section provides an in-depth description of the important features of the registry.

The Ontology Model

The ontology registry is charged with making ontology descriptions available to client components. The ontology registry also performs logical operations over the ontology information, such as inference and reasoning. In this manner, the ontology registry is more active than typical service registries. The ontology registry is more than a simple list of service and device types, rather it is like a library catalogue (like that described in Section 5.1). Facts can be implied from the meta data about the devices and services contained in the registry. As new information is added to the registry, new information can be inferred about existing device and service descriptions. In simple terms, a description of a component may grow from the original description provided to the registry in the first instance. This may happen unbeknown to the providing component.

The ontology registry uses the Jena Semantic Web Framework for managing the ontology operations. Jena is a Java library designed to support semantic applications, and can utilise RDFS and OWL ontology languages. Jena internally represents all ontology information in the form of RDF graphs, which provide a structured and scalable approach to storing ontology information. When communicating with users, Jena provides ontology information in a Model form, which stores information in statements or RDF triplets. Reasoning and inference is performed over the graph when operations are performed on the Model (such as the removing or adding of information). The Model object offers an API to the graph, allowing users to create, delete and modify ontology information.

The Simple Query Interface

This work assumes that no network client should need to understand the complexities of the any ontology querying language or how ontology information is represented internally within the registry ontology model in order to function within the network. To this end, a querying interface called the Simple Query Interface (SQI) has been developed to decouple the discovery process from any particular implementation of the inner ontology model. This interface is part of the system architecture, and is described in section 9.3 along with an implementation of this query interface included as part of the registry implementation.

The Display Manager

The registry also offers a visual display manager which is designed to provide a common presentation area for protocol translators. The display manager provides a centralised point of focus for displaying any information

concerned with the description or discovery aspects of the ontology-driven network. For example, protocol translators may wish to provide a wizard interface for users to configure network descriptions, alert to new network components, or to explicitly add or remove descriptions from the registry.

The display manager also offers functions to client components for drawing focus to a particular presentation or for providing alerts to users. For the most part, the display manager is hidden unless it is in use by a client component. As the registry is implemented within an OSGi framework, the display manager is implemented using Java Swing, and is therefore able to provide a consistent layout across multiple platform.

Ontology Plugin Management

The registry exposes functions for protocol translator components to exchange and modify information. The simplest approach to loading descriptions into the registry is to pass a string URL of the component description to the registry. The registry can then download the corresponding ontology. This approach, however, is not suitable for use with protocol translators as descriptions are typically dynamically generated, and are therefore unlikely to have a concrete URL from where they can be downloaded.

This issue is addressed in a unique way by this implementation. When a translator joins the network, it requests its own ontology model from the registry. This model contains the default information found within the HNOS, up to the Generic layer of the stack. The translator then loads protocol ontologies concerned with their own domain into their private model. The translator can then use this private model to create their own ontology classes and descriptions. When a description has been create within the private model of the translator, it is then *merged* with the model contained within the registry. In this manner, the registry no longer requires a concrete URL to the description, as it is given a fully-formed ontology class. Each private model is referred to as an ontology plugin.

There are a number of advantages gained by using this approach. Firstly, a sizeable amount of the reasoning process can be offloaded from the registry onto protocol translators. As ontology descriptions are created, they are reasoned over using the information available within the private model. As mentioned, the private model comes already loaded with a sizeable amount of information from the HNOS. When ontology descriptions are added to the ontology registry, they have already been subjected to a large amount of reasoning which the ontology registry is no longer required to perform. This does not mean the registry performs no additional reasoning, as additional ontology information may be present within the registry model which was not passed as part of the private model.

This approach also allows protocol translators to *buffer* multiple new descriptions and then add them to the registry in one interaction. As models are merged, the registry copies all new information from the private model into the registry model. Determining new information when merging models is a process handled by the Jena Framework. Having a degree of separation between private models and the registry model also proves useful in protecting against inconsistent descriptions. As descriptions are generated locally, inconsistencies can be detected and addressed within the private model before being added to the registry.

The Ontology Plugin Management component also contains a number of functions to allow translator bundles to quickly query the registry for meta data. These functions include:

- `getInstances(String classURI)`

  This function returns a list of all instances of the specified class type.

- `getPropValues(String classURI, String propURI)`

  This function returns a list of property values on the specified class type.

- `removeClass(String classURI)`

  This function requests the registry ontology model to remove the specified class. This function would be used when devices or services leave the network.

- `removeModel(Model model)`

  Translators can use this function to remove a number of classes from the registry ontology model. For example, when a translator is removed from the network, this function can be used to remove all classes contained within the private model from the registry.

9.3 The Simple Query Interface

Query formats within home networking protocols typically take the format '*Find me an instance of Class X which has attribute Y*'. A Class may be a service interface, or an instance of a device adhering to a taxonomy classification. This format can be represented through a format similar to an RDF or OWL statement: **(subject, property, object)**, or in this case **(Class X, attribute, Y)**. To represent this, the Simple Query Interface (SQI) provides a statement-driven interface for querying the ontology registry.

The SQI provides mechanisms for expressing three types of queries:

- Simple Queries: These queries follow a simple **(subject, property, object)** format.
- Complex Queries: A complex query represents multiple queries over the same subject.
- Meta Queries: A meta-query encapsulates queries which contain sub queries.

Using this set of queries, users can interact with the ontology registry, without needing prior understanding of how the information is represented within the registry, or the specific querying mechanisms used by the ontology model. Providing a querying interface also allows the underlying ontology registry to be replaced if required.

The implementation included in this work is designed for interacting with the Jena model. Jena provides a querying language called SPARQL [44], which is designed for RDF and OWL information. The implementation converts queries formed using the SQI into SPARQL queries which are specific to the registry implementation. The conversion of the three queries types are implemented in the following manner.

9.3.1 Simple Queries

As mentioned, a Simple Query follows a standard **(subject, property, object)** format and is represented through the interface as:

```
public SimpleQuery(String subject, String property, String object)
```

An example Simple Query may be:

```
new SimpleQuery('Device:TV', 'Location:hasLocation', 'Location:LivingRoom') ;
```

This query wishes to discover any instances of a TV within the living room. This query is broken into a SPARQL query which would take on the form:

```
"SELECT ?match " +
"WHERE {?match <Location:hasLocation> <Location:LivingRoom> .
  ?match <rdfs:SubclassOf> <Device:TV>}" ;
```

In basic terms, this query requests any classes which match the given pattern, with *?match* being used as a wildcard variable. Any classes matching are returned to the SQI in a result set. The SQI then parses this result set, extracts the elements denoted by *?match* and then returns this list to the querying client in the form of an array of string URIs. Each URI corresponds to a matching ontology class within the registry.

9.3.2 Complex Queries

Complex Queries are similar to Simple Queries in that they concern a single subject, but may have multiple conditions attached to them. Users can construct Complex Queries using the following interface:

```
public ComplexQuery(String subject) {
public void addCondition(String property, String object) ;
}
```

An example Complex Query may be:

```
ComplexQuery complexQuery = new ComplexQuery('Device:TV') ;
complexQuery.addCondition('Location:hasLocation', 'Location:LivingRoom') ;
complexQuery.addCondition('TV:receivesSignal', 'TV:Digital') ;
```

```
String queryString = "SELECT ?match WHERE {";
for (int i = 0; i<query.getConditionSize() ; i++) {
  Condition condition = query.getCondition(i) ;
  queryString += " ?match <"+condition.getProp()+"> <"+condition.getObj()+"> . }"
}
queryString += ?match <rdfs:SubclassOf> <"+query.getSubject()+"> }" ;
```

Figure 9.2: Algorithm for converting Complex Queries

```
String queryString = "SELECT ?match WHERE {";
       ?meta <Location:toLeftOf> <Location:Hall> .
       ?meta <rdfs:SubclassOf> <Location:Room> .
       ?match <Location:hasLocation> ?meta .
       ?match <rdfs:SubclassOf> <Device:Lamp> }" ;
```

Figure 9.3: Algorithm behind Meta Queries

This query expands on the Simple Query to discover a TV which can also receive a digital signal. The SQI implementation handles Complex Queries by iterating over the conditions on the query, and adding a new condition to the SPARQL query as demonstrated in Figure 9.2.

Any viable matches are then returned within an array to the querying component.

9.3.3 Meta Queries

Meta Queries are the mechanism by which clients can submit sub-queries as part of their query. For example, '*Find me a Lamp device which is located in the room to the left of the Hall*'. This method of querying embeds an inner query as the object of the query, allowing clients to query upon unknown information. In this example, the client does not explicitly state what the location of the device should be, but rather indirectly describes the location (as left of the hall).

In this example, the sub-query is an instance of a Simple Query described earlier. Meta Queries can also accept Complex Queries as the object of the query. This would be akin to adding an extra condition onto the object query, which in this case is querying for a location.

Meta Queries can be accessed using the following interfaces:

```
MetaQuery(String subject, String property, SimpleQuery query)
MetaQuery(String subject, String property, ComplexQuery query)
```

Suppose we wanted to form the query given in the above example using a Meta Query. This could be achieved by using the interface in the following manner:

```
SimpleQuery simple = new SimpleQuery('Location:Room',
                           'Location:toLeftOf', 'Location:Hall') ;
MetaQuery meta = new MetaQuery('Device:Lamp', 'Location:hasLocation', simple) ;
```

SPARQL offers a flexible approach to sub-queries, making use of wildcard variables. The SQI translates Meta Queries by first separating the query into sub-query parts (i.e into Simple and Complex Queries), and then substituting wildcard variables for unknown classes. For example, the above query would be translated into SPARQL using the sudo-code shown in Figure 9.3.

In this example, *?meta* would contain the valid matches for the location part of the query. This variable may hold more than one value, and so the query matches against any Lamps found within any locations matching the location criteria. When submitting Complex Queries as part of a Meta Query, each new condition is added onto the criteria for *?meta*. By allowing clients to query over information unknown to them, the number of interactions with the registry is reduced. This also allows clients to remain abstracted from information which they do not directly need.

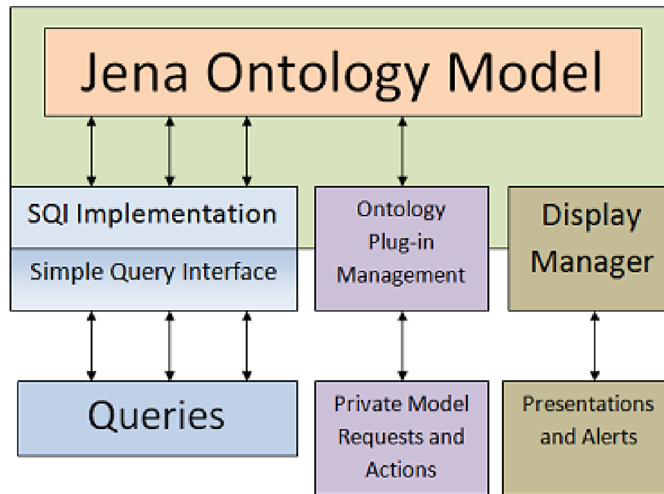


Figure 9.4: The Ontology Registry Architecture

9.4 Summary of the Ontology Registry Architecture

Figure 9.4 shows the architecture of the registry. The Simple Query Interface, Ontology Plugin Management and Display Manager are all exposed to client components, while the SQI implementation and the Jena ontology model is only accessible from within the registry. For the purpose of this work, all registry components were developed as part of the implementation, excluding the Jena ontology models.

This work is built around a centralised registry. This provides a single location for the addition of new information, and ensures information about the network is always up-to-date, consistent and dependable. An issue still remains over how to insert descriptions into the registry, which is handled by protocol translator bundles.

9.5 Protocol Translators

Protocol translators are charged with interpreting protocol specific descriptions into terms contained within the HNOS. Protocol translators provide the buffer between protocol specific terminologies and schemas and the common terminology found within the ontology stack. Protocol translators are specific in their implementation, with each translator designed specifically for a single protocol.

Protocol translators are in effect one way translators, from the specific to the generic. This is simply a design issue, and it is entirely feasible that translators could eventually become bi-directional or even multi-lingual. Having a one-way translator ensures this work is lightweight in its approach and deployment. Certainly, for the purpose of evaluation and research, a one-way approach is suitable.

9.5.1 The Translation Process

A translator interacts with its protocol domain, discovering new devices and services, parsing their descriptions, and adding ontology-generated descriptions to the registry. The translator contains a list of hard-coded associations between well known service and device categories, and their ontology equivalent. Where applicable, a translator also contains associated rules between protocol specific attributes and their ontology equivalent. For example, if a protocol vocabulary has an attribute denoting the manufacturer of a device, the translator will contain an association between this attribute and a **hasManfacturer** property within the protocol ontology. These associations are stored in hash tables called *association tables*.

The protocol ontologies within the Protocol layer of the stack contain the relationships between the protocol specific ontology attributes and those within the lower levels of the stack. As each ontology is tailored to a specific protocol, so the amount of association contained within a translator is dependent on the level of structure within the protocol description framework. As ontologies have been designed to describe UPnP, Jini and X.10 protocols, so three corresponding protocol translators have been implemented.

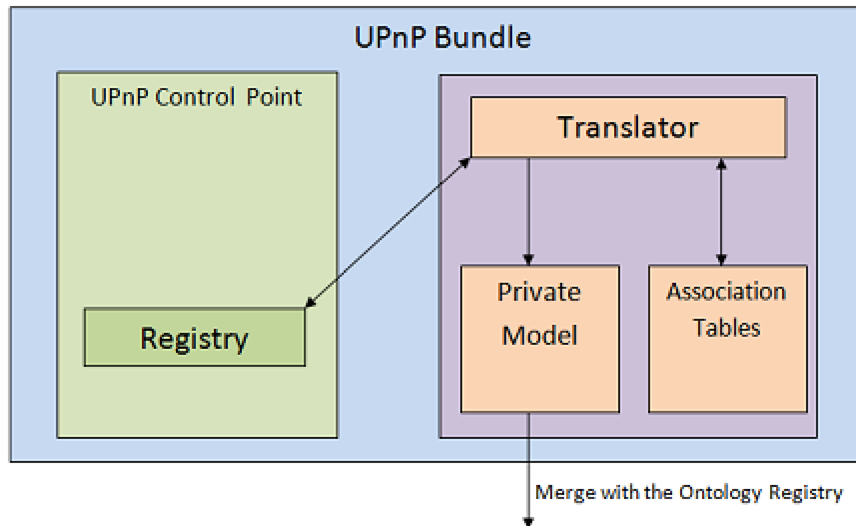


Figure 9.5: Translation within the UPnP Translator

9.5.2 The UPnP Translation Bundle

The UPnP translator is build upon an UPnP Control Point (described in Chapter 3). A Control Point is capable of discovering any UPnP device within the network, and retrieving its description. Leveraging this capability, the UPnP translator listens for all new devices joining the network. When a new device joins, the translator retrieves the device description.

Section 3.2 describes the contents of a UPnP description. In general, an UPnP description comprises of descriptive attributes of the device, along with URLs which provide descriptions of the services offered by the device. The UPnP translator parses the description using the functionality offered by the UPnP Control Point. The OSGi based Control Point processes an UPnP description into a series of UPnP objects, which can then be explored by the translator.

When the UPnP translator joins the network, it requests an Ontology Plugin object from the registry. As devices join the network, the translator enters a description into its own private model before merging the model with the registry. As a device leaves, the translator removes the ontology description from its private model, and indicates to the registry that it should also remove the description.

The translator contains a number of association tables which contain UPnP device and service types, and their ontological equivalent. In a similar manner, there are symmetrical relationships between UPnP properties and properties specified in the UPnP Ontology.

Figure 9.5 shows the process taken to translate UPnP descriptions. On being notified by the UPnP control point of a new device, the translator extracts the values of the UPnP descriptive elements from the control point registry. On retrieving the UPnP description, the translator consults with the association tables to construct an ontology description and enters it into its private model. The translator then initiates a merge of the private model with the registry.

At this point, there is a correlation between the UPnP object created by the control point, and the UPnP class created by the translator. Using the association tables, it would be possible to translate the ontology description back into an UPnP description.

The UPnP translator also makes use of the Display Manager offered by the registry to provide alerts for user interactions. When a UPnP device joins the network, an alert is presented to the user to allow them to indicate the location of the device within the home. An example alert is shown in Figure 9.6. The list of rooms is provided by requesting a list of Room instances from the registry.

It should be noted that the UPnP driver is not tailored to discover a subset of devices. Any UPnP device which joins the network will have a description *of some kind* generated. If a device joins and no corresponding associations are found within the association tables, a generic UPnP description is created and added to the registry, along with all properties and attributes found within the description.

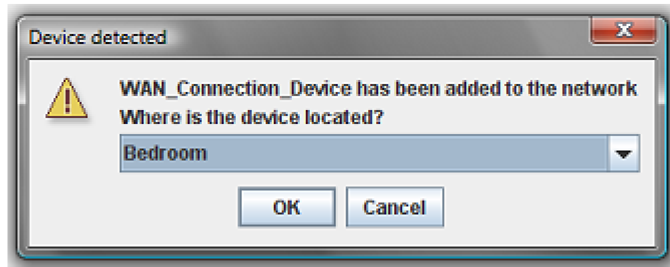


Figure 9.6: A UPnP Alert

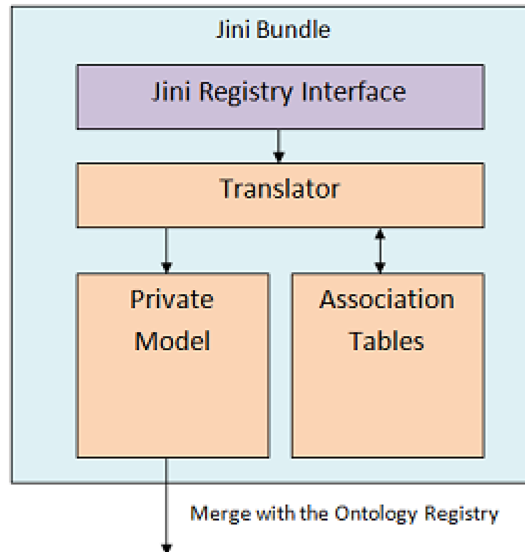


Figure 9.7: Translation within the Jini Translator

9.5.3 The Jini Translation Bundle

A Jini framework makes use of a service registry component for discovery within a network. The registry accepts and maintains a list of available resources within the network. To this end, the Jini translator is built upon an implementation of the Jini Service Registrar interface provided by this work. The implemented Jini registry responds to client discovery messages, and creates descriptions as service implementations are added to the registry. To a Jini client, the Translator is no different from a Registrar object designed to manage Jini networks.

Jini provides a unique challenge to the traditional view of the home network. A Jini service is a Java object. In essence, it is the object itself which offers services to the network, through the class functions. For the purposes of this work, a Jini service is treated as a device, or more correctly, a software interface to a device. The functions of the Jini service are therefore considered the services offered by the software interface.

As discussed in Section 8.2.5, Jini describes attributes through Entry objects. The type of each Entry object determine their ontology equivalent attributes. The translator recognises the set of Entry objects provided by the Jini API, using association tables to store relationships between Entry attributes and ontology equivalents. One entry object may contain a number of attributes.

The Jini translator also contains associations between a sample set of Jini service interfaces and ontology devices and services described within the stack. When a known service interface is registered, the translator creates a complete description of a Jini_Device class, including attributes described through Jini Entry objects. Ontology service descriptions correspond to the functions offered by the Jini interface.

Figure 9.7 shows the interactions within the Jini translator when a new Jini registration is received by the Jini registry implementation. The registry notifies the translator and passes the Jini description along with any Entry objects. The Translator accesses the association tables and creates a corresponding ontology class within its private model. Once completed, the model is merged with the ontology registry.

Figure 9.8: The X.10 User Wizard

The Jini translator suffers from a lack of well known service interfaces in the Jini framework which may be encountered in the home network. This is not a weakness in this approach, but rather a limitation on protocols which utilise interfaces in a similar manner. In a full-scale environment, a number of Jini translators may exist, with each translator designed for a set number of interfaces.

9.5.4 The X.10 Translation Bundle

In contrast with the Jini and UPnP translation bundles, the X.10 translator is user driven. A user interface has been created to support a user in integrating an X.10 device into the network, shown in Figure 9.8. The user interface concentrates on the device type, network address and room location of the device. As with the UPnP Translator, the list of rooms is populated through querying the ontology registry for Room instances. The list of devices is limited to those described within the X.10 ontologies. In the absence of any native description attributes, this information is enough to provide an ontology description of an X.10 device. The X.10 translator also contains a limited set of associations which translate between ontology references and human readable references used within the wizard.

An ontology description of the X.10 device is generated as the user describes the device through the interface, as shown in Figure 9.9. X.10 ontology descriptions benefit directly from the logical properties of the OWL language. For example, an X.10 device offers no explicit services to the network. If an X.10 Lamp is described in the registry, it will logically be related to the Generic Lamp described within the Generic layer of the stack. It therefore inherits the services offered by an instance of a Lamp device.

The X.10 translator allows this work to evaluate the efficiency of applying a service-orientated environment over existing simple home appliances. As X.10 modules allow existing appliances, such as lamps and air-fans, to be included in a home network, it is important that the protocol is not excluded in the description and discovery process.

9.5.5 Summary of the Translation Approach

Section 4.2 discussed the three main approaches towards description between the main home network protocols. The protocol translators implemented represent an instance of each approach:

- X.10: This protocol represents the ‘No Description’ approach. This requires heavy user interaction.
- Jini: This protocol represents the ‘Attribute/Value’ approach (as well as the centralised registry approach). The translator is tailored to what is known at the time. If an unknown Jini service joins, it can only be represented as a generic Jini_Device in the registry. Unknown Entry objects may also be encountered, as users are able to create their own attributes.

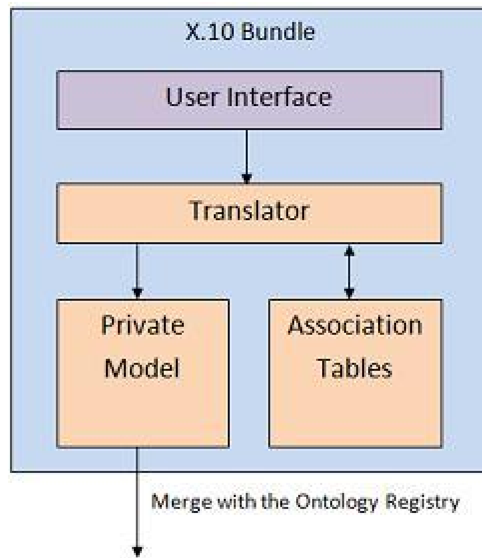


Figure 9.9: Translation within the X.10 Translator

- UPnP: This protocol represents the 'Schema' approach (as well as an environment without a registry). The translator knows what attributes can be expected, and has a good understanding of device types, which are tightly controlled. This translator is not final, in that it will require updating as new service and device types are created.

Using the system architecture described, this work can evaluate the suitability and relevance of the ontology vocabulary in various domains. In some situations, descriptions can be generated automatically, and invisible to the user. In other situations, user intervention is required. This represents the current state of the home network, which has not yet reached the levels of the 'zero configuration' environment.

By using the ontology vocabulary described in Chapter 8, and the system architecture approach described within this Chapter, this work is able to:

- Provide a single language required for discovering available devices and services in the network.
- Provide a single point of discovery for all network components, leveraging existing middleware frameworks
- Incorporate existing protocols, and their terminologies, within the ontology vocabulary.
- Blur distinctions between protocols at the description level.

The following Chapter evaluates and examines aspects of this approach which further allow this work to:

- Present a description framework which can evolve as the environment changes, by the inclusion of new protocols, service and device types. The framework is also adaptive to a changing definition of the home network environment.
- Recognise existing approaches, and can be reconciled with relevant existing upper ontologies.

This work does not aim to disregard previous work by other ontology and home network developers. Instead, this work identifies a weakness in current approaches and trends, and addresses this issue in a manner which can be applied retrospectively. To this end, Chapters 10 and 12 evaluate the contributions of this work. Chapter 10 revisits the scenario presented in Chapter 3 using an implementation of this work and clearly highlights the benefits of this approach in such a situation. Chapter 12 is concerned with evaluating the approach within existing environments, including measuring the effectiveness and scalability within home network domains.

Chapter 10

Deploying the Approach

The main claim of this work is that ontology languages can be used to support a protocol-independent discovery environment. This work has been concerned with providing both a language and framework for supporting this claim. The language is composed through an ontology-based syntax, and the framework has been augmented with components capable of understanding and managing the ontology language.

As a recap, the purpose of this approach is to support clients in discovering desired services, devices or other network resources. Without this support, clients would need to make provisions for the possibility of numerous protocols existing within the network when discovering network components. Returning to the scenario given in Section 3.11, suppose a client wishes to discover if there is a lamp device within the network capable of being client controlled through a power service. This seemingly simple operation can require the client to begin numerous processes which will allow this wish to be carried out in a multi-protocol environment.

10.1 Applying the Approach to a Home Network Environment

An underlying aim of this work is to reduce the protocol:vocabulary relationship to x:1. In such an environment, a single query can cover multiple domains. By using the ontology approach presented in Chapter 8, a client can query a registry using a single instance of a query, and be confident that all relevant matches will be returned. To evaluate this claim, the experiment described in Section 3.11 is revisited, using the implementation of this work described in Chapters 8 and 9. The purpose of this experiment is to highlight the advantages gained using this approach over existing solutions, showing where interactions are simplfied, and where protocol-specific interactions are removed.

10.1.1 Modifications and Assumptions

The ontology vocabulary has been designed to describe services and devices regardless of protocol. In this manner, services and devices can be abstracted from their protocol specific descriptions. It is not feasible to expect existing protocols to change their description approaches in order to comply with the ontology vocabulary. In some cases, descriptions may be hard-wired into the component, and in other cases components may not contain enough computational resources to adapt to a new vocabulary. The service providers used in the initial experiment are reused and unchanged, while the service clients have been re-engineered to make use of the ontology approach.

By allowing the existing Lamp services to remain untouched provides a level ground to compare this approach with existing approaches. By using the translation bundles described in Chapter 9, the protocol domains also remain unmodified, strengthening the claim of this approach that it can be applied in current and future home networks.

The Lamp Client

The service client has been embedded with a knowledge of how to interact with the Simple Query Interface. The client has not been augmented with any ontology understanding outside knowledge of classifications and attributes within the HNOS.

```
[A3_Lamp, rdfs:#subClassOf, X10:#X10_Lamp]
[A3_Lamp, rdfs:#subClassOf, X10:#X.10_Device]
[A3_Lamp, rdfs:#subClassOf, Core:#Device]
[A3_Lamp, rdfs:#subClassOf, Light:#Lamp]
[A3_Lamp, rdfs:#subClassOf, A3_Lamp]
[A3_Lamp, device:#hasAddress, "A3"]
[A3_Lamp, Device:#hasLocation, "Living room"]
```

Figure 10.1: Ontology Description of the X.10 Lamp

Client Assumptions

In this experiment, it is assumed that the client knows how to interact with services from each of the domains. As has already been stated, this work is not concerned with providing interaction information within a service or device description. It is assumed that, by examining an ontology description, a client user can extract information which then enables the client to retrieve protocol specific details from the service provider. For example, if a client discovers a suitable UPnP device, it can extract the UPnP address from the description, and then query the device directly for invocation details.

In the case of X.10, which has no native invocation description, a client can retrieve the address of a device from the ontology description. Using this information, a client can then interact with the device.

One reason for resisting the temptation of providing interaction details is this: This approach is designed to work in tandem with both existing protocols and existing middleware frameworks. This work does not offer any implementation of a specific middleware framework, other than augmenting existing component registries. Middleware frameworks for home networks already exist which can support cross protocol interaction, with OSGi being an example. It is assumed that client users operating within a home network environment will already have some concept of the distinction between protocols at the invocation layer. There is little common ground between interacting with Jini, X.10 and UPnP components, and so clients may adopt different approaches for interacting with each protocol (or more correctly, with each protocol driver).

With this in mind, it is assumed that by ignoring interaction details within a description, this work does not place any further burden on a client to know about how to interact with each protocol. Existing protocol drivers are able to blur protocol distinctions at the invocation layer, while this work is only concerned with the descriptive layer.

10.1.2 Ontology-Based Descriptions

In the OSGi-based experiment described in Section 3.11, descriptions generated for OSGi contained distinct attributes which largely were conveying similar meaning. For example, there were three different approaches to representing the type of service, and two different attributes for describing the location of each service. As the ontology vocabulary abstracts from the protocol, the ontology descriptions of each instance of the Lamp service take on a common set of attributes, most of which are either identical or logically related.

Within the OSGi registry, a reference to the controlling service or device is provided through the Java implementation object. When a suitable match is found in the registry, a client user can retrieve this reference in order to manipulate the service or device. As an ontology description is separated from the implementation, a reference is needed. An ontology description can contain implicit clues as to the protocol of the device or service being described. Properties may be protocol specific but have logical roots within the generic level of the ontology stack. By examining the properties of a description a client user can infer the protocol of the component being described. For simplicity, the protocol of the component is also referenced through the **usesProtocol** attribute of the device and service ontologies. Using this protocol reference, a client can, in most cases, then query the protocol domain to retrieve the service.

X.10 Registration

Within this experiment, the generated X.10 description differs slightly from that of the OSGi description, and is shown in Figure 10.1. A3_Lamp represents the X.10 Lamp, which is declared to be a sub class of a Lamp device, which is described in the Light ontology (denoted by Light:#Lamp).

```
[CyberGarage_Light_Device, rdfs:#subClassOf, upnpDev#UPnPLamp]
[CyberGarage_Light_Device, rdfs:#subClassOf, upnpDev#UPnPDevice]
[CyberGarage_Light_Device, rdfs:#subClassOf, Core:#Device]
[CyberGarage_Light_Device, rdfs:#subClassOf, Light:#Lamp]
[CyberGarage_Light_Device, rdfs:#subClassOf, CyberGarage_Light_Device]
[CyberGarage_Light_Device, Device:#offersService, upnpservice:power:1]
[CyberGarage_Light_Device, Device:#hasSerialNumber, "1234567890"]
[CyberGarage_Light_Device, isRealService, "true"]
[CyberGarage_Light_Device, Device:#hasLocation, Living_Room]
[CyberGarage_Light_Device, upnpDev#hasFriendlyName, "CyberGarage Light Device"]
```

Figure 10.2: Ontology Description of the UPnP Light

```
[upnpservice:power:1, rdfs:#subClassOf, Core:#Service]
[upnpservice:power:1, rdfs:#subClassOf, Service:#Power_Service]
[upnpservice:power:1, rdfs:#subClassOf, urn:schemas-upnp-org:serviceId:power:1]
[upnpservice:power:1, upnpSer:#hasSCPDURLValue, "/service/power/description.xml"]
[upnpservice:power:1, upnpSer:#hasEventSubURLValue, "/service/power/eventSub"]
[upnpservice:power:1, upnpSer:#hasServiceIDValue,
"urn:schemas-upnp-org:serviceId:power:1"]
[upnpservice:power:1, Service:#usesProtocol, UPnP]
[upnpservice:power:1, upnpSer:#hasControlURLValue, "/service/power/control"]
```

Figure 10.3: Ontology Description of the UPnP Service

UPnP Registration

The UPnP description is similar in content to that generated for an OSGi registration. As UPnP provides both device and service descriptions, so there are two descriptions generated, shown in Figures 10.2 and 10.3. Note that the UPnP device is denoted as a sub class of Light:#Lamp. This fact is generated by an association between the UPnP device type and a classification from within the Generic layer of the ontology stack. The UPnP service is declared as a sub class of a power service for the same reason. The UPnP device description contains a link to the power service through the **offersService** property.

Jini Registration

The Jini description generated for this experiment is shown in Figure 10.4. In this instance, the subclass relationship between the Jini service and the Light:#Lamp interface is inferred by the registry. This is due to the presence of the Device:#hasCategoryType and Device:#offersService properties. The inference is based upon the pre-existing of the Light:#Lamp description.

```
[RegistrarClient.LampImpl, rdfs:#subClassOf, Jini:#JiniDevice]
[RegistrarClient.LampImpl, rdfs:#subClassOf, Core:#Device]
[RegistrarClient.LampImpl, rdfs:#subClassOf, Light:#Lamp]
[RegistrarClient.LampImpl, Device:#hasCategoryType, Device:#Lighting]
[RegistrarClient.LampImpl, Device:#offersService, Service:#Power_Service]
[RegistrarClient.LampImpl, Jini:#isInBuilding, "LivingRoom"]
[RegistrarClient.LampImpl, Jini:#isOnFloor, "Home"]
[RegistrarClient.LampImpl, Jini:#hasRoomLocation, "1st"]
```

Figure 10.4: Ontology Description of the Jini Light Interface

```
SimpleQuery query = new SimpleQuery(
"core:Device", "rdfs:subclassOf", "Light:#Lamp") ;
ArrayList instances = registry.submitQuery(query) ;
```

Figure 10.5: Simple Ontology Query

10.1.3 Discovery with an Ontology Vocabulary

An ontology description is similar in layout to schema-based descriptions, but may also leverage properties of the ontology language. An ontology description contains well-defined properties, which are logically grounded in generic and abstract ontologies. Ontology classes are also logically defined, containing sets of properties which can be inherited by child classes. Armed with this knowledge, client users can access a powerful discovery environment.

For example, suppose Device X is defined as a Device with Device_Category *x*, and offering Service *y*. A client who has knowledge of Device X may also know about the attributes of the class. If a client wishes to discover Device classes which offer Service *y*, they need only search for instances of Device X. It is obvious that such a simple assumption would not be sufficient within a real-world example, as multiple Device classes may also offer Service *y*. This example serves to highlight the logical assumptions that can be made while discovering components within the home network. Just as schema based descriptions can be relied on to contain a set of defined properties, so an ontology description can be relied upon to hold true to any class definitions contained. If Device XImpl is of type X, it can be assumed to adhere to the definitions specified within the definition of Device X.

These properties of an ontology description greatly simplify the discovery process within a home network environment. To support this claim, consider the query expressed to discover all instances of a Lamp Device, given in Figure 10.5.

This query requests all instances of all Devices (denoted by core:Device) which are children of the generic Lamp_Device class (denoted by rdfs:subclassOf, Light:#Lamp). By examining the descriptions of the X.10, UPnP and Jini devices, the ontology registry matches the query to each description, as each description shares a root within the Light:#Lamp class. All descriptions are also instances of the core:Device class.

On submitting this query to the ontology registry, the following results are returned:

```
[RegistrarClient.LampImpl, CyberGarage_Light_Device, A3_Lamp]
```

The first outcome of this experiment is concerned with the protocol:vocabulary relationship. A single query has been expressed which discovers all instances of the Lamp Device. The query makes no distinctions between protocols, concerned only with discovering all available devices. It is built upon the assumption that all instances of the Lamp_Device class offer a power service. As already discussed, this assumption is viable due to the logical properties of ontology languages: A child class inherits the attributes of the parent class. Based upon this assumption, the protocol:vocabulary relationship has been reduced to a simple *x*:1 relationship. One query is sufficient because all available descriptions utilise a single vocabulary, the ontology stack.

Assumption-less Queries

As has been shown, ontology queries can be formed upon assumptions drawn from class descriptions and the ontology language. Suppose a client user does not have the knowledge to express the query given above. This client has no concept of the ontology within the Generic layer of the stack in which the Light:#Lamp class is described. Instead, the client only has knowledge of the lower levels of the ontology stack. It is important that network clients which do not have full knowledge of all represented ontology classes are not prevented from discovering desired components. In lieu of knowledge about the Light:#Lamp class, a client can discover all available Lamp Devices by constructing a more complex query, given in Figure 10.6.

This query requests to be notified of any instances of the Device class, which has category ‘Lighting’ and offers a power service. This query is in essence constructing a class which matches the Light:#Lamp class. The registry parses the query in a sequential fashion:

- The registry first lists all instances of the core:Device class.
- From this list, the registry discards any instance which do not match the **hasCategory** property.

```
ComplexQuery query = new ComplexQuery(core:Device) ;
query.addCondition("Device:#hasCategory", "Device:#Lighting") ;
query.addCondition("Device:#offersService", "Service:#PowerService") ;
ArrayList instances = registry.submitQuery(query) ;
```

Figure 10.6: A Complex Ontology Query

- From the resulting list, the registry discards instances which do not offer a PowerService service.
- The registry then returns the list[1].

In this manner, clients can discover desired services and devices based upon attributes rather than explicit classification. This allows the discovery process to scale as the number of classes or classifications increase. This process also maintains the x:1 relationship between protocols and vocabularies required for discovery. As the vocabulary remains abstract from the protocols in the environment, clients can be assured that their queries will be applicable regardless of what protocols are in existence.

Extracting Meta-Data From the Registry

It is important that any description framework is able to support connections between the component description and the component protocol. When a client discovers suitable services or devices, a mechanism is required to indicate to the client how to interact. The values returned by the registry are references to services or devices within the home network, and do not convey any protocol details to the client. By themselves, these references only specify what components exist within the network.

Interaction is presumed to be protocol specific. In this implementation, the middleware provides support for protocol specific interaction, and therefore removes the need for details to be included within a component description. A client need only discover what protocol a component uses, before then utilising the middleware for interaction. The ontology registry provides further querying methods for extracting meta-data from component descriptions. On retrieving a reference from the registry, a client can query the registry as to the protocol of the referenced component:

```
ArrayList protocols = registry.getPropValues(reference, "Device:#usesProtocol") ;
```

Meta-data queries are concerned with a particular attribute or property. As an ontology class may contain more than one instance of an attribute, the registry simply returns all known values. In the above example, the query simply retrieves the value of the **Device:#usesProtocol** property on the given reference.

In the same manner, clients can request meta data about any given ontology class.

Logical Querying

Up until this point, querying over ontology information, or even querying within an OSGi registry, relies on string matching. In other words, the properties and values expressed within a query are explicitly matched to classes within the ontology registry. As discussed in Section 5.4, OWL is more than a simple mark-up language. OWL provides logical properties for describing classes and concepts. The ontology registry leverages this logical advantage in its management of the ontology information. It is right that the querying process be able to leverage these same advantages.

For example, the Location ontology (found within the stack) contains a property, **isNearTo**, for describing relative positions of rooms within a home. In addition to this property, two more properties have been created for defining relative locations of rooms: **toLeftOf** and **toRightOf**. These properties are declare to be sub-properties of **isNearTo**: If Room A is to the left of Room B, then it can also said to be near Room B. This logical property can be taken advantage of in the querying process.

In this experiment, additional context information has been added to the ontology:

```
(Bedroom, location:toLeftOf, LivingRoom)
```

[1]The registry first removes erroneous classes from the list. These are classes which logically satisfy the query, but are irrelevant to any home network clients.

```
ComplexQuery query = new ComplexQuery(core:Device) ;
query.addCondition("Device:#hasCategory", "Lighting") ;
query.addCondition("Device:#offersService", "Service:#PowerService") ;
query.addCondition("location:isNearTo", "Bedroom") ;
ArrayList instances = registry.submitQuery(query) ;
```

Figure 10.7: A Logic Based Query

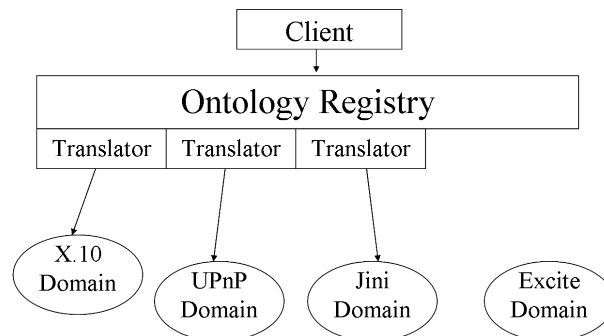


Figure 10.8: Unsupported Excite Protocol

The previous query presented in Figure 10.6 can be modified to include an additional property, shown in Figure 10.7. This query requests a Lamp Device which has a location near to the Bedroom. This query successfully matches the three instances of Lamp Devices present within the network. This is because the **(location:isNearTo, Bedroom)** part of the query is disseminated into **(location:toLeftOf, Bedroom)** and **(location:toRightOf, Bedroom)**. This logical inference therefore matches on the descriptions of the available components.

10.1.4 Adding New Protocols

In evaluating this approach, three sets of protocol ontologies have been developed for X.10, UPnP and Jini. This is obviously enough for networks which contain only these three protocols. In environments where other protocols exists, suitable protocol ontologies require to be developed.

Suppose a new protocol emerges, *Excite4Home*. In the existing home network environment used in this evaluation, any Excite4Home components are undiscoverable by the ontology approach. Without any protocol driver or ontology, there is no way to reconcile the Excite vocabulary with the ontology stack. It is important that an approach which can be applied in existing home network environment is also applicable to future environments. If a standard emerges for describing existing protocols which is not scalable or extensive, emerging protocols and components are required to conform, or be backward compatible with that standard.

The approach presented in this work is able to scale as the environment develops. To incorporate Excite4Home into an ontology-supported home network, a translator component and a Excite4Home ontology set requires to be developed. The translator is responsible for associating Excite description attributes, interfaces and classes with terminology within the Excite4Home ontology and home network stack.

In this manner, an ontology-based Excite4Home vocabulary emerges. This vocabulary, which is grounded within the ontology stack, allows Excite components to be registered within the ontology vocabulary. Once components are registered, they can be discovered by clients. Suppose a client wishes to discover available data storage components before the Excite translator is installed in the framework. An Excite hard disk is added to the network, but is undiscovered by the client, shown in Figure 10.8.

After some time the translator is installed and available Excite component descriptions are translated and registered in the ontology registry. The client submits the same query to the registry, and is now returned the Excite hard disk as an appropriate component.

This scenario highlights the extensiveness of this approach. As new protocols emerge, they can be installed into the ontology domain by use of translator components and the development of a protocol ontology set. Clients within the network can remain oblivious to the addition of a new protocol, as the ontology vocabulary abstracts from protocol specific attributes. The installation of new translators and ontologies can be performed at run time,

without necessarily requiring a framework restart. As the domain grows, the ontology vocabulary also grows to ensure that the approach remains relevant to the number of protocols in the network.

10.1.5 Conclusion of the Ontology Approach

The ontology approach presented in this work simplifies the discovery process in the home network. It achieves this by moving the complex nature of cross-protocol discovery from network clients onto the protocol driver and translator components within middleware framework. In this manner, network clients no long require knowledge of multiple vocabularies and description approaches when discovering available services and devices.

A network client need only have knowledge of the ontology vocabulary in order to discover any instance of the component, regardless of protocol. This approach relies on the translator bundles converting protocol specific descriptions, and the registry matching client requests to available network components. The registry is involved in applying a logical process to the discovery process (e.g. searching for an Audio service will discover UPnP and HAVi Audio services). This logical step may be taken when descriptions are added to the network (classification based upon existing information in the registry) or at discovery time (e.g. a client submits a description of a desired class, the registry classifies this description, and then returns instances of that class).

The ontology approach is not a static one. It can scale as the environment expands (as discussed in Sections 8.2.6, 10.1.3 and 10.1.4), allowing the approach to be applicable to evolving network environments. Allowing the vocabulary to be customised allows the approach to be applied in environments where a higher level of description is required.

To further highlight the robust and versatile nature of this approach, Chapter 11 describes the use of this approach and system architecture with the MATCH project. The MATCH project is concerned with providing home care systems to a variety of users. As part of the home care system, the project requires an abstracted discovery framework to support the high-level approach toward service description and usage. This work has been deployed within the MATCH system to assist in the provision of home care through network services.

Chapter 11

The MATCH Project

MATCH (Mobilising Assistive Technologies for Care at Home) is a project concerned with designing and deploying home systems for providing a level of care and support. A MATCH system is specifically tailored to the home user or users, thus systems may differ in implementations. The purpose of a MATCH system is to provide support for both those with complex care needs and those who provide support. This may include family members, formal and informal carers, and health professionals.

A MATCH system is built upon a number of home care services and devices, which work together to achieve *goals*. Goals may include supporting the independence of a user, providing user data to health care professionals, or monitoring the state of the home to prevent dangerous scenarios. Similar to a home network, a MATCH system utilises existing devices, services and assistive technologies which may be of differing protocols and vendors. A MATCH system can be customised to suit the needs of a user and their support network (carers and health professionals). This customisation is not limited just to when the system is installed, but can also occur over time, as the needs of the system users change. This requires the system to be adaptive and resilient, capable of carrying out desired goals by using the network resources available to it.

In order to provide reliable resource discovery within a potentially evolving environment, the MATCH system utilises the approach described in this work to support the description of services and devices. An implementation of the work described in Chapters 8 and 9 has been customised to the domain of home care systems [96, 95, 89]. This chapter describes this implementation and customisation, highlighting the aspects of the project which have benefited from this work.

11.1 System Architecture

A typical MATCH system can be described through three different viewpoints:

- How the system interacts with users.
- How the system communicates internally.
- How the system performs its tasks.

11.1.1 User Interaction

As already discussed, a MATCH system utilises existing technology to achieve its goals, and interact with users and the environment. This technology includes standard sensors (e.g. movement), motors, X.10 modules, Infra-Red transceivers, touch screen monitors and audio equipment (e.g. microphones and speakers). To allow control over these devices, relevant software services are deployed within an OSGi framework. In this manner, the functions of hardware devices can be offered to the system through software interfaces. In addition, the MATCH system also provides a natural interface for users to configure or create goals for the system.

Utilising this technology, a MATCH system can interact with users in multiple ways (e.g. touch, speech, movement). New technology can be added to the MATCH system as it becomes available. Similarly, interactive technology may leave the system, and so the system is adaptive to the resources available to it.

Care services operating within the OSGi framework provide functionality to the MATCH system. These services may be simple software interfaces to hardware (e.g. a service which offers audio output), or higher level services which offer more complex operations (translate string values into an audio format).

11.1.2 System Communication

A MATCH system may contain a variety of protocol specific devices and services (much like a typical home network). An OSGi framework can offer support for cross protocol communication through driver bundles. MATCH does not utilise this feature, instead making use of a message broker system to communicate with system components. By using a message broker, components can be developed and deployed quickly, without reliance on appropriate driver bundles being present. Communication can instead take place by abstracting from any protocol or platform, similar to that of a web service environment.

To communicate, system components send and receive messages in a publish-subscribe manner. For example, a movement sensor interface publishes movement messages to the broker. A movement monitoring service can subscribe to these messages, and therefore receive any relevant messages from the broker network.

Components which offer messages to the system own a specific channel within the broker system. Using this channel, a component can publish messages to the network. Message types are specific to a component classification. For example, all sensors which monitor movement within the home may publish messages of the same type, MovementMessage. In this manner, a component may have the following attributes:

```
Component Type: PIR Sensor
Channel:        movement1
Message Type:   MovementMessage
```

A service designed to monitor movement in the home would then subscribe to the channel **movement1**, and begin receiving MovementMessages. As messages are exchanged in a manner abstracted from protocol, drivers are not required to support any interaction.

A message router is responsible for subscribing clients to channels, as well as routing message to appropriate subscribers. A system component need only locate the message router in order to act within the system. In this manner, there is no explicit binding between system components.

11.1.3 Core System Components

In each MATCH implementation, there are four main components which configure and control the main aspects of the system. These are:

- The Policy Server.
- The Task Manager.
- The Interaction Manager.
- The Resource Registry.

Using these components, a MATCH system can evolve to suit a user's need, while being able to adapt to changes in the available network resources. Each component plays a specific role in the MATCH system.

11.1.4 The Policy Server

The policy server allows all users of the system (which may include carers and health professionals) to configure desired goals for the system to achieve. For example, suppose a user wishes to be notified of any daily appointments at 9am every day. A user can configure the system by implementing a *policy* to achieve this goal. A policy is a rule, or set of rules, which are triggered by an event. In this example, the policy would be triggered at 9am every day.

Policies may be triggered by events internal and external to the system. Policies provide a means for users to customise the system, in a fashion which can be applied at run time. The policy server maintains a library of policies, which may be active or inactive. The server also contains relevant context information about all aspects of the environment, such as home layout and user capabilities.

11.1.5 The Task Manager

As no explicit binding takes place between system components, the task manager co-ordinates interaction between system components. Suppose a policy expressed the goal: When someone walks in the front door, switch on the hall light. Rather than contain explicit details on which light to switch on, or what sensor to subscribe to, a task is created which encompasses this action. This task is a continuous action, which maintains a relationship between any sensor which monitors the front door and any light which is in the hall. Once a task has been created by the task manager, it becomes an independent entity. The task manager may modify or stop a task, but it is no longer responsible for the actions of the task.

A task may be an abstract entity which is required to be initialised with parameters. For example, a **SwitchOn-When** task requires to have two parameters to be initialised: A component to switch on, and an event to listen for. A task may also require no parameters, being pre-configured, and simply requires to be started by the task manager.

In some cases, a task requires explicit tailoring in order to perform successfully. Consider the example given in the policy manager description. The task to be performed requires the system to communicate with the user to notify of daily appointments. In this example, there is no specifications as to how the system should communicate with the user. In these circumstance, the task manager requests assistance from the interaction manager.

11.1.6 The Interaction Manager

The interaction manager is charged with providing interaction options to the task manager. The interaction manager provides recommendations based upon various information sources, such as available components, user capabilities and contextual information. Using the appointment example, the task manager requests appropriate recommendations from the interaction manager. The interaction manager may consult with the policy manager to determine the capabilities of the user. If the user is deaf or hard of hearing, then any audio notification is not suitable. Similarly, if the user finds it hard to point or touch surfaces, a touch based confirmation is not suitable.

The interaction manager also consults the resource registry in order to determine available suitable resources. The registry contains descriptions of all available devices and services within the system and network. The interaction manager queries the registry and retrieves relevant components. The interaction manager ranks discovered components in order of relevance to the task, and returns recommendations to the task managers.

11.1.7 The Resource Registry

The resource registry maintains a list of device and service descriptions. The resource registry is an implementation of the ontology registry described in Chapter 8. It provides a means of describing system components in a manner abstracted from protocol or vendor, with descriptions being grounded within the home network ontology stack. An overview and evaluation of the registry is presented in Section 11.2.

This implementation of the registry is less reliant on protocol drivers, and more on user-driven configuration. This is because the MATCH system provides a method of communication independent of protocol, and so varying levels of user-driven configuration will already be required (e.g. for providing communication and context information). As part of the resource registry, a configuration wizard is provided for quick installation, modification and removal of component descriptions.

11.1.8 System Review

By utilising the message broker system, and by the inclusion of the four main system components, a MATCH home care system can provide a high degree of customisation for system users. Having a component-like approach toward providing care services allow a library of care services to be created. In this manner, services can be selected for relevant situations, and removed when no longer required. By utilising the policy server and resource registry, the system can be manipulated at run time, without requiring a system restart.

As mentioned, the resource registry is an implementation of the ontology registry, described in this work. In order to fully support the MATCH system, shown in Figure 11.1.8, the ontology stack has been expanded, with ontologies specific to MATCH being added. An overview and evaluation of the use of an ontology approach is given in the following sections.

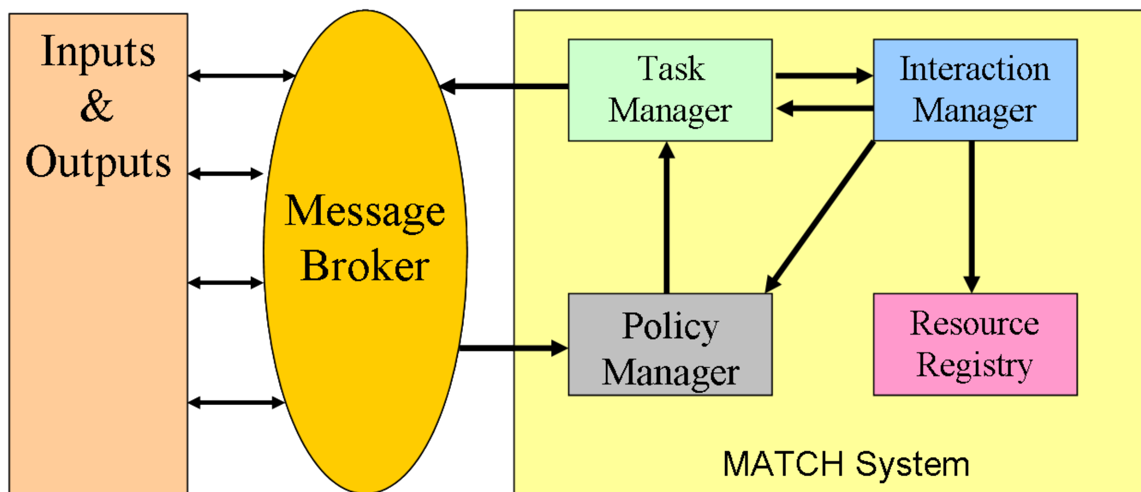


Figure 11.1: The MATCH System

11.2 The Role of Ontologies within MATCH

The MATCH project utilises an ontology-based component discovery approach for the following reasons:

- The ability of the ontology language to describe components in a manner abstracted from protocol or vendor.
- The ability of the vocabulary to adapt and scale to meet user and system requirements *at run time*.
- The ability of the vocabulary to describe interaction details of components.
- The ability of the ontology registry to provide a logic-based discovery environment.

Some of these advantages are inherit within the approach itself, while others are found by customising the approach to the project domain. As the MATCH system operates at a level above protocol specific details, the use of the HNOS and ontology registry are well suited to this domain. The description language is independent of the system platform, allowing descriptions to persist, even if the underlying system is changed.

11.2.1 Describing Components

Initially, the MATCH project utilises a limited range of assistive technology. The catalogue of devices used has been translated into a set of ontologies, which are positioned within the core and generic layers of the stack. In a similar manner, a small set of available network care services has also been described.

MATCH components not only abstract from protocol, but also from concept of device or service. Components are represented through a Component ontology class, but continue to have many of the attributes contained within the core levels of the stack, such as category and location. In addition to describing Components, MATCH ontologies also contain scope for describing Tasks. Tasks are a special instance of the Component class, reserved for use within the task manager.

11.2.2 Evolving Vocabulary

A MATCH system may be deployed within a users home for various reasons. Some systems may be a short-term solution to user needs. Other systems may be designed to provide continuous, long-term support for users and carers. Components within the system may be replaced with new versions, and new components may be added as technology matures.

It is the intention of the project that a system should not require constant on-site maintenance or system restarts when applying system changes. To support this intention, a MATCH system is deployed upon an OSGi framework which allows remote framework management. As discussed in Sections 8.2.6 and 10.1.4, adding new ontologies to the stack can be performed at runtime. The registry stores all ontology information in a model, which is constantly changing, based upon the ontology information available.

```
ComplexQuery query = new ComplexQuery("Base:Component") ;
query.addCondition("Component:#hasCategory", "Device:#MovementMonitor") ;
query.addCondition("Component:#offersMessageType", "Message:#MovementMessage") ;
ArrayList results = registry.submitQuery(query) ;
```

Figure 11.2: A Sample MATCH Query

```
[TomsPersonalTracker, rdfs:#subClassOf, Match:#PersonalTrackingComponent]
[TomsPersonalTracker, rdfs:#subClassOf, Core:#Component]
[TomsPersonalTracker, Device:#hasCategoryType, MATCH:#PersonalMovementMonitor]
[TomsPersonalTracker, Movement:#tracksPerson, "Tom"]
[TomsPersonalTracker, Component:#ownsChannel, "TomTracker"]
[TomsPersonalTracker, Component:#offersMessageType, Message:#PersonalMovementMessage]
```

Figure 11.3: Description of TomsPersonalTracker Held by the Registry

11.2.3 Describing Interaction Details

Components within the MATCH system use channels to exchange messages through the message broker. Once a component has been discovered using the registry, the component client requests the interaction details from the registry, found within the component description.

Tasks and Components both contain an **ownsChannel** and **offersMessageType** which provide information on how system components can interact through the message broker. Excluding these properties, a MATCH component is described in a similar manner to that of a device or service. For example, consider the description of a wireless PIR sensor presented below:

The MessageType pointed to by the **offersMessageType** property is a concrete ontology class. In this manner, a MessageType may contain attributes and relationships. In particular, MessageType classes can be sorted into sets of messages. For example, a PersonalMovementMessage is concerned with the movements of a specific individual within the home. A PersonalMovementMessage is sub-class of the MovementMessage class.

It is important to note that the original HNOS is not concerned with describing any interaction details, as system interaction is largely system dependent. Home networks are not guaranteed to share a common communication approach, like that found within UDDI-like web services. The inclusion of interaction details within the MATCH project demonstrates the versatility of OWL in describing and unifying various domains. The MATCH project required the addition of specific ontologies to the HNOS vocabulary, but this requirement does not invalidate the intention or usefulness of the vocabulary. It instead serves to highlight the suitability of this approach in addressing various home network scenarios, in this case: a home care network.

11.2.4 Logic Based Discovery

A MATCH system can leverage the advantages of logic-based discovery already described in this work (Section 10.1.3). For example, suppose a component which monitors movement around the home wishes to discover any components which detect movement and publish MovementMessage information. In interacting with the registry, the query shown in Figure 11.2 is formed.

This query simply requests instances of Components which are of the category MovementMonitor and offer MovementMessages. Suppose a PersonalTrackingComponent offers real time information about a specific person as they move around the home. Within the registry, this component can be represented as that given in Figure 11.2.4.

The logical aspect of the ontology descriptions allow the query of the monitoring component to locate TomsPersonalTracker. This is because of the logical relationships within the vocabulary. Figure 11.4 depicts the relationships between classes found in the description of TomsPersonalTracker and those within the MATCH ontologies (and subsequently the HNOS). With the logical sub-class relationships in the class descriptions, the registry matches **Device:#MovementMonitor** to **MATCH:#PersonalMovementMonitor** and **Message:#MovementMessage** with **Message:#PersonalMovementMessage.**

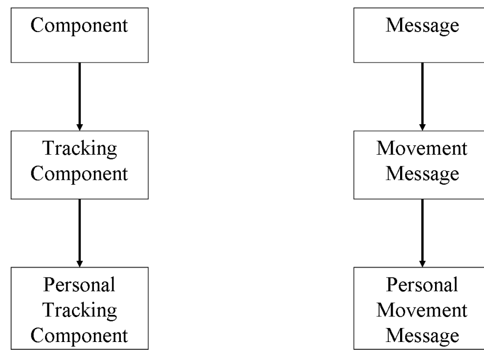


Figure 11.4: Relationships within the MATCH Ontology

Chapter 12

Evaluation

This work presents an approach to describing and discovering components within a home network. Evaluating this approach proves a challenge within itself. Some description frameworks, such as OWL-S and SWSF, are under continual development or are designed for a web service domain. This work has identified issues within these frameworks and, where applicable, has attempted to address these issues.

Relevant work within home network domains, such as the NASUF (Section 7.1.1) and GAS (Section 7.1.2) ontology projects, are concerned with exploring the novelty of an ontology approach within the home network domain. General evaluation is given to the success of the approach, rather than comparisons with existing approaches or middleware frameworks.

Upper ontologies, such as those described in Section 7.3, are typically large, collaborative efforts. Some projects, such as the Cyc project [55], have provided open-source versions of their ontologies. It was an initial aim of this work to relate terms within the HNOS to relevant terms found within publicly available upper ontologies, in particular the OpenCyc ontology provided by the Cyc project. After initial investigation, it was found that this aim was not immediately feasible, due to the size of the single ontology provided.

In the absence of a standardised form of ontology or framework evaluation, the following aspects of this work have been evaluated:

- The Protocol/Vocabulary Relationship.
- The Reasoning Approach of the Registry.
- The Responsiveness of the Registry.
- The Range of the Vocabulary and Approach.
- The Versatility of the Approach.
- The Limitations of the Approach.

12.1 The Protocol/Vocabulary Relationship

Chapter 1 introduced the issue of multiple vocabularies within home network environments. As the number of protocols within the domain increase, so the number of vocabularies used may also increase. Existing home network protocols and middleware do not typically share common sets of vocabularies. In this environment, clients are required to know multiple vocabulary sets in order to discover across multiple domains. In this environment, the relationship between existing protocols and vocabularies required can be naively represented as x:x. This relationship is highlighted in Section 3.6.

Section 10.1 describes the experiments carried out within a home network environment supported by the ontology approach. This approach provides a supportive description and discovery environment, designed for networks which may contain multiple vocabularies. This approach unifies existing protocol vocabularies. In this manner, existing protocols are not rendered obsolete or incompatible with the middleware framework. As new protocols join the network, new ontology representations of the protocol vocabulary are added to the HNOS within the registry. Existing clients can discover new components, described using existing ontology-based terminology.

A client therefore only requires to know the ontology vocabulary, which is initially captured within the HNOS. Therefore, the protocol/vocabulary relationship can be represented as *x:1*.

12.2 The Reasoning Approach of the Registry

The registry implementation is proactive in reasoning over new information. As new information is added to the registry, the reasoning process is invoked to ensure the ontology model contains all relevant information as soon as it is available. This is similar to the *eager loading* pattern commonly found within software programming, where objects are fully initialised when created [35].

Issues associated with eager loading are concerned with the relationship between the amount of time and resources required to fully initialise an object compared to the extent to which it is used. For example, suppose object A contains ten fields, with each field referencing another object. In initalising object A using eager loading would require each field to also be initialised, and hence a further ten objects require to be initialised. Suppose a client of object A is only interested in the first field. After examining this field, the client disregards object A as it has no further use for it. In this instance, the process of intialising object A is very inefficient, as nine other referenced objects are unneeded but are still initialised.

The converse approach to eager loading is *lazy loading* [35]. Lazy loading is concerned with only initialising and loading objects and fields when required, in an 'on-demand' manner. Using the example of object A, when the client wishes to interogate the first field, object A first checks to see if this field has been initialised. If it has not been initialised, it initialises the field and then returns the result to the client. If the client then has no further need for object A, a substantial saving has been made as the other nine objects are not initialised.

Within the ontology registry, eager loading involves the registry reasoning over all new information as it becomes available. For example, suppose ontlogy class A was stated to be a subclass of class B as part of its description. On being submitted to the registry, all known information about class B is added to the description of A. Suppose as part of this information, class B is also related to class C. This information is also added to the class A description on the initial load of information.

A potential drawback to this approach is that this information may never be accessed by any client, either directly or indirectly. No client may desire to discover information about class A or class B. Additionally, the reasoning process may be a time-intensive operation, as numerous facts and information may be infered. The registry takes this approach to ensure all information is available when reasoning over new information.

Lazy loading within the registry would require the opposite approach, where reasoning is only initiated when a client wishes to know about a specific ontology class or instance. From a functional point of view, ontology information would therefore only be reasoned over when directly required. In practice however, to perform full reasoning, the whole ontology model would require to be reasoned over to ensure any applicable entailements are added. For example, to perform reasoning over class A would require class B to be reasoned over, which in turn may require class C to be reasoned over. In practice, lazy loading simply moves the processing time from components uploading descriptions to those querying for descriptions. It is for this reason that this work adopts a pro-active, eager-loading approach to reasoning, ensuring client quering times are kept to a minimum.

12.3 The Responsiveness of the Registry

This section is concerned with the data stored within the registry, and the responsiveness to querying and modification. This evaluation will also measure the volume of data captured and inferred by the registry, and correlate the volume with the registry responsiveness. The volume of data will be measured in terms of RDF statements and number of ontology classes present. The test-bed machine used in evaluation contains a dual-core CPU running at 2.4GHz, with 4GB of memory. This evaluation is performed in three stages:

- Evaluation of the Home Network Ontology Stack.
- Evaluation of a MATCH system environment.
- Evaluation of a registry with additional ontologies.

The ontology registry stores a model of ontology metadata, including classes and relationships. This model can change continuously, as new data is added or removed. If the model has changed, the new data must be

Stack Level	Ontology Classes	RDF Statements	Initial Query (s)	Secondary (s)
Base	12	84	0.065	0.001
Core	148	1368	0.189	0.006
Generic	168	1511	0.226	0.009
Protocol	197	1886	0.27	0.009

Figure 12.1: Response times of the Registry using the HNOS

Stack Level	Ontology Classes	RDF Statements	Initial Query (s)	Secondary (s)
MATCH	322	2725	0.479	0.010

Figure 12.2: Response Times of the Registry Within the MATCH system

reasoned over. If the model remains stable, no reasoning is performed. It is for this reason that querying may encounter two different registry states. If a query concerns ontology information which has not yet been reasoned over, the registry first performs reasoning, and then returns relevant matches back to the user. Additional queries over the same information do not initiate the reasoning process. In this manner, initial queries may take longer to return than subsequent queries. Querying time is largely dependent on the amount of information within the registry.

This evaluation has captured the querying time from two different perspectives: The initial query, and the secondary query. The initial query represents a query over un-reasoned data, while the secondary query represents subsequent queries over the same data. The query expressed is intended to represent the worst case scenario: the retrieval of all instances (and sub classes) of the owl:Thing concept. This concept is the root of all ontology classes, and all ontology classes are a member of the owl:Thing set.

This evaluation is initially performed using each level of the HNOS stack.

12.3.1 Evaluation of the HNOS

The stack is built in a 'bottom up' manner, with each level of the stack containing more meta data than the level below. In this manner, the higher the levels of the stack used, the more information is contained within the registry. This is reflected in the query response times of the registry, shown in Figure 12.1.

This data set represents an environment where the registry contains the complete meta-data set contained within the HNOS. This environment does not contain any 'live' network components, but simply the descriptions of possible components (within the Generic and Protocol layers). An implementation of the MATCH system has been used to implement a more 'real world' environment.

12.3.2 Evaluation of the MATCH System Environment

The MATCH project involved the development of additional ontologies for describing available MATCH devices and services. In particular, the MATCH ontologies describe several assistive technology devices and network services used by MATCH components. These ontologies, coupled with the HNOS, provide a rich environment for describing home care components, while also providing a large amount of meta-data within the registry. The query response time of the registry within a MATCH system is shown in Figure 12.2.

What is becoming clear is that, while the initial query time would seem very much tied to the amount of data within the registry, the secondary querying time remains extremely low throughout. As explained, the initial query instigates a reasoning process by the registry on all unreasoned data within the model. When no information is reasoned over, the response times are minimal.

A home network is potentially unbounded in size. As the number of network components increase, so the amount of information within the registry also increases. To further examine the emerging properties of the query response times, external ontologies have been imported into the registry to increase the amount of information present.

12.3.3 Evaluation of the Registry with Additional Ontologies

To fully evaluate the capability of the ontology registry, larger ontologies have been imported in order to increase the amount of data. These ontologies have been developed by Gavin Campbell and Ken Turner for supporting

Ontology Name	Ontology Classes	RDF Statements	Initial Query (s)	Secondary (s)
GenPol	250	3672	0.395	0.009
WizPol	606	7757	0.620	0.014
HomeCare	790	9196	0.744	0.017
Sensor	871	9619	0.791	0.015

Figure 12.3: Response Times of the Registry using Imported Ontologies

the Appel Policy Language [14, 13]. These ontologies are not designed for a home network environment, but are written in OWL, and therefore share the same logical properties as the HNOS. The query used for evaluation continues to be applicable to these ontologies, as all ontology classes are members of the OWL:Thing class. The evaluation results are given in Figure 12.3. As can be seen, while the initial query time continues to increase as the data increases, the secondary (and subsequent) query time remains stable. For example, the Sensor ontology taken from the Appel ontologies contains more than three times the RDF statements contained within the HNOS and MATCH ontologies, and yet the secondary querying time is a mere 0.005 seconds longer.

12.3.4 Response Evaluation

As a component description can own one or more RDF statements, and a home network is potentially unbounded in components, it is difficult to equate the amount of information used within this evaluation with any upper bounds or threshold within a home network environment. For this reason, this section is simply an evaluation of the registry with respect to the volume of data used, with no concrete conclusions drawn over what the data represents.

It should be mentioned that the final entry in this evaluation (the Sensor ontology) contains nearly three times the ontology classes than the HNOS and MATCH ontologies. From a naive viewpoint, these extra classes could equate to 500 device and service descriptions within the network if it is assumed that each new class represents a description instance of a device or service. Similarly, the Sensor ontology contains nearly four times the RDF statements, which can directly equate to just under 7000 descriptive statements.

Figure 12.4 shows a comparison between the initial and secondary response times. The initial query time displays a clear relationship with the amount of data within the registry.

The initial query represents the first ontology operation to be performed upon the whole model. In other words, no operations have been carried out upon the data other than consistency checking. It is for this reason that the initial querying time is not in line with the secondary time. In comparison to existing home network protocols, the initial response time for the Sensor ontology (0.8 seconds) would be deemed appropriate. The UPnP and HAVi architectures both specify a timeout period greater than 1 second [90, 50]. As this response time is only applicable to the initial query, client A may receive a delay of 0.8 seconds, but client B would then experience a delay of only 0.015 seconds (and subsequently for client C). What becomes clear is that the initial response time will not be experienced by the vast majority of network clients. Instead, the secondary response time can be used as the standard waiting time clients should expect to experience. From a naive perspective, the initial response time appears to be follow an exponential trend as the amount of data grows.

The response time is due to various factors such as ontology complexity, the size of the model and the computational resources available. As these factors can vary, it is difficult to present a relationship between the volume of data and the response time of the registry. One key claim of this work is that it is scalable, and suitable to apply in a potentially expanding environment. This evaluation supports this claim.

Consider the table shown in Figure 12.5. In this comparison with the Core level ontologies, the Sensor ontology increases the ontology classes found within the registry by a factor of 6, and RDF statements by a factor of 7. The response time is increased by a factor of 2. While it may be difficult to form this relationship into a formula, it is clear that as the volume of data increases within the registry, the response time does not scale in a similar manner. The response time remains extremely favourable, and well within acceptable response times (such as those presented by UPnP and HAVi). In this manner, this work is valid in its claims to be scalable within the domain.

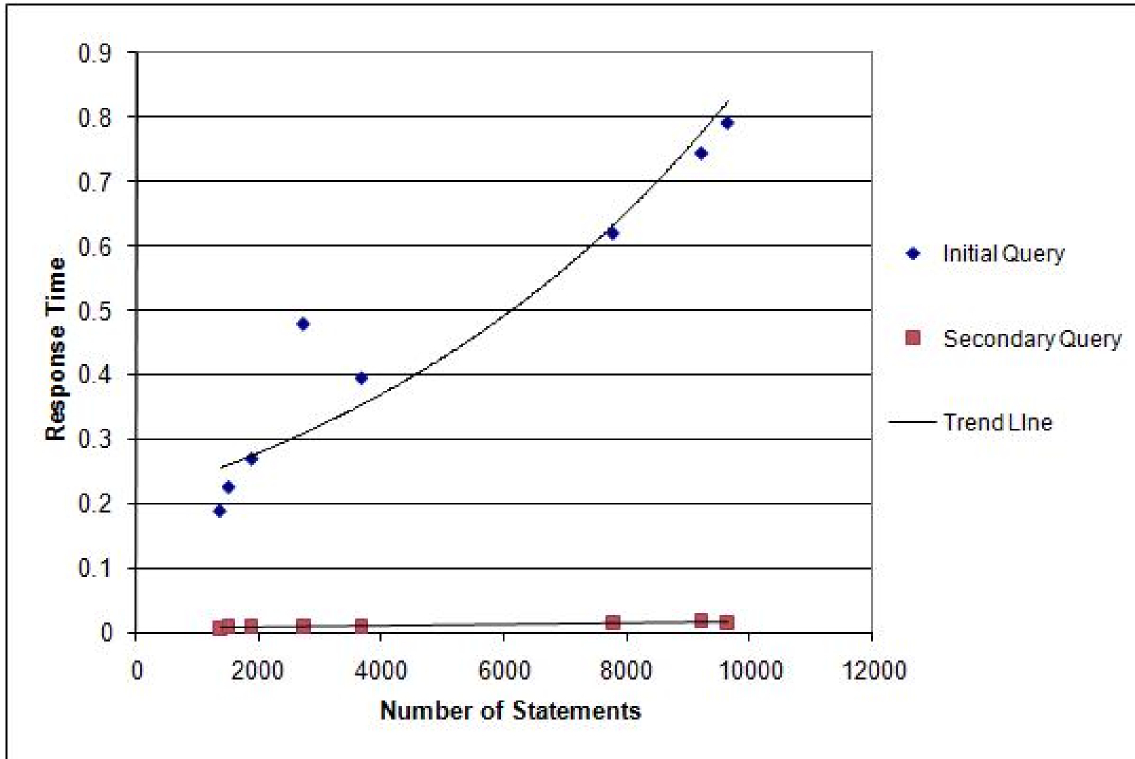


Figure 12.4: Comparisons of Response Times

Ontology Name	Ontology Classes	RDF Statements	Response Time (s)
Core	148	1368	0.006
Sensor	871	9619	0.015

Figure 12.5: Comparison of Secondary Response Times

12.4 The Range of the Vocabulary and Approach

This section will explore the potential domains out with the home network where the HNOS and registry approach could be applied. This section will concentrate on two domains in particular: A web service domain, and a peer-to-peer domain.

12.4.1 Web Service Deployment

Successful web service environments are built upon a common description framework and messaging format (discussed in Chapter 6). Standard approaches, such as UDDI, already exist within the domain for describing services. As this work is designed to provide unification to existing description vocabularies, it would have little effect in a domain which contains a single framework approach.

Web Service definitions, such as UDDI, OWL-S and SWSF, contain a structured format for describing services, specifying a set of defined properties which may be found within a description. Some properties can be constrained in their values, but others may open ended with no set range defined. One such open ended property found within UDDI, OWL-S and SWSF is that which specifies the category of the service. This property is deliberately left unbounded as services may refer to a number of external taxonomies. At this point, part of the approach presented by this work may be used.

One of the main purposes of the HNOS is to allow protocol specific descriptions to be grounded within more generic attributes and classes. In this manner, a client can request services of a generic category and discover all protocol specific implementations. By leveraging this characteristic of the HNOS, web service registries could be augmented to allow clients to discover using the same principle. Web clients could then discover using a generic classification, with the registry performing the logical operation of determining all instances of that classification.

In this manner, web service clients could operate in multiple web service domains. Web service registries can then take the logical step between the generic classification requested, and the specific classification used within the domain.

12.4.2 Peer to Peer Domains

Peer-to-Peer networks provide a unique network domain. Peer-to-peer (P2P) networks can vary in type and structure, but in general are devoid of centralised resource registries. In such a variable environment, it is difficult to see how the approach presented in this work can be easily applied. In networks where peers are responsible for maintaining their own registries, the amount of ontology information which would be replicated in each peer may deem this approach prohibitive. In this environment, it would be important to restrict the amount of ontology information to only that which is relevant to the node. To support this requirement, it would be equally important for the ontologies within the ontology stack to be designed in a tight manner, with each ontology severely limited in what it describes. A node would then contain only information limited to the services provided, including any possible network variants. Such variants would correspond to sub-services or requesting using different terminologies.

It would be sufficient to assume that the stack-approach presented in this work would require to be more tightly defined and controlled in order to be applied to a P2P environment. As P2P is typically adhoc in both peer capability and service provision, the vocabulary would require to be equally wide-ranging. While the approach presented in this work is extensive, the management of the descriptions (the SSDB) is suited to a controlled environment where services are for a particular domain. It is at this existing work, such as the DiSUS [31] component within the NASUF, and the GloServ architecture [6, 5], would prove to be more suited to this environment.

12.5 The Versatility of the Approach

Chapters 8 and 10 discuss aspects of the approach which can be extended and reused. This work claims these aspects ensure the approach remains relevant and applicable as the domain changes. This section examines these claims in terms of the ontology stack, the system architecture and implementation.

12.5.1 The Home Network Ontology Stack

Section 8.2.6 discusses the extendibility of the ontology stack, citing the use of the approach within the MATCH project. This project required the vocabulary within the core and generic levels of the stack to be extended. To

achieve this requirement, the stack was extended to incorporate new devices and service types, as well as other context information specific to the home care domain. This project highlighted the extensiveness of the vocabulary approach. The core and generic levels are potentially unbounded in the amount of information which they may contain, and it is these layers which allow the stack to claim to be extensive.

From a high level view, the core level provides the vocabulary for the approach, while the generic level provides the classification aspect. The core level can therefore be extended as required to describe the domain, while the generic level provides *meaning* to the vocabulary. As an aside, this work recognises that new information being added to the stack is of little use if no other ontologies make reference to it. Some support can be gained from logical inference, but in general new information requires new ontologies to make use of it to avoid being redundant.

This work provides three sets of protocol vocabularies developed to address three specific protocols. It is important the approach described within this work is not limited to domains containing only these protocols. Protocol ontologies are built upon all other levels within the stack. As the top-most level of the stack, the protocol layer is versatile, allowing protocols to be removed and modified with ease. This is because no ontologies depend on the information within this layer. The exception to this statement is that protocol ontologies may reuse ontology information within other protocol ontologies, but this is likely to be confined to protocol ontology sets. (E.g. UPnP ontologies may reuse information found within their own ontology set, but are unlikely to reuse ontology information from other protocols.) In this manner, the ontology level of the stack can be extended to include new protocols, and modified by removing or updating existing protocol ontologies without invalidating the ontology stack. It is this characteristic of the protocol layer which allows this work to claim to scalable. The ontology stack is viable regardless of the number of protocols and protocol ontologies within the domain, whether this be many or none at all. As the domain scales (new protocols entering the network) the stack can scale with it by simply adding a new protocol ontology into the protocol layer.

12.6 The System Architecture and Implementation

The Ontology Registry supports the extensive aspect of the Ontology Stack by allowing new ontologies to be added to the core and generic levels of the stack at run time. As new information is added to the stack, the registry reasons of the new data and infers any new facts discovered. While the registry is in itself a static component, the amount of ontology data it can managed is limited only by computational resource. While this would appear to initially be an obvious and unimportant aspect of the registry, it obscures an important characteristic. The registry does not constrain what *type* of ontology information is submitted, other than it is expressed in an OWL format. This allows information which may be initially unrelated to the Ontology Stack to be loaded into the registry, potentially extending the purpose of its use beyond service and device discovery. While this aspect is not one which this work intentionally offers, it does highlight the open-ended implementation of the architecture.

In symmetry with the scalable aspect of the Ontology Stack, the system architecture is designed and implemented with scope to allow multiple protocol translators to be present within a network. The registry is not required to provide protocol translators, but rather translators are required to work with the registry. This allows the system architecture to scale with the domain. If only one protocol and corresponding translator is present within the network, the registry behaviour is unchanged from there being multiple protocols and translators. The registry does not force a one-to-one relationship between protocols and translators. Multiple translators can act on behalf on a single protocol when interacting with the registry. In simple terms, as the number of protocols expand, and the number of protocol ontologies increase, the protocol translators can scale similarly to support the universal discovery approach.

In removing the burden of providing and managing protocol translators from the registry, the system architecture can claim to be component orientated. Translation is provide where the facilities exist, allowing the approach to be dynamic and lightweight. One drawback of this approach is that there is no guarantee of an appropriate translator being available when relevant protocol ontologies exist. In this scenario, implicit knowledge about discovery in a certain domain can exist without any means to carry this discovery out. It is believed that this weakness is outweighed by the lightweight approach with the system architecture. To address this weakness would require the registry to own translators for every possible protocol, which can quickly become a heavyweight approach.

12.7 Limitations of the Approach

This Section will discuss the limitations of this work, and provide suggestions as to how these limitations may be addressed.

12.7.1 Logical Metadata

This work is grounded within OWL, an ontology language which contains logical properties capable of supporting inference and reasoning. As a descriptive logic language, OWL descriptions contain a number of statements which are always logically true, but may have no relevance to the purpose of the description. For example, consider an ontology class named ClassX. Within an ontology model, such as that used by the ontology registry, the following statements potentially exist about ClassX:

```
[ClassX, rdfs:#subclassOf, ClassX]
[ClassX, rdf:#type, ClassX]
[ClassX, rdfs:#subclassOf, owl:#Thing]
[ClassX, rdfs:#subclassOf, owl:#Class]
[ClassX, rdfs:#disjointWith, owl:#Nothing]
```

For every ontology class that exists within the model, there may be up to five statements which exists describing logical properties of the class. (In this case, ClassX is both a subclass of and type of ClassX. It is also a subclass of the owl:Thing and owl:Class classes stated within the OWL syntax.) Suppose a home network contains only three component descriptions. The registry would contain up to fifteen statements which simply describe logical properties which can either be taken for granted, or are not relevant to the description purpose. This relationship is proportional, but can quickly give way to scenarios where ten descriptions infer up to fifty statements. These statements are of course logically true, and do not mislead in any way. If the registry is executing on a powerful platform, this meta data should not affect performance. However, on platforms where computational resources are at a premium, this logical meta data may begin to slow down the discovery process.

12.7.2 Agreement on Standards

In a domain where issues have arisen because of a difference in description vocabularies and approaches, there is a need for existing and emerging protocol developers to agree on the terms and ontologies within the HNOS. Where the HNOS does not meet their need, perhaps because of new technology or classification, it is important that when adding a new ontology to the stack, the principal of Generic ontologies is maintained.

For example, suppose one developer creates the ontology class Sony:3DTelevision and another Panasonic:3DTelevision. If there is no 3DTelevision class declared within the generic level of the stack, both classes remain distinct from each other. Protocol, or in this case Vendor, specific components require to be grounded within the generic and core layers of the stack. Otherwise, clients wishing to discover 3DTelevisions would require to search for the Sony and Panasonic classes separately.

This issue can be difficult to resolve if two developers create similar ontology classes at the same time. If Sony and Panasonic develop their televisions within the same time frame, who should be charged with defining the generic ontology class? In such an instance, it would be more effective for a third party organisation to be charged with defining new ontology classes for the core and generic layers of the ontology stack.

12.7.3 Component View of the Network

The approach given in this work requires a particular view of a home network. The ontology stack views a service as being offered by a device. For the most part, this assumption is suitable: Networked devices offer networked services. Jini highlights where this view becomes less straight forward. Jini services are not offered by a device, but by a software component. More specifically, within the Jini domain a service offers functions. In this manner, the Jini service is the offering component, and the functions are the services offered.

What this observation highlights, is that it is important for developers to attach meaning to the properties of a class, rather than a class name. If a component offers functionality, then it can be classified as a *Device* without requiring to be a manifestation of a device. Similarly, functionality is classified as a service, even if it is not classified as such within its own domain.

This process of thinking is crucial to the success of this approach, and yet it can be difficult to reconcile with traditional views of devices and services. For example, suppose a component offers a Lighting service. In discussing home network resources, the ‘Lighting’ term represents one way of thinking about a service. Within this approach, a Lighting service does not immediately resolve into a simple description. Answers must be asked of the ‘Lighting’ term.

For example - what does a Lighting service do? A simple assumption is that a Lighting service provides light in a location through being switched on and off. This assumption highlights the need for a specific view of the home network. A Lighting service allows a *Light* to be *switched on and off.* The component is the Light, while the function being offered is a power function (on/off). ‘Lighting’ may represent a high level concept, but it must be disseminated into Device and Service terms in order to be discoverable within the network.

As an aside, some of the awkwardness of this approach may be resolved by swapping Device with Component. In this manner, Components offer Services. Scenarios, such as Jini, become simple as a service is no longer a Device but a Component. This change is merely a change of the URI of the Device component.

12.7.4 Translators

This work presents a vocabulary which is scalable and extensive. The work also describes a middleware implementation of a resource registry which is supported by protocol translators. The success of this approach relies on these translators to plug into multiple protocol domains. As a protocol specific component joins the network, an ontology description is generated based upon the protocol vocabulary. The reliance on translators has the potential to limit the impact of this approach within the home.

A translator is capable of translating between two known vocabularies. In the case of a schema-based vocabulary, such as UPnP, this translation is supported by description templates. These templates, including properties and values, can be known in advance by translators. Translator developers can configure the transformation between the protocol vocabulary and the ontology vocabulary in advance. This approach works extremely well where protocols already contain a tightly controlled description vocabulary.

This approach is less effective in domains where the vocabulary is not controlled. Jini contains a small set of description attributes (Entry Objects) which are provided with the architecture. The Jini Architecture also allows Jini developers to create their own set of Entry objects. This is a common feature of protocols which adopt an attribute/value approach toward descriptions. Users may specify their own attributes and values. Protocol translators have no way of determining ontology equivalents if the knowledge is not provided by developers before hand. If an unknown attribute or value is encountered, a translator can not provide an equivalent description in the registry.

This issue is not limited to this work. It is a potential weakness in all home network environments, supported or unsupported. Clients are typically pre-programmed to search for specific resources with specific attributes. For a successful discovery process, clients and providers must utilised a shared vocabulary, which may be defined only for the local network, or may be standardised by a third party.

As new vocabularies emerge, and new devices and services are created, translators require to be updated to ensure the most recent conversion knowledge is available. Translators may eventually contain multiple protocol specific attributes which correspond to a single ontology attribute. There is potential for translators requiring constant updating to incorporate new vocabularies. The work and approach presented in this work is capable of supporting this scenario, as translators, which are OSGi bundles, can be updated at runtime, without any need to restart the registry or framework.

Chapter 13

Conclusions and Future Work

This work has identified issues which required to be addressed for the home network vision to become reality. The scenario explored in Section 3.6 and experiment detailed in Section 3.11 highlight the motivation behind this work. A home network should be able to contain multiple components and inter-operate without exposing any protocol distinctions to the user.

13.1 Home Network Conclusions

To achieve this vision, a network service provider is required to:

- Describe the service in a manner which can be discovered by any components within the network.
- Expose the service in a manner which allows clients to use the service.

Similarly, network service clients must be able to:

- Express discovery requests in a manner understandable by all components within the network.
- Interact with the service in a manner understandable by both provider and client.

In existing home network solutions and approaches, a vast amount of research and development has been given to the secondary requirement of both clients and providers. One well-explored option to support service invocation is through middleware frameworks. A second option which emerges is by using common communication protocols, and by using well-known interfaces or invocation languages, such as WSDL. It is the observance of this work that this requirement is well addressed by the research and industry communities.

It is also the observance of this work that the initial requirement of both providers and clients has received little attention. In the majority of work encountered and observed, networks rely on an attribute/value approach towards describing providers, and submitting similar attribute/value queries from clients to the network. As has been discussed, this approach does not scale well, and is certainly not suitable for adhoc and evolving environments. There is a need for all components to share a common approach to description and to utilise a shared vocabulary. As the environment may change through emerging protocols, components and requirements, this description approach cannot afford to be static.

This requirement is not easily solved. To require all existing and future components to adhere to a specific approach may not be feasible within itself, due to the amount of vendors, developers and protocol providers involved in the domain. This is not an issue limited to home network domains. As presented in Section 1.1, a real life object such as a computer can be described using various sets of vocabulary, with each unique to a nation or ethnic group. In a real-world setting, societies which contain many different languages may become separated and distinct. This possibility is reflected within home networks, highlighting the need to address this issue.

Chapters 6 and 7 reviewed existing work within web service and home network projects which were concerned with describing services. On examining these projects, an important issue emerged. To simply provide a new description approach risks alienating existing protocols, while providing no clear incentive for developers to adhere to the approach over other existing description approaches.

For these reasons identified, this work has provided a description framework, a vocabulary (Chapter 8), and a method to apply this approach to existing and emerging home networking protocols and solutions (Chapter 9). Using this approach, home networks need not be distinct and isolated at the descriptive layer. This approach is able to scale as the domain grows (Section 12.1), and able to be tailored to emerging protocol specific description schemes (Section 8.2.6). By applying this work, home network developers and users can ensure that protocol distinctions are resolved by the network, without the need for user intervention. In simple terms, the complexity of discovery within multi-protocol environments has been removed from the network clients and onto the middleware framework and description vocabulary.

The implementation described in Section 10.1 included protocols which use each approach identified within Section 4.2. It is recognised that this evaluation is by no means exhaustive in terms of the number of protocols. These protocols were chosen due to existing integration tools for the purpose of experimentation. While a more comprehensive evaluation would include more existing protocols, this work demonstrates there is no requirement for these existing protocols to be modified or rendered obsolete in ontology-based description environments.

The implementation demonstrates that this approach is more than simply a theory or future standard, but a immediate solution to existing home network issues. The implementation also shows this approach to be generic and versatile, being able to be deployed in environments out with the standard home network (as demonstrated within the MATCH project discussed in Chapter 11). In this manner, this work can claim to be technically relevant and applicable to existing environments, being able to be implemented and automated with existing technology and protocols. This claim is important in a domain where existing approaches and projects are commonly described and discussed at a level abstracted from implementation detail.

13.2 Review of Work

In the initial chapter of this work, a number of aims were specified. These aims are required to be met in order to provide a comprehensive solution which can support existing and future home network environments.

Provide a generic home network vocabulary

The Home Network Ontology Stack (HNOS) has been developed specifically for describing network components at a generic level. By using terms and concepts from the Generic level of the stack, and those below, network components can be described independently of how they are provided.

Inclusion of existing protocol vocabulary

The protocol level of the HNOS is designed to allow existing vocabularies to be incorporated into the framework. In this manner, protocol specific properties can be captured and remain specific to the domain, while generic properties can be related to properties found lower in the stack. In this manner, terms from protocol specific domains which are semantically similar can be logically deduced to be so. Syntactical differences no longer render properties logically distinct from each other.

Providing an extensive and scalable framework

Sections 10.1.3 and 10.1.4 discuss the scope of this approach in terms of scalability and extensiveness. Section 12.2 evaluates the efficiency of the implementation presented in this work. In both cases, the approach proves resolute: capable of including new vocabulary, which can immediately participate in discovery operations, without encountering relative slow-down in the response times of the registry.

Provide support for existing protocol specific components

The approach presented in this work makes use of translation components to incorporate existing home network components. In this manner, existing home network components are not excluded from the approach. Network users are therefore able to compose high level services based upon existing technology which is not constrained to a particular protocol or vendor. This approach can also be applied to emerging protocol specific components, blurring the distinctions between protocol domains.

Devise an approach suitable for multiple environments

Section 12.4 discussed the method behind applying this work in different environments. The description vocabulary presented is independent of any particular environment or implementation. Similarly, the middleware approach presented is also generic, while the implementation is specific to an OSGi middleware framework. Section 12.4 discusses the requirements of new environments where this approach may potentially be applied. This work need not be constrained to a single middleware framework, or a specific set of protocols.

13.3 Future Work

The idea and approach presented in this work is based upon a simple idea. A home network may contain many protocols. Each protocol may have its own approach in terms of communication, operation and description. These aspects can render a protocol incompatible with other protocols in the network.

The ontology vocabulary described in this work is abstract from protocol, vendor or platform. While the translation implementation is specific to OSGi, the approach behind this implementation is applicable in any domain capable of supporting a multi-protocol environment. This work can be taken as an on-going approach to providing an environment where all aspects of home network management and use can be provided at a level higher than protocol, platform or vendor. This section proposes future work in the areas of middleware frameworks and service invocation.

13.3.1 Deploying the Approach Within Other Middleware and Environments

This work presents an home network implementation managed by an OSGi middleware framework. OSGi is well suited to this approach as existing protocol driver bundles can be modified to support the translation process. It is important that this approach is not tied to a single middleware platform. This work is concerned with removing boundaries between protocols and platforms, it would seem conflicting to only be applicable to a single middleware environment. For this reason, it is important that the approach be deployed within other middleware frameworks. Candidate frameworks require to have the following features or capabilities:

- Support a Java Virtual Machine: A JVM is necessary for the Ontology Registry to operate. The underlying ontology management application (Jena) and querying interface (SQI) is written in Java.
- Provide translation 'Services' which can interact with protocol domains: These services provide interaction across protocol boundaries in terms of description and registration. These components may be built upon existing protocol drivers, or developed for this purpose. There is no requirement for these components to be able to invoke protocol-specific actions or services. These actions may be carried out by other components.

A natural choice for an additional implementation would be upon a Jini framework which, like OSGi, is service orientated.

A Jini-Based Implementation

A Jini framework satisfies the above requirements. The Jini architecture is written in Java, and requires all service, or at least all service proxies, to be written in Java. A Jini environment differs to an OSGi environment, as all network services are free to execute independently of the framework. (In OSGi, all services execute within a single JVM). This allows the home network to take a distributed approach to discovery. The Jini registry would be used by clients to discover the ontology registry.

For the most part, the translation bundles used within the OSGi implementation could be easily adapted to a Jini network environment. These bundles would also require to discover the ontology registry by means of the Jini service registry. After discovery of the ontology registry, the translation bundles could operate identical to that of an OSGi-based environment.

This example is concerned with a Jini-based environment, but highlights the potential of this approach. After the initial discovery of the ontology registry, translation bundles and clients can interact out with any support from the Jini framework. Service-Orientated approaches exist for performing discovery within environment without a centralised registry. It seems appropriate that future work also be given into investigating the application of this approach within an environment without middleware.

A Middleware-less Environment

One approach to distributed service discovery can be found within UPnP. In UPnP, service clients discover desired services by means of broadcasting discover messages (more specifically using SSDP). In porting this approach into a distributed environment, it becomes clear that this approach would require to be adapted to the environment. The ontology registry would require to be modified into a component capable of responding to broadcast discover messages. Similarly, the SQI part of the registry would require to be modified to convert between the domain-specific method of message passing (if one exists) and the querying format accepted by the registry.

Assuming the translation components are modified in a similar manner, the normal base of interactions can now take place. This approach is portable as the underlying principles of the approach are not platform or protocol specific. That is, the ontology registry implementation is confined to a Java-based execution environment, but the information which it manages is abstracted from this confinement. Similarly, the translation bundles are not required to operate on any specific platform, with the only constraint being that they can communicate with the ontology registry.

The two environments described are concerned with future work within extending and evaluating the implementation of this approach. This work would be useful in examining the limitations and requirements in ensuring this approach is not tightly bound to a specific middleware. Aside from this work, an interesting expansion of this approach may be found by describing abstract forms of service invocation and actions.

13.3.2 Abstract Actions

As earlier discussed,some network-like domains, such as web services, share a common description approach in order to increase interoperability. These domains may also contain a common communication protocol, such as SOAP, to further support the collaborative environment. As has been shown, in an environment where differing protocols already exist within the domain, an intervention approach is required. This approach is required to provide inclusion for existing network components to interoperate regardless of the protocols used. For this to be achieved, different parts of the network environment require to be abstracted from protocol boundaries.

This work has described an approach toward abstracting network component descriptions from specific domain vocabularies and approaches. Another feature of network interoperation which could potentially be abstracted is the service or device invocation stage. One solution to bridging protocol boundaries at the invocation level discussed in this work is using protocol drivers. These components bridge communication from one protocol into another. This approach still involves a large amount of protocol-specific implementation.

For example, a driver translates into a protocol A. A client of this driver requires to know how to interact with it, including what format the message should be passed in and what errors or state messages may be returned by the driver. Suppose the client wishes also to interact within protocol B. This requires additional knowledge of how to interact with the protocol B driver. Drivers A and B may require different messages values, which are appropriate for the respective domains. For example, suppose a client wishes to switch on two different lamps, one using protocol A and another using B. Protocol A requires a 'ON' action value, while B requires the value '1'. Thus, a client requires to know how to interact with both protocol drivers, which in turn may require the client to have some understanding of protocol-specific message types and values.

This issue can be partly resolved by protocol drivers sharing a common interface, with each driver being a protocol-specific implementation of the interface. Using an interface approach does not scale well, as it must take into account every possible action which can be carried out by a network component. For example, suppose the driver interface is represented in a Java environment. The Java object would require methods for each component action, which maybe become infeasible as new components emerge.

Instead of anticipating every possible action, it may be more feasible to represent the action in an abstract manner, independent of protocol-specific values and operations. For example, a switch on action could be represented in the following string: '*deviceid:LivingRoom1, action:power, values:status=on,;*'. In this environment, the vocabulary and structure used can be abstracted from the protocol domain. When a driver receives this string, it can interpret the intention of the client in a protocol specific domain. The driver would contain associations between the abstract vocabulary, and the protocol specific messages, actions and values relevant to the domain. These associations would be similar to those found within the ontology translation components.

In this manner, clients may interact with any protocol drivers without requiring any domain specific knowledge. Drivers can therefore decouple the invocation process from any domain specific format or approach. This approach would require a standardised format of the action string, as well as a universally applicable vocabulary. This requirement is not unlike the issues addressed by this work.

The area of abstract actions has already received a degree of investigation and implementation within policy based systems, such as that used within the MATCH project. In this environment, policies or rules can be formed to invoke or manipulate services and devices in the home. The policy system used within the MATCH project represents any service or device based actions in an abstract format. This abstract action is then converted into a protocol specific action by the system. The vocabulary used within the system is largely user and system specific, but the abstract action format is generic to any domain, as specified by the Appel policy system specifications.

The area of future work discussed within this section could be built upon the work done within the MATCH and Appel systems. This work could extend the scope of the approach into creating a standardised ontology-based vocabulary for representing actions and action values. In tandem with the description approach presented in this work, network components could be both described and used in a protocol-independent manner. Clients would be able to interact without any required knowledge of the existing protocols within the network.

13.4 Final Remarks

The approach presented in this work meets the aims stated. The approach allows the discovery process within home networks to be a natural process, removing the need for clients to require extensive amounts of terminology and inbuilt knowledge. The binding between generic and protocol specific terminologies is moved onto the middleware framework, allowing existing clients to continue to operate in future domains. New sets of vocabularies need not render existing discovery queries obsolete, as the ontology language allows semantically similar concepts to be logically related.

It is important to note that the discovery aspect of home networks is not the only issue which requires to be addressed. The discovery operation is merely the initial part of any network interoperation. This work attempts to reverse the common occurrence of new protocols and description frameworks being developed in isolation from existing work. In this manner, this work can be used as a foundation upon which true cross-protocol transactions can occur within the home.

References

[1] ESSI WSMO Working Group . Web Service Modeling Ontology. Technical report, The World Wide Web Consortium, June 2005.

[2] Carlisle Adams and Sharon Boeyen. Uddi and wsdl extensions for web service: a security framework. In *XMLSEC '02: Proceedings of the 2002 ACM workshop on XML security*, pages 30–35, New York, NY, USA, 2002. ACM.

[3] Heejune Ahn, Hyukjun Oh, and Chang Oan Sung. Towards reliable osgi framework and applications. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1456–1461, New York, NY, USA, 2006. ACM.

[4] Kathleen Ahrens, Siaw Fong Chung, and Chu-Ren Huang. Conceptual metaphors: ontology-based representation and corpora driven mapping principles. In *Proceedings of the ACL 2003 workshop on Lexicon and figurative language*, pages 36–42, Morristown, NJ, USA, 2003. Association for Computational Linguistics.

[5] Knarig Arabshian and Henning Schulzrinne. Combining ontology queries with key word search in the gloserv service discovery system. In *MC '07: Proceedings of the 2007 ACM/IFIP/USENIX international conference on Middleware companion*, pages 1–6, New York, NY, USA, 2007. ACM.

[6] Knarig Arabshian and Henning Schulzrinne. An ontology-based hierarchical peer-to-peer global service discovery system. *Journal of Ubiquitous Computing and Intelligence*, 1(2):133–144, December 2007.

[7] Steffen Balzer, Thorsten Liebig, and Matthias Wagner. Pitfalls of owl-s: a practical semantic web use case. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 289–298, New York, NY, USA, 2004. ACM.

[8] Sujata Banerjee, Sujoy Basu, Shishir Garg, Sukesh Garg, Sung-Ju Lee, Pramila Mullan, and Puneet Sharma. Scalable grid service discovery based on uddi. In *MGC '05: Proceedings of the 3rd international workshop on Middleware for grid computing*, pages 1–6, New York, NY, USA, 2005. ACM.

[9] Martin Bauer, Christian Becker, and Kurt Rothermel. Location models from the perspective of context-aware applications and mobile ad hoc networks. *Personal Ubiquitous Comput.*, 6(5-6):322–328, 2002.

[10] André Bottaro and Anne Gérodolle. Home soa -: facing protocol heterogeneity in pervasive applications. In *ICPS '08: Proceedings of the 5th international conference on Pervasive services*, pages 73–80, New York, NY, USA, 2008. ACM.

[11] Yérom-David Bromberg and Valérie Issarny. Indiss: interoperable discovery system for networked services. In *Middleware '05: Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, pages 164–183, New York, NY, USA, 2005. Springer-Verlag New York, Inc.

[12] Axel Brsch-Supan. Labor market effects of population aging. *Labour*, 17(SpecialIssue):5–44, 2003.

[13] Gavin A. Campbell. Ontology for Call Control. Technical Report CSM-170, Department of Computing Science and Mathematics, University of Stirling, UK, June 2006.

[14] Gavin A. Campbell. Ontology Stack for a Policy Wizard. Technical Report CSM-169, Department of Computing Science and Mathematics, University of Stirling, UK, June 2006.

[15] Pablo Cesar, Konstantinos Chorianopoulos, and Jens F. Jensen. Social television and user interaction. In *Comput. Entertain.*, volume 6, pages 1–10, New York, NY, USA, 2008. ACM.

[16] S. Chetan, J. Al-Muhtadi, R. Campbell, and M. D. Mickunas. Mobile gaia: a middleware for ad-hoc pervasive computing. In *Proceeings of The Second IEEE Consumer Communications and Networking Conference*, pages 223–228, 2005.

[17] Eleni Christopoulou, Christos Goumopoulos, and Achilles Kameas. An ontology-based context management and reasoning process for ubicomp applications. In *sOc-EUSAI '05: Proceedings of the 2005 joint conference on Smart objects and ambient intelligence*, pages 265–270, New York, NY, USA, 2005. ACM.

[18] Eleni Christopoulou, Christos Goumopoulos, Ioannis Zaharakis, and Achilles Kameas. An ontology-based conceptual model for composing context-aware applications. In *Workshop on Advanced Context Modeling, Reasoning and Management, 6 th Int. Conference on Ubiquitous Computing*, 2004.

[19] Eleni Christopoulou and Achilles Kameas. Gas ontology: an ontology for collaboration among ubiquitous computing devices. *Int. J. Hum.-Comput. Stud.*, 62(5):664–685, 2005.

[20] The OWL-S Coallition. *OWL-S: Semantic Markup for Web Services*, November 2004.

[21] Semantic Web Services Language Committee. Semantic Web Services Framework (SWSF) Overview. Technical report, The World Wide Web Consortium, September 2005.

[22] W3C Advisory Committee. Semantic web activity statement, October 2008.

[23] Lorcan Coyle, Steve Neely, Gatan Rey, Graeme Stevenson, Mark Sullivan, Simon Dobson, and Paddy Nixon. Sensor fusion-based middleware for assisted living. In *4th International Conference On Smart Homes & Heath Telematics*, pages 281–288, Belfast, UK, 26/06/2006 2006. IOS Press, IOS Press.

[24] Kiev Santos da Gama. An osgi middleware for mobile digital tv applications. In *Mobility '07: Proceedings of the 4th international conference on mobile technology, applications, and systems and the 1st international symposium on Computer human interaction in mobile technology*, pages 690–695, New York, NY, USA, 2007. ACM.

[25] Giliard Brito de Freitas and Cesar Augusto Camillo Teixeira. Ubiquitous services in home networks offered through digital tv. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1834–1838, New York, NY, USA, 2009. ACM.

[26] Jorge E. López de Vergara, Víctor A. Villagrá, Carlos Fadón, Juan M. González, José A. Lozano, and Manuel Álvarez Campana. An autonomic approach to offer services in osgi-based home gateways. *Comput. Commun.*, 31(13):3049–3058, 2008.

[27] Wolfgang Degen, Barbara Heller, Heinrich Herre, and Barry Smith. Gol: toward an axiomatized upper-level ontology. In *FOIS '01: Proceedings of the international conference on Formal Ontology in Information Systems*, pages 34–46, New York, NY, USA, 2001. ACM.

[28] Anind K. Dey and Jennifer Mankoff. Designing mediation for context-aware applications. *ACM Trans. Comput.-Hum. Interact.*, 12(1):53–80, 2005.

[29] Roman et Al. Web Service Modeling Ontology. *Applied Ontology*, 1:77–106, 2005.

[30] P. Fergus, M. Merabti, M. B. Hanneghan, Taleb-Bendiab, and Minghwan. A. A semantic framework for self-adaptive networked appliances. In *IEEE Consumer Communications & Networking Conference*, pages 229–234. IEEE Computer Society, 2005.

[31] P. Fergus, A. Minghwan, M. Merabti, and M. B. Hanneghan. Disus: Mobile ad hoc network unstructured services. In *Personal Wireless Communications.*, pages 75–82. ACTA Press, 2003.

[32] T. Finin, A. Joshi, L. Kagal, J. Niu, R. Sandhu, W. Winsborough, and B. Thuraisingham. Rowlbac: representing role based access control in owl. In *SACMAT '08: Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 73–82, New York, NY, USA, 2008. ACM.

[33] Organization for the Advancement of Structured Information Standards (OASIS). *Introduction to UDDI: Important Features and Functional Concepts*, October 2004.

[34] The Apache Software Foundation. River proposal, 2006 December.

[35] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[36] Adrian Friday, Nigel Davies, and Elaine Catterall. Supporting service discovery, querying and interaction in ubiquitous computing environments. In *MobiDe '01: Proceedings of the 2nd ACM international workshop on Data engineering for wireless and mobile access*, pages 7–13, New York, NY, USA, 2001. ACM.

[37] Eva Gahleitner, Wernher Behrendt, Jürgen Palkoska, and Edgar Weippl. On cooperatively creating dynamic ontologies. In *HYPERTEXT '05: Proceedings of the sixteenth ACM conference on Hypertext and hypermedia*, pages 208–210, New York, NY, USA, 2005. ACM.

[38] Alfonso Gárate, Nati Herrasti, and Antonio López. Genio: an ambient intelligence application in home automation and entertainment environment. In *sOc-EUSAI '05: Proceedings of the 2005 joint conference on Smart objects and ambient intelligence*, pages 241–245, New York, NY, USA, 2005. ACM.

[39] Nikolaos Georgantas, Sonia Ben Mokhtar, Yerom-David Bromberg, Valerie Issarny, Jarmo Kalaoja, Julia Kantarovitch, Anne Gerodolle, and Ron Mevissen. The amigo service architecture for the open networked home environment. In *WICSA '05: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, pages 295–296, Washington, DC, USA, 2005. IEEE Computer Society.

[40] Nubia M. Gil, Nicolas A. Hine, John L. Arnott, Julienne Hanson, Richard G. Curry, Telmo Amaral, and Dorota Osipovic. Data visualisation and data mining technology for supporting care for older people. In *Assets '07: Proceedings of the 9th international ACM SIGACCESS conference on Computers and accessibility*, pages 139–146, New York, NY, USA, 2007. ACM.

[41] P. D. Gray, T. McBryan, N. Hine, C. J. Martin, N. M. Gil, M. Wolters, N. Mayo, K. J. Turner, L. S. Docherty, F. Wang, and M. Kolberg. A scalable home care system infrastructure support domicilary care. Technical report, Department of Computing Science and Mathematics, University of Stirling, August 2007.

[42] RDF Core Working Group. RDF Vocabulary Description Language 1.0: RDF Schema. Technical report, The World Wide Web Consortium, February 2004.

[43] RDF Core Working Group. Resource Description Framework/W3C Semantic Web Activity. Technical report, The World Wide Web Consortium, 2004.

[44] RDF Data Access Working Group. *SPARQL Query Language for RDF*. W3C, January 2008.

[45] The Web Ontology Working Group. OWL Web Ontology Language Overview. Technical report, The World Wide Web Consortium, February 2004.

[46] Web Services Description Working Group. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. Technical report, The World Wide Web Consortium, June 2007.

[47] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199–220, 1993.

[48] Tao Gu, Hung Keng Pung, and Da Qing Zhang. Toward an osgi-based infrastructure for context-aware applications. *IEEE Pervasive Computing*, 3(4):66–74, 2004.

[49] Karthik Harihar and Stan Kurkovsky. Using jini to enable pervasive computing environments. In *ACM-SE 43: Proceedings of the 43rd annual Southeast regional conference*, pages 188–193, New York, NY, USA, 2005. ACM.

[50] HAVi, Inc. *The HAVi Specification: Specification of the Home Audio/Video Interoperability (HAVi) Architecture*, 1.1 edition, May 2001.

[51] Cristian Hesselman, Hartmut Benz, Pravin Pawar, Fei Liu, Maarten Wegdam, Martin Wibbels, Tom Broens, and Jacco Brok. Bridging context management systems for different types of pervasive computing environments. In *MOBILWARE '08: Proceedings of the 1st international conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications*, pages 1–8, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[52] Ian Horrocks and Peter F. Patel-Schneider. Three theses of representation in the semantic web. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 39–47, New York, NY, USA, 2003. ACM.

[53] Ian Horrocks and Peter F. Patel-Schneider. A proposal for an owl rules language. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 723–731, New York, NY, USA, 2004. ACM.

[54] Richard Hull and Jianwen Su. Tools for composite web services: a short overview. In *SIGMOD Rec.*, volume 34, pages 86–95, New York, NY, USA, 2005. ACM.

[55] Cycorp. Inc. The cyc knowledge base.

[56] Cycorp. Inc. Opencyc for the semantic web.

[57] Cycorp. Inc. Opencyc.org.

[58] Naohiko Kohtake, Jun Rekimoto, and Yuichiro Anzai. Infopoint: A device that provides a uniform user interface to allow appliances to work together over a network. *Personal Ubiquitous Comput.*, 5(4):264–274, 2001.

[59] Panu Korpipää, Jonna Häkkilä, Juha Kela, Sami Ronkainen, and Ilkka Känsälä. Utilising context ontology in mobile device application personalisation. *Proceedings of the 3rd international conference on Mobile and ubiquitous multimedia*, 1:133–140, 2004.

[60] V. Koufi, F. Malamateniou, and G. Vassilacopoulos. A medical diagnostic and treatment advice system for the provision of home care. In *PETRA '08: Proceedings of the 1st international conference on PErvasive Technologies Related to Assistive Environments*, pages 1–7, New York, NY, USA, 2008. ACM.

[61] Chin-Feng Lai and Yueh-Min Huang. Context-aware multimedia streaming service for smart home. In *Mobility '08: Proceedings of the International Conference on Mobile Technology, Applications, and Systems*, pages 1–5, New York, NY, USA, 2008. ACM.

[62] Meeyeon Lee, Seung Soo Park, and Jung-Won Lee. Ontology-based service layering for facilitating alternative service discovery. In *ICUIMC '08: Proceedings of the 2nd international conference on Ubiquitous information management and communication*, pages 465–470, New York, NY, USA, 2008. ACM.

[63] Lin Lin and I. Budak Arpinar. Discovering semantic relations between web services using their pre and post-conditions. In *SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing*, pages 237–238, Washington, DC, USA, 2005. IEEE Computer Society.

[64] David Liu and Dao Xian. Home environmental control system for the disabled. In *i-CREATe '07: Proceedings of the 1st international convention on Rehabilitation engineering & assistive technology*, pages 164–168, New York, NY, USA, 2007. ACM.

[65] Emerson Loureiro, Frederico Bublitz, Nadia Barbosa, Angelo Perkusich, Hyggo Almeida, and Glauber Ferreira. A flexible middleware for service provision over heterogeneous pervasive networks. In *WOWMOM '06: Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and Multimedia Networks*, pages 609–614, Washington, DC, USA, 2006. IEEE Computer Society.

[66] Lorena F. Maia, Danilo F. S. Santos, Ricardo S. Souza, Angelo Perkusich, and Hyggo Almeida. Seamless access of home theater personal computers for mobile devices. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 167–171, New York, NY, USA, 2009. ACM.

[67] D Marples and P. Kriens. The open services gateway initiative: an introductory overview. *Communications Magazine, IEEE*, 39:110–114, December 2001.

[68] Tony McBryan, Marilyn R. McGee-Lennon, and Phil Gray. An integrated approach to supporting interaction evolution in home care systems. In *PETRA '08: Proceedings of the 1st international conference on PErvasive Technologies Related to Assistive Environments*, pages 1–8, New York, NY, USA, 2008. ACM.

[69] Marilyn Rose McGee-Lennon. Requirements engineering for home care technology. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 1439–1442, New York, NY, USA, 2008. ACM.

[70] A. Mingkhwan, P. Fergus, O. Abuelma'Atti, M. Merabti, B. Askwith, and M. B. Hanneghan. Dynamic service composition in home appliance networks. *Multimedia Tools Appl.*, 29(3):257–284, 2006.

[71] Tatsuo Nakajima. Experiences with building middleware for audio and visual networked home appliances on commodity software. In *MULTIMEDIA '02: Proceedings of the tenth ACM international conference on Multimedia*, pages 611–620, New York, NY, USA, 2002. ACM.

[72] Ian Neild, Paul Bowman, and David Heatley. Sensor networks in telecare. In *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors*, pages 18–22, New York, NY, USA, 2007. ACM.

[73] 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.

[74] Confederation of British Industry. Effects of an Ageing UK Population: CBI evidence to the House of Lords Inquiry. http://www.cbi.org.uk.

[75] Institute of Cognitive Science and Italian National Research Council Technology. Laboratory for applied ontology - dolce.

[76] Sung Ho Park, Myung Jin Lee, and Soon Ju Kang. Multimedia room bridge adapter for seamless interoperability between heterogeneous home network devices. In *MG '08: Proceedings of the 15th ACM Mardi Gras conference*, pages 1–9, New York, NY, USA, 2008. ACM.

[77] Olena Parkhomenko, Yugyung Lee, and E. K. Park. Ontology-driven peer profiling in peer-to-peer enabled semantic web. In *CIKM '03: Proceedings of the twelfth international conference on Information and knowledge management*, pages 564–567, New York, NY, USA, 2003. ACM.

[78] Jyotishman Pathak, Neeraj Koul, Doina Caragea, and Vasant G. Honavar. A framework for semantic web services discovery. In *WIDM '05: Proceedings of the 7th annual ACM international workshop on Web information and data management*, pages 45–50, New York, NY, USA, 2005. ACM.

[79] Adam Pease. The suggested upper merged ontology(sumo) - ontology portal.

[80] Anand Ranganathan, Jalal Al-Muhtadi, and Roy H. Campbell. Reasoning about uncertain contexts in pervasive computing environments. *IEEE Pervasive Computing*, 3(2):62–70, 2004.

[81] Nirmalya Roy, Gautham Pallapa, and Sajal K. Das. An ontology-driven ambiguous contexts mediation framework for smart healthcare applications. In *PETRA '08: Proceedings of the 1st international conference on PErvasive Technologies Related to Assistive Environments*, pages 1–8, New York, NY, USA, 2008. ACM.

[82] Michele Ruta, Tommaso Di Noia, Eugenio Di Sciascio, Massimo Paolucci, Floriano Scioscia, and Eufemia Tinelli. A semantic-based registry enabling discovery, composition and substitution of pervasive services. In *MobiDE '08: Proceedings of the Seventh ACM International Workshop on Data Engineering for Wireless and Mobile Access*, pages 63–70, New York, NY, USA, 2008. ACM.

[83] Marta Sabou, Chris Wroe, Carole Goble, and Gilad Mishne. Learning domain ontologies for web service descriptions: an experiment in bioinformatics. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 190–198, New York, NY, USA, 2005. ACM.

[84] Stephen Strom. Building a large-scale generic object model: applying the cyc upper ontology to object database development in java. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 37–38, New York, NY, USA, 2000. ACM.

[85] Ali Maleki Tabar, Arezou Keshavarz, and Hamid Aghajan. Smart home care network using sensor fusion and distributed vision-based reasoning. In *VSSN '06: Proceedings of the 4th ACM international workshop on Video surveillance and sensor networks*, pages 145–154, New York, NY, USA, 2006. ACM.

[86] Andrea Taylor, Richard Wilson, and Stefan Agamanolis. Supporting carers in their caring role through design. In *CHI EA '09: Proceedings of the 27th international conference extended abstracts on Human factors in computing systems*, pages 3985–3990, New York, NY, USA, 2009. ACM.

[87] Linda Tetzlaff, Michelle Kim, and Robert J. Schloss. Home health care support. In *CHI '95: Conference companion on Human factors in computing systems*, pages 11–12, New York, NY, USA, 1995. ACM.

[88] Alessandra Toninelli, Antonio Corradi, and Rebecca Montanari. Semantic-based discovery to support mobile context-aware service access. *Comput. Commun.*, 31(5):935–949, 2008.

[89] K. J. Turner, L. S. Docherty, F. Wang, and G. A. Campbell. Managing Home Care Networks. In R. Bestak, L. George, V. S. Zaborovsky and C. Dini, editor, *Proc. 8th Int. Conf. on Networks (ICN'09)*, pages pp. 354–359. IEEE Computer Society, March 2009.

[90] UPnP Forum. *UPnP Device Architecture 1.1*, October 2008.

[91] Upkar Varshney. Pervasive healthcare and wireless health monitoring. In *Mob. Netw. Appl.*, volume 12, pages 113–127, Hingham, MA, USA, 2007. Kluwer Academic Publishers.

[92] Antonio Vilei, Gabriella Convertino, and Fabrizio Crudo. A new upnp architecture for distributed video voice over ip. In *MUM '06: Proceedings of the 5th international conference on Mobile and ubiquitous multimedia*, page 2, New York, NY, USA, 2006. ACM.

[93] Jim Waldo. The jini architecture for network-centric computing. In *Commun. ACM*, volume 42, pages 76–82, New York, NY, USA, 1999. ACM.

[94] Agustinus Borgy Waluyo, Isaac Pek, Xiang Chen, and Wee-Soon Yeoh. Design and evaluation of lightweight middleware for personal wireless body area network. In *Personal Ubiquitous Comput.*, volume 13, pages 509–525, London, UK, 2009. Springer-Verlag.

[95] F. Wang, L. S. Docherty, K. J. Turner, M. Kolberg, and E. H. Magill. Service and Policies for Care At Home. In J. C. Chachques J. E. Bardram and U. Varshney, editors, *Proc. 1st Int. Conf. on Pervasive Computing Technologies for Healthcare*, pages pages 7.1–7.10. Institution of Electrical and Electronic Engineers Press, New York, USA, November 2006.

[96] Feng Wang and Kenneth J. Turner. Towards personalised home care systems. In *PETRA '08: Proceedings of the 1st international conference on PErvasive Technologies Related to Assistive Environments*, pages 1–7, New York, NY, USA, 2008. ACM.

[97] Hai Wang, Zengzhi Li, and Lin Fan. An unabridged method concerning capability matchmaking of web services. In *WI '06: Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence*, pages 662–665, Washington, DC, USA, 2006. IEEE Computer Society.

[98] Xiao Hang Wang, Da Qing Zhang, Tao Gu, and Hung Keng Pung. Ontology based context modeling and reasoning using owl. In *PERCOMW '04: Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*, page 18, Washington, DC, USA, 2004. IEEE Computer Society.

[99] Xiaohang Wang, Jin Song Dong, ChungYau Chin, SankaRavipriya Hettiarachchi, and Daqing Zhang. Semantic space: An infrastructure for smart spaces. *IEEE Pervasive Computing*, 3(3):32–39, 2004.

[100] The Web Services Interoperability Organization (WS-I). The Web Services - Interoperability (WS-I).

[101] XML Protocol Working Group. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*, April 2007.

[102] Kai Yang and Robert Steele. An ontology mediated web service aggregation hub. In *WI '07: Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*, pages 572–576, Washington, DC, USA, 2007. IEEE Computer Society.

[103] Qi Yu, Xumin Liu, Athman Bouguettaya, and Brahim Medjahed. Deploying and managing web services: issues, solutions, and directions. In *The VLDB Journal*, volume 17, pages 537–572, Secaucus, NJ, USA, 2008. Springer-Verlag New York, Inc.

[104] Thomas G. Zimmerman and Keng-hao Chang. Simplifying home health monitoring by incorporating a cell phone in a weight scale. In *PETRA '08: Proceedings of the 1st international conference on Pervasive Technologies Related to Assistive Environments*, pages 1–4, New York, NY, USA, 2008. ACM.